Contention-Aware Performance Monitoring Counter Support for Real-Time MPSoCs

Javier Jalle[†,*], Mikel Fernandez[†], Jaume Abella[†], Jan Andersson[⋆],
Mathieu Patte[∓], Luca Fossati[§], Marco Zulianello[§], Francisco J. Cazorla[†,‡]

[†]Barcelona Supercomputing Center [*]Universitat Politècnica de Catalunya [§]European Space Agency
[⋆]Cobham Gaisler [∓]Airbus Defense and Space [‡]Spanish National Research Council (IIIA-CSIC)

***Abstract*—Tasks running in MPSoCs experience contention delays when accessing MPSoC's shared resources, complicating task timing analysis and deriving execution time bounds. Understanding the Actual Contention Delay (ACD) each task suffers due to other corunning tasks, and the particular hardware shared resources in which contention occurs, is of prominent importance to increase confidence on derived execution time bounds of tasks. And, whenever those bounds are violated, ACD provides information on the reasons for overruns. Unfortunately, existing MPSoC designs considered in real-time domains offer limited hardware support to measure tasks' ACD losing all these potential benefits. In this paper we propose the Contention Cycle Stack (CCS), a mechanism that extends performance monitoring counters to track specific events that allow estimating the ACD that each task suffers from every contending task on every hardware shared resource. We build the CCS using a set of specialized low-overhead Performance Monitoring Counters for the Cobham Gaisler GR740 (NGMP) MPSoC – used in the space domain – for which we show CCS's benefits.**

I. Introduction

As part of the validation and verification process of Critical Real-Time Embedded Systems (CRTES)[1] functional and temporal requirements need to be assessed to obtain enough evidence about the proper operation of the CRTES. Testing, which is an integral part of the validation and verification process, is intended to find both temporal violations (i.e. a function overruns its assigned time budget) and functional bugs (i.e. a given function does not perform its work). The absence of errors during the testing phase increases the confidence on the system correct behavior. In this paper we focus on the temporal validation and verification of CRTES, which requires deriving execution time bounds for software units – also referred to as Worst-Case Execution Time (WCET) estimates. The most common method used to obtain those bounds is measurement-based timing analysis [42]. In processor architectures with limited complexity the challenge lies in finding a set of (program) inputs that lead to the WCET. Tools have been developed and qualified to help on this task. For instance, Rapita's Verification Suite [1] can be used to test the achieved execution-path coverage with the user provided inputs. Of course, the level of rigour required varies depending on the criticality defined for the function under analysis (e.g. modified condition/decision coverage is required for DAL A functions under DO-178C [39]).

In recent years CRTES are witnessing an unstoppable transition towards MultiProcessors System-on-Chip (MPSoC) – featuring cache memories and multicores – to respond to the increased performance requirements of CRTES in domains such as avionics, space and automotive. This, in turn, is required for CRTES to cope with more sophisticated value-added software functionalities [41]. MPSoC promised benefits come at the cost of complicating CRTES temporal verification. In particular the contention between tasks in MPSoCs hardware shared resources has been acknowledged as one of the most complex elements for temporal analysis [3][30]. This occurs because the load a given task puts on hardware shared resources affects its co-runners and vice versa.

For static timing analysis, despite its mathematical rigour, its correctness rests on the quality of user-supplied data. For instance, the correctness of the information about hardware's internal functioning – used to derive timing models of the hardware – is heavily challenged in MPSoCs [3]: hardware is so complex that even the information provided in processor reference manuals is subject to several errata releases as undiscovered timing behavior appears [18][5]. In other cases, the lack of information of the impact of contention in hardware shared resources makes industry rely on measurement-based techniques to derive such information [30] and provide it as input to the static timing analysis tool. Measurement-based timing analysis also faces several difficulties, in particular capturing in the performed tests all the hardware/software factors impacting execution time, which is increasingly becoming an unaffordable task [3]. The dependence of execution time on the task under analysis, its co-runner tasks and the shared resource access policies can potentially cause a state explosion in the set of experiments to perform.

It follows that no WCET technique provides full confidence on derived execution time bounds for MPSoC. In particular, the estimated bounds for the Worst Contention Delay (WCD) in hardware shared resources are exposed to several inaccuracies that decrease the confidence on them [13]. In this scenario, confidence can only be regained via proper testing in the validation and verification phase. Whenever a task overruns it is key determining whether the overrun is caused by the contention on the hardware shared resources or it is due to the application intrinsic (in isolation) behavior. In the former case, one would want further information on the resource where the contention delay is taking place and which of its contenders is causing it. This would be very valuable information for validation, verification – acknowledged as time and effort consuming steps – and optimization purposes.

This paper makes the case for the Actual Contention Delay (ACD) metric and the Contention Cycle Stack (CCS) approach to improve the temporal-related testing of MPSoCs. ACD captures the time tasks spend stalled in a shared resource due to contention with their co-runner tasks. The CCS is a stacked representation of a task's ACD to understand the particular contention time the task spends in each shared resource. For each such resource the CCS also allows determining the contending task causing the contention. The use

[1]In CRTES Criticality relates to safety, availability, security, mission or business needs of the system.

of CCS brings benefits at different abstraction levels, both during the integration tests and once the system is deployed. This includes (i) determining whether a task overruns due to (unnoticed) systematic hardware failures or software faults; or (ii) the overrun is caused by inaccuracy of the WCD used by the timing analysis tool; and (iii) optimizing energy usage and average performance by scheduling tasks such that ACD is reduced. Overall, our contributions in this paper can be summarized as follows:

1) We make an in-depth analysis of the Performance Monitoring Counters (PMC) provided in several processor chips targeting high-performance and real-time domains including designs by IBM, Intel, Freescale, ARM and Gaisler. Rather than providing an exhaustive list of PMCs, we focus on the events related with contention that PMCs can track. Our analysis reveals that, despite some of the studied architectures having hundreds of PMCs, they are not meant to track ACD, which prevents deriving the CCS in a cost-effective manner.

2) With focus on the GR740 processor – which implements the latest NGMP (Next Generation Multipurpose Processor) architecture and which is planned to be used by the European Space Agency in future missions – we propose a new set of low-overhead CCS-aware PMCs for the two main shared resources in the GR740: the AHB AMBA bus and the memory controller (note that the L2 cache is partitioned using the hardware support provided by the GR740, so tasks suffer no contention delay in this resource).

3) We evaluate our proposal in two solid setups. First, a GR740 simulator whose accuracy has been tested to be 97% at cycle level for EEMBC and real space applications against an ASIC implementation. And second, the CCS-aware PMCs for the AMBA bus are implemented in a FPGA with a real version of the GR740. For both setups, our results show that CCS-aware PMCs effectively capture the contention delay suffered by each task which we present with CCS. We also show that our proposal incurs low hardware cost since it reuses the available PMC infrastructure in the GR740, which is multiplexed for different event counts, adding only 16 new events and its corresponding wiring on the FPGA.

Overall, CCS increases the confidence on the bounds of tasks' WCD, and hence the CCS becomes an instrumental means for the testing of CRTES – arguably the only way to consider increasing dynamic contention in MPSoCs. In case of a task overrun, CCS allows to ascertain whether this is due to an inaccuracy in the model or it is due to other systematic behaviors intrinsic to the task or the hardware. Finally, CCS also enables other scheduling optimizations.

The rest of this paper is organized as follows: Section II presents the problem attacked and the expected results. Sections III presents a taxonomy of the PMC in the architectures studied in this paper based on the detailed analysis done in Section VI. Section IV shows our proposal of CCS aware PMCs. Section V evaluates our proposal. Section VII presents the most relevant related work. Finally, Section VIII presents the main conclusions of this paper.

II. The Contention Cycle Stack

During the analysis phase of the system, bounds are derived to the WCD that a task can suffer. At deployment (operation) tasks suffer delays due to contention, which we call actual contention delay (ACD). It is worth nothing that, while several works focus on deriving bounds to WCD, understanding the ACD that tasks suffer has been barely studied in the literature. In this paper we show the benefits that deriving ACD brings for validation/verification and optimization purposes.

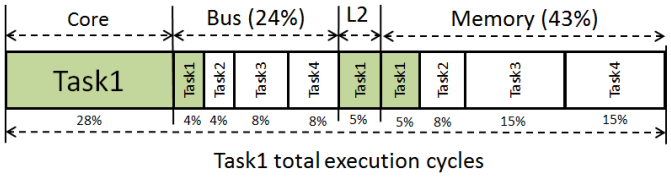

Fig. 1: Synthetic CCS for a given Task1 on a quad-core. Note that we usually represent the CCS vertically, though for space constraints this figure presents it horizontally.

A. Introduction to the CCS

Figure 1 shows a synthetic example of the CCS of a given task[2], *Task1*, derived for a 4-core processor with a shared bus and memory (the L2 is split per core), when *Task1* runs in a workload with *Task2*, *Task3* and *Task4*. Under each CCS component we stack[3] all the cycles that *Task1* spends on a given resource. The *core component* covers the time *Task1* spends executing locally in the core (28% in the example).

For the off-core resources, the CCS breaks down per task the time in each component. For instance, the bus component covers the cycles, called *working cycles*, in which the processor is stalled due to the processing of a request in the bus, labelled as *Task1*; and the cycles in which *Task1* is waiting to get access to the bus while Task2, Task3 and Task4 are using it, called *contention cycles*. In the figure, we observe that Task1 is stalled during 24% of its overall execution time due to the bus: 4% of the cycles to handle its own requests, while 4% waiting for Task2, 8% for Task3 and 8% for Task 4. Likewise, the CCS also provides information about Task1's working and contention cycles in the L2 cache and memory controller. The shaded elements in each component represent Task1's working cycles, i.e. its behavior in isolation, while the other elements provide information about the time Task1 spends stalled in each shared resource due to contention caused by each contending task.

B. Applicability

Temporal Validation and Verification. Deriving bounds to the contention delay in MPSoCs, either by building timing models to feed static timing analysis tools or using measurements, is challenging. In both cases the complexity of MPSoCs affects the confidence one can put on the derived bounds.

For timing models, one could infer the contention that requests accessing a shared resource cause on each other from the reference documentation. However, manuals are becoming multi-thousand documents so extracting timing information is an error-prone process. As a matter of example, some parts of the Freescale P4080 specification have 2,000 pages [17] and the Infineon XMC4500 microcontroller documentation has more than 2,500 pages [21]. Even if the chip vendor explicitly provides this information in the manuals, it is the case that MPSoCs documentation can be inaccurate or outdated with respect to the deployed chip implementation [3]. For instance, the FreeScale e500mc core documentation has already reached the third revision with details about non-negligible changes

[2]In this paper, for the sake of simplicity, we consider a correspondence between task and core, i.e., *TaskX* runs in CoreX.

[3]Cycles in each component do not occur consecutively during the execution of the task. In the CCS we *stack* those cycles (and show them) consecutively.

TABLE I: Analysis of whether each processor contains PMC in each of the categories identified in our taxonomy (Detailed description provided in Section VI).

	Cycle Count			Event Count				Data Count	Current Events
	Use	Busy	Idle	Use	Busy	Threshold	Inst. Type		
IBM	✓	✓	✓	✓	✓	✗	✓	✓	✗
Intel	✓	✓	✓	✓	✓	✓	✓	✗	✓
ARM	✓	✓	✗	✓	✓	✗	✓	✗	✗
Freescale	✓	✓	✗	✓	✓	✓	✓	✗	✗
GR740	✓	✓	✓	✓	✓	✓	✓	✗	✗

across revisions [18]. Similarly, the documentation of processors such as the ARM Cortex R5 – specifically targeting CRTES – have abundant errata [18][5] despite being relatively simple. All these difficulties have made that real-time industry and static timing analysis tool providers use measurement-based approaches to derive contention bounds [30].

With the goal of deriving bounds to WCD, several measurement-based techniques exist that use specialized kernels, called resource stressing kernels (*rsk*) [14], which aim at putting high, ideally the highest, load on shared resources. Those *rsk* are used to put a given task under analysis under stressful contention scenarios. When there are several shared resources, it is virtually impossible to design a *rsk* that puts the highest load on all of them simultaneously [34]. Hence, it is hard – if at all possible – to determine from the execution time whether, under a given workload, a task hits the WCD in one shared resource and not in another or whether the WCD was not hit for any of the resources.

The CCS allows determining the ACD a task suffers per resource (and per task) which enables verifying and validating contention bounds by determining how close is the ACD to the theoretical WCD. The CCS can be used in those scenarios to provide evidence about the validity of contention time bounds – increasingly derived with measurements [30][13] – such that the contention suffered by the task in the workload matches the worst possible latency it can suffer [31]. Also, in case a bound is exceeded, the CCS can detect it and identify the reason behind it. For instance, assuming that Task 1 in Figure 1 had an overrun due to Tasks 3 and 4 memory contention, the CCS allows identifying this situation by providing the information that memory contention is 38%, having Tasks 3 and 4 a 15% of impact each. CCS also provides the execution time in isolation just by removing the ACD. For instance, if Tasks 2, 3 and 4 are removed from the CCS in Figure 1, what remains is the execution time of Task 1 when no other core is running.

Optimization. The CCS provides valuable information that can be used for optimization purposes. For instance, the CCS allows identifying, in the example in Figure 1 that Task 1 spends 43% of its execution time in memory, 30% of which is due to interference from Tasks 3 and 4. In such a case system designers can either reduce memory accesses from Task 1 by improving cache locality of accesses or prevent Tasks 3 and 4 from being scheduled together with Task 1. Such optimizations bring the advantage of having more slack in the schedule, so that other tasks can also be consolidated onto the same hardware platform, or some energy can be saved.

To obtain the CCS we need detailed information of the processor events affecting its timing, which is obtained through its PMC. In the next section, we analyze those PMC available in some MPSoCs useful for measuring contention.

III. Taxonomy of PMCs in Real Multi-Core Processors

We have analyzed in detail the available information about the PMCs of the IBM POWER7, the latest Intel architectures, the ARMv7-A, the Freescale P4080 and the GR740. Our goals are identifying the type of events that can be tracked with those counters and whether this can be used to build the CCS, rather than presenting an extensive list of all counters of the analyzed architectures – that in several cases exceed 500. It is worth noting that, since our focus is on contention interference effects, we analyze the trackable events on hardware shared resources. From this perspective, the events happening inside the computing cores private resources (e.g. pipeline and local caches) are of no interest.

From our analysis we have produced a PMC taxonomy and have derived some conclusions on PMC that hold across all platforms. For readability reasons, we defer a detail explanation of the PMC support in each architecture to Section VI and we focus on this section on the result of the analysis, i.e. the taxonomy which is shown in Table I.

Cycle count covers those counters that measure time (cycles) . This is further broken down into the following. *Use*: number of cycles that a resource is in use; *Busy*: number of cycles that a resource is unavailable and causes a core stall. And *Idle*: number of cycles that a resource remains unused.

Event count category groups those counters that measure the number of times an event happens. This category is further divided into the following. *Use*: number of times a resource has been accessed; *Busy*: number of times a resource has been unavailable because it was busy; *Threshold*: number of times a set threshold has been exceeded; And *Instructions by type*: number of times an instruction of a given type is executed.

Data count covers the counters that measure the amount of data transferred or managed.

Current events groups those counters that provide the current status of the processor.

We find few counters that help understanding to some extent the effect of contention interferences:

POWER7: The PMC PMC-CMPLU-STALL-THRD provides the number of cycles a task is stalled due to inter-task interferences in the reorder buffer (a.k.a. as Global Completion Table). No information on contention in core-external shared resources is provided.

Intel: The MEM-TRANS-RETIRED.LOAD-LATENCY counter can be configured to track the number of times memory load operations latencies exceed a user defined threshold. This allows the user to approximate the worst observed behavior, and to upper bound it. However, such counter neither measures actual contention nor identifies the reasons behind such contention.

GR740: PMCs can be configured in *maximum count mode*. While in this mode, the counter keeps the maximum amount

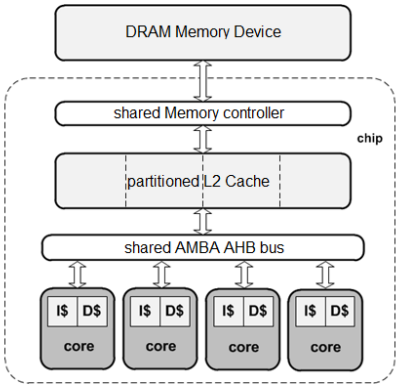

Fig. 2: GR740 Processor architecture.

of time the selected event has been asserted. It is also possible in this mode to count the maximum amount of time between two event assertions. Maximum count mode might be useful as a first step to implement CCS PMCs. Using Maximum Count Mode it could be possible, for example, to count the longest burst of bus cycles, or the longest amount of time the bus has been without having a read access. As in the case of Intel and IBM counters, these counters are neither designed to measure actual contention nor to identify the cause of such contention.

In all the studied architectures, PMCs are used to improve average system performance by monitoring software execution, characterizing processor behavior, and/or helping system developers bring up and debug their systems, but their focus is not monitoring contention or interferences across tasks, so they do not help deriving the CCS. We note that some information about contention interference can be derived in controlled scenarios by means of experimentation. For instance, in a first experiment the program under study is run in isolation recording *cycle count* PMC readings. In a subsequent set of experiments the program under study is run again as part of a workload. By subtracting the PMCs in the first run from those in the second run some inter-task interference information can be obtained, though likely it will not be precise on which shared resource and which task caused it. Furthermore, this process is complex due to reproducibility issues and, in some cases, because the system cannot be used to carry out those extra runs. For instance, if a deadline is missed in a fail-safe system (1) it may be difficult – if at all possible – to reproduce the scenario that led to the deadline miss and (2) the system cannot be used to run some experiments because it would jeopardize its availability given that its operation needs to be quickly resumed.

IV. PMC PROPOSAL FOR OBTAINING CCS

The CCS provides a representation of the working and contention cycles of each task (τ_i) running in a multicore. Tasks spend some processing cycles at core level[4], p_i, and some others accessing core-external hardware shared resources ($\mathcal{R}$) that cause cores to stall and consume cycles, s_i. For each shared resource $r \in \mathcal{R}$, stall cycles are broken down into working cycles, w_i^r, and contention cycles, c_i^r. The former corresponds to the cycles τ_i spent actually using the resource, as it happens when running in isolation, i.e. without contention. The latter covers the cycles in which τ_i was stalled due to some inter-task (contention) interference activity generated by another contending task τ_j. In our reference architecture, the CCS can be expressed as shown in Equation 1:

$$t_i = p_i + s_i = p_i + \sum_{r \in \mathcal{R}} (w_i^r + c_i^r) \tag{1}$$

Let $c(\tau_i)$ be the (contending) tasks executing at a given point in time with a given task τ_i. The number of tasks in $c(\tau_i)$ varies from 0 to $Nc-1$, i.e. the number of cores minus one. For τ_i, its CCS for each resource c_i^r, can be further broken down so that it provides information about the contention each of its contenting tasks cause on τ_i, called $c_{i \leftarrow j}^r$.

$$c_i^r = \sum_{\tau_j \in c(\tau_i)} c_{i \leftarrow j}^r \tag{2}$$

By combining Equation 2 and Equation 1, the CCS can be expressed as shown in Equation 3:

$$t_i = p_i + \sum_{r \in \mathcal{R}} \left(w_i^r + \sum_{\tau_j \in c(\tau_i)} c_{i \leftarrow j}^r \right) \tag{3}$$

In this paper we explore the cost and benefits of the CCS in the Cobham Gaisler GR740. This is a representative MPSoC planned to be used for years by the European Space Agency in future on-board systems in space missions.

A. Cobham Gaisler GR740

The GR740, sketched in Figure 2, is a system-on-chip with four 32-bit LEON4 SPARCv8 processors. Each core comprises private instruction (IL1) and data (DL1) caches, and write buffers. Cores do not stall unless waiting for either a read miss in any of the L1 caches or write requests that find the write buffer full. In both cases, the core accesses the L2 cache through a shared bus in which it either gets the bus granted and transfers the request or waits for other cores accessing the bus. DL1 is write-through, L2 write-back and all caches use LRU replacement policy. The bus connecting the cores to the L2 cache uses a round-robin arbitration scheme.

The GR740 implements hardware mechanisms to partition the L2 cache so that each core can only access its assigned cache ways. Hence, there is no contention once a request arrives at the L2 and all cycles spent in the L2 are working cycles. If the request misses in L2, it accesses memory. The memory controller behaves as a FIFO queue, with the request on top accessing memory, i.e., consuming working cycles, and the other requests in the queue waiting, i.e., consuming contention cycles.

B. CCS for the GR740

The GR740 comprises three main on-chip hardware shared resources, the memory, L2 cache and the bus, so $\mathcal{R} = \{mem, L2, bus\}$. Building the CCS for the GR740 requires deriving the *processing cycles* (p_i) for the core where the task under analysis is, *working cycles* (w_i^r) for each shared resource and *contention cycles* ($c_{i \leftarrow j}^r$) for each contending task in the bus and the memory. The L2 cache is a special case because it is partitioned, which means that contention on the access to it is removed, i.e., $c_{i \leftarrow j}^{L2} = 0$ for all τ_i, τ_j.

[4]We include in this category also cycles in which the pipeline is stalled (no instruction can be fetched) due to a local stall, for instance a floating-point operation blocking the processor due to its long latency.

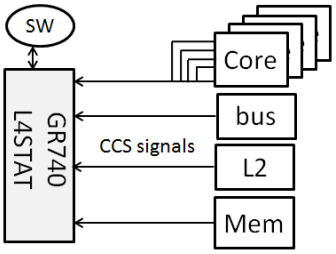

Fig. 3: CCS module and input/output signals.

C. Processing and stall cycles

We deploy existing PMCs in the GR740 to derive p_i. In particular, we use the *execution cycles*, t_i, and the *stall cycles*, s_i to compute processing cycles as $p_i = t_i - s_i$. Execution cycles t_i are obtained from existing PMC (processor event *time*:`0x15` [11]). When deriving s_i it is worth noting that the stall cycles are obtained as a combination of three existing PMCs since the cycles that the processor is stalled, s_i, are caused by the i)IL1, ii)DL1 or iii)the write-buffer when they wait for a request to be completed outside of the core. These three events can be directly measured from their respective available PMCs (processor events *ichold*:`0x02`, *dchold*:`0x0A` and *wbhold*:`0x10` [11]) and their addition gives s_i.

D. Working and contention cycles

Our extension to the GR740 PMC infrastructure consists in adding new low-overhead PMCs that allow accounting for w_i^r and $c_{i \leftarrow j}^r$ for the bus and the memory. L2 working cycles, w_i^{L2}, can be obtained indirectly from available PMCs. Since hit and miss latencies are known, and available in the documentation [11], an estimate of the time spent in the L2 cache can be derived using these latencies and the number of hits and misses, which can be directly measured from their respective PMCs (events l2hits:`0x60` and l2miss:`0x61` [11]). If L2 latencies were not available in the documentation, L2 working cycles could be easily obtained as the remaining execution time cycles, since all the cycles, except L2 cycles, can be accurately classified as either core, bus or memory cycles. Note that all L2 cycles are working cycles, because there is no contention in the L2.

For the bus and the memory, working (and contention) cycles cannot be indirectly obtained from available PMCs. For instance, the GR740 provides PMCs to count the number of bus accesses, however, since the latency per access is not fixed – and in some cases depends on other components [23] – *working* and *contention cycles* cannot be derived.

We propose extending the available processor's statistics unit, called L4STAT in the GR740, with new events needed by the CCS, called CCS module (CCSm). Figure 3, sketches the CCSm and how it is connected to other hardware blocks in the GR740. Each shared resource provides two pieces of information to the CCSm on every cycle: the core that is using the resource, which allows tracking the working cycles, and the cores that are waiting for the resource, which allows tracking if a core is interfered and in that case, the actual core using the resource is designated as the interferer.

It is worth noting that certain resources inside the core, such as the write-buffer, can hide the latency of some requests. This makes that even when there are outstanding requests in the bus or in memory, the processor core is not necessarily stalled, since the write-buffer can hide that latency. The CCSm requires identifying the cycles that the processor is stalled to know if it should account cycles in shared resources, either as *working* or *contention* cycles. Otherwise there would be more accounted cycles than cycles spent in reality, thus reducing the accuracy of the CCS.[5] For instance, if the write-buffer hides the latency of a write request, even if that request is using the bus, the core is working, thus consuming processing cycles and those cycles should not be accounted for the bus. For that purpose, the core should provide a *stalled* signal to the CCSm, indicating whether the core is stalled waiting for a request or not. As mentioned at the beginning of this section, the *stalled* signal can be easily obtained as a combination of the three *trigger signals* for the PMCs that measure the cycles that IL1, DL1 or the write-buffer are waiting for a request, which correspond to the stall cycles of the core, as seen before.

Although in theory some events could overlap, and so some cycles could be accounted twice, this occurs seldom in practice. In particular, a core may have two requests in flight simultaneously (one in the bus and another in the memory controller). However, this can only happen if the first request is a write operation – so it does not stall the core – and the second one a read operation. In this case, since the write operation cannot stall the execution, any stall due to contention or access delay is accounted to the second request, since it is the only one that can impact execution time. Still, our results show that the frequency of occurrence of that combination of events is negligible in practice.

E. Shared AMBA AHB bus

One of the most important hardware shared resources in the GR740 (and in many other MPSoC for real-time systems) is the backbone bus, since it connects the different cores with the memory/cache subsystem (and possibly other devices or subsystems). The Advanced Microcontroller Bus Architecture (AMBA) [6] is one of the most – if not the most – broadly used bus interface. AMBA is used in a wide range of architectures, providing flexibility in the implementation and backward-compatibility with existing AMBA interfaces. AMBA AHB is used in multicore processors for real-time industry, e.g. LEON3-based GR712RC [9] and LEON4-based GR740 [11].

A recent study [23] shows that a single AMBA request from a given task can block other tasks' accesses to the bus for long periods. This reinforces the need to have the CCS for the bus. In [23] authors also talk about the high cost that changing the AMBA interface would incur, specially regarding compatibility and development/usage of third-party intellectual-property cores. *Therefore, our PMC support for the AHB AMBA bus must not require any change in its interface.*

Under the AMBA protocol, the arbitration process involves several hardware blocks (the arbiter and one or several masters) and several signals. We focus on two of these signals: $\texttt{HBUSREQ}_i$ and $\texttt{HGRANT}_i$. Master m_i asserts $\texttt{HBUSREQ}_i$ to indicate the arbiter that it is requesting the bus. The arbiter asserts $\texttt{HGRANT}_i$ when it grants access to m_i, according to its arbitration policy (not specified by AMBA protocol, though in our case is round-robin [11]). As an illustrative example, Figure 4 shows one arbitration process[6] for m_1. Once master (core) 1 is ready to send a request, it asserts the

[5]This also stands for the counters that count L2 hit/miss, counting only when the processor is stalled. Otherwise, original PMCs would account hits/misses for write requests that do not stall the processor.

[6]The timing on the figure is an abstraction of the real AMBA timing, see section 3.11 of [6] for details.

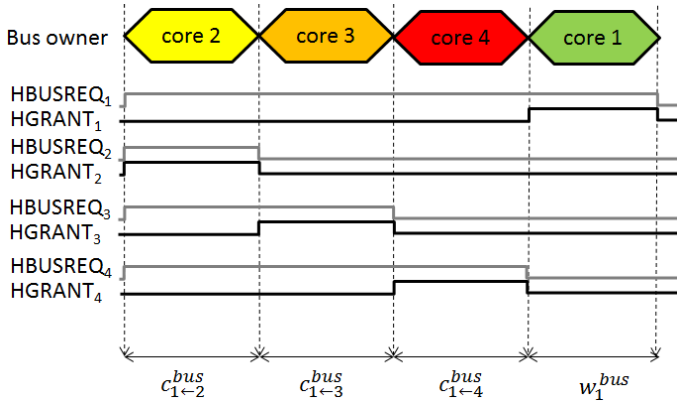

Fig. 4: Simplified AMBA arbitration process.

$\texttt{HBUSREQ}_1$ signal. At that point in time, core 2 is using the bus, since $\texttt{HGRANT}_2$ is active. In this example, let us assume that core 3 and core 4 are also waiting for the bus ($\texttt{HBUSREQ}_3$ and $\texttt{HBUSREQ}_4$ are active) and according to the round-robin arbitration policy they both have higher priority than m_1 at this point. The figure shows how the grant is passed from m_2 to m_3 and m_4 respectively. When the arbiter grants access to core 1, it sets the $\texttt{HGRANT}_1$ signal for m_1.

We propose to forward $\texttt{HBUSREQ}_i$ and $\texttt{HGRANT}_j$ signals from the arbiter to the CCSm. By checking these signals the CCSm infers on each cycle which master m_j is using the bus, thus the working cycles w^{bus}_j, and whether another master m_i is waiting for master m_j, i.e. the contention cycles $c^{bus}_{i\leftarrow j}$. Our proposal maintains the same bus interface, since no signals are introduced or modified. We simply forward the existing signals to the CCSm as shown in Figure 5. The CCSm has $N_c \times N_c = 16$ Bus Contention Counters ($bcc_{i,j}$). $bcc_{i,j}$ stores the number of cycles that $(\texttt{HBUSREQ}_i)$ & $(\texttt{HGRANT}_j)$ holds. $bcc_{i,j}$ where $i \neq j$ hold the contention cycles that master i suffers from master j, i.e. $c^{bus}_{i\leftarrow j}$. Counters $bcc_{i,i}$ store the working cycles that master m_i uses to process its requests, i.e, w^{bus}_i. In the bottom part of Figure 4 we show how cycles are accounted for core 1.

It is worth noting that the $\texttt{HBUSREQ}_i$ and $\texttt{HGRANT}_j$ signals are common to other bus interfaces such as Wishbone (grant and cycle signals), Avalon (waitrequest and read/write signals), VCI (valid and acknowledge signals) or CoreConnect (command and response send and accept signals). For other types of interconnects, such as AMBA AXI, similar signals are available that allow to monitor the interconnect usage.

Overall, our approach to account CCS does not change the AMBA interface or protocol. Instead we simply snoop the AMBA AHB arbiter signals and with this information we measure the time that a given task waits for the others when it tries to get access to the bus and the time that spends using it, so that all types of accesses are captured [23]. The hardware cost in terms of storage is 16 counters $\times 32$ bits each. That is 64 bytes.

F. Memory controller

The memory controller comprises a FIFO queue, with one entry per core, the memory bus, and a command translator that translates AMBA requests into DRAM commands. When a request from a core c_i arrives at the FIFO queue, if the queue is empty, it is put at the top of the queue and accesses the memory immediately. Otherwise, when other requests are

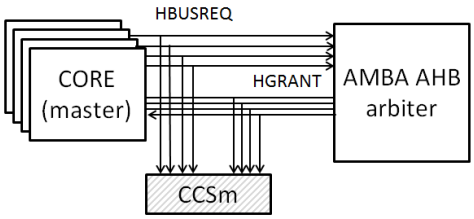

Fig. 5: AMBA Signals and CCSm.

in the queue, it has to wait for them to finish since the memory bus and memory controller only accept one request at a time. The former corresponds to the time the request takes to be processed once it is granted access and it cannot be further delayed by any preceding request. The latter is the time the request waits to get access to the memory controller.

At any given cycle, the core using the memory could be determined if information is provided about the id of the core that is at the top of the FIFO queue, and hence whose request is being processed. The id of the requests in a FIFO queue entry – other than that at the top position – are those cores suffering contention. Hence, the knowledge of the core id for each request in the FIFO queue is needed. In the GR740, the id of the core generating a request is kept in the AMBA AHB as master id. However, once the request accesses the L2 this information is no longer kept.

Intuitively this would require keeping the core id on every L2 cache line, which would incur a significant increase in the L2 cache size. However, in reality the core id is kept in the Miss Status Holding Register (MSHR) of the L2 cache. The process goes as follows. On the event of an access to the L2, the L2 cache determines whether that access is a hit or a miss. In case of a miss, the request is stored in the MSHR with the core id as part of the miss request, to be able to respond to the appropriate master afterwards.

Our proposal propagates the core id from the MSHR to the memory controller FIFO queue. When a request is sent from the L2 to the memory controller it is tagged with the core id. Both the MSHR and the FIFO queue are relatively small with sizes up to 8/16 entries in general. In our case both have 4 entries. Hence our proposal incurs an increase in area of $4 \times \log_2(Nc = 4) = 8$ bits (1 byte).

In terms of logic, each position in the FIFO queue sends a signal to the CCSm with the core id of the request in that position, if any. The core i at the top of the FIFO queue, $\texttt{FIFO}(0) = i$, is the one accessing the memory and the rest of the cores in the FIFO queue, $j \in \texttt{FIFO} | j \neq i$, are those interfered by i. The CCSm considers working cycles in memory for core i, those cycles when the core is at the FIFO's top entry. If there is any other request from another core j in the FIFO queue, the CCSm accounts contention cycles in the memory for them caused by core i.

The CCSm has $N_c \times N_c = 16$ Memory Contention Counters ($mcc_{i,j}$) for the memory controller with an associated size of $16 \times 32/8 = 64$ bytes. Counter $mcc_{i,j}$ stores the number of cycles that $(\texttt{FIFO}(0) = i)$ & $(j \in \texttt{FIFO})$ holds. Counters where $i \neq j$ hold the contention cycles that core j suffers from core i. Counters with $i = j$ store the working cycles that core i uses to process its requests.

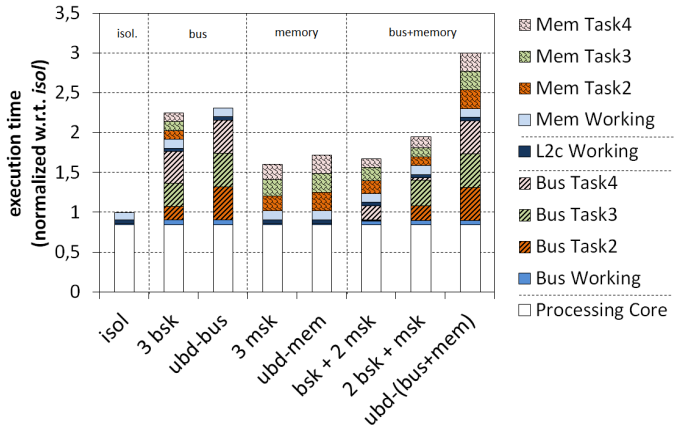

Fig. 6: EEMBC with different contention workloads.

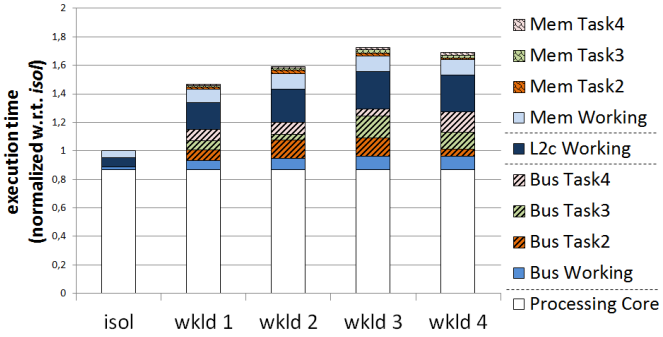

Fig. 7: *cacheb* EEMBC in different workloads.

V. Evaluation

We carried out the evaluation of CCS under two different setups, both focusing on the GR740 [11]. First, we model the GR740 on a simulation tool in which we implemented the PMCs presented in Section IV. In our second evaluation environment we focus on an FPGA model of the GR740 in which we implemented the PMCs for the bus, the CCSm and their signals.

Benchmarks. As reference applications we use the EEMBC Autobench suite [33], which behave as some real-world critical applications. In particular we use: *a2time*, *aifftr*, *aifirf*, *aiifft*, *basefp*, *cacheb*, *canrdr*, *idctrn*, *iirflt*, *matrix*, *pntrch*, *puwmod*, *rspeed*, *tblook* and *ttsprk*. We also developed a set of synthetic kernels that inject constant high pressure either on the shared bus or on the shared memory. The Bus-Stressing Kernel, or *bsk*, comprises memory read requests that always miss the L1 and hit the L2, thus maximizing the traffic on the bus. This is done by having 5 memory accesses that access the same set of the L1 cache, thus exceeding its 4 ways. These accesses hit on the L2 cache by targeting different sets on the L2 cache. The Memory-Stressing Kernel, or *msk* comprises memory read requests that always miss on the L1, but in this case they also miss on the L2, following the same procedure of targeting the same set, done for the L1. When our task under analysis runs against three of these kernels, it finds very high contention on the bus or the memory respectively.

For simplicity, in our experiments we run four-task workloads in which the task in core 1 (also referred to as task 1) is the task under analysis (TUA) for which the CCS is derived. The tasks on the other cores are considered contending tasks.

A. Simulator evaluation

Setup. We model the GR740 [11] running at 200MHz using a modified version of the SoCLib [38]. Each core's private instruction (IL1) and data (DL1) caches are 16KB, 4-way with 32-byte lines. The shared second level (L2) 256KB cache is split among cores, each receiving one way of the L2, so that inter-task contention only happens on the bus and the memory controller. With DRAMsim2 [40] we model a 2-GB one-rank DDR2-667 [27] system. We validated the simulator at cycle-level against the NGMP implementation in the N2X [12] evaluation board, as part of an internal study carried out in the European Space Agency. A low-overhead kernel allowed cycle-level validation. The deviation in terms of accuracy is less than 3% on average for EEMBC benchmarks and 1% for the NIR HAWAII benchmark [24].

We implemented the CCSm and its signals, as presented in Section IV, including the PMCs required to directly measure the working and contention cycles on the bus and the memory controller. With these modifications we build the CCS directly from the measures obtained from the PMCs in one execution.

Timing Validation. In our first experiment, we use the CCS to cross validate two different methods deriving bounds to the WCD tasks suffer accessing GR740 shared resources. For that purpose we run each EEMBC benchmark under different 4-task workloads and collect the ACD obtained with CCS and compare it with the expected WCD.

WCD bounds with measurements. In this case we run each EEMBC against several copies of the *bsk* or *msk*, which are expected to generate high (potentially the highest) contention in the bus and the memory to the TUA.

Theoretical WCD bounds. In this case, WCD bounds are predicted by multiplying the number of accesses the benchmarks does to the bus (and the memory) by the maximum delay each request can suffer – called Upper-Bound Delay or *ubd* [31] – in the access to the bus (and the memory). In the case of the bus, assuming that all accesses target the L2 cache, *ubd-bus* is 27 cycles that corresponds to the latency of three contending cores sending requests that have the highest latency (9 cycles) in our setup. In the memory case, *ubd-mem* is 69 cycles that corresponds with three complete row accesses of the contending cores, i.e. $3 \cdot t_{RC}$, with $t_{RC} = 23$.

Figure 6 shows the CCS obtained when running each EEMBC benchmark in isolation (*isol*); when using the measurement-based WCD model that runs each EEMBC against either three *bsk* or *msk*; and when using the theoretical WCD model for which the CCS for *ubd-bus* and *ubd-mem* are constructed replacing the interference on *bsk* and *msk* with the theoretical *ubd* for each resource respectively or for both in the *ubd-bus+mem* case. For simplicity, Figure 6 shows results averaged across all the EEMBC Autobench.

We observe that the measurement-based WCD model, even if *bsk* and *msk* put a high load on the resource, never exceeds the bounds provided by the theoretical *ubd*-based model. In particular for every shared resource and contending task the theoretical bounds are higher than the measured ones.

This experiment shows that the approach based on running each TUA against 3 copies of *msk* and *bsk* does not capture that theoretical bound. For the case of *bsk* we observe that these three benchmarks generate some activity in the memory (i.e. `Mem Task2`, `Mem Task 3` and `Mem Task 4` are non zero). This can be the reason why these benchmarks do not generate maximum contention delay in the bus. Said that, it is the case that this models approximates quite well the maximum

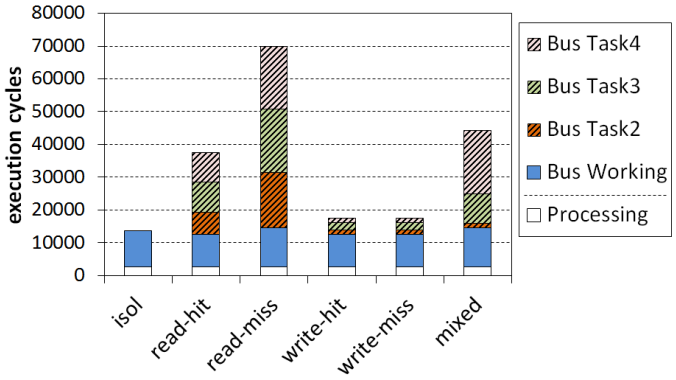

Fig. 8: *r-bsk* under analysis against different bus contenders.

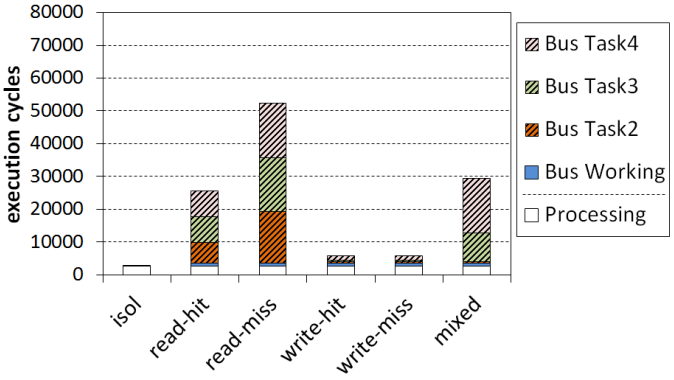

Fig. 9: *w-bsk* under analysis against different bus contenders.

theoretical value using *ubd* in both, bus and memory, but not in a combined manner [34] as shown in the rightmost bar (*ubd-bus+mem*) of Figure 6.

The result of this experiment also shows that in the experiments done nothing suggests that theoretical WCD bounds are violated for any of the contender tasks in any resource. This increases confidence on the validity of those bounds.

Scheduling Optimizations. In our second experiment we show how CCS could be used to improve other system metrics. For this experiment we choose *cacheb* EEMBC benchmark as TUA. Figure 7 shows the CCS obtained for the TUA under different workloads, composed of randomly picked EEMBC benchmarks. First, we see that most of the interference is suffered on the bus, because the L2 is filtering the accesses to the memory. We also see that Task2 in workload 2, i.e., (wkld:2,task:2)=(2,2), has high impact of TUA's execution time due to the interference on the bus. On the other hand, (2,3), (3,4) and (4,2) have a low impact on TUA's execution time. This information can influence the scheduling decisions, for instance, allocating the tasks with the highest interference, such as (2,2), into the same core as the TUA so that they do not interfere each other.

B. FPGA Evaluation

Setup. We implemented the CCS for the AMBA AHB bus on a real GR740 FPGA prototype using a Xilinx ML510 evaluation board. In particular we implemented the bus CCS-aware PMCs that measure the working and contention cycles on the processor AHB bus. We used the commercially available Cobham Gaisler GRMON2 [10] debug monitor software to directly extract the CCS from the statistics unit (L4STAT) of the GR740, without affecting execution. The CCS is directly constructed from the readings obtained in one execution of the task, i.e, no further post processing is required.

The real cost of the modifications is low since we reuse the available counters and infrastructure of the GR740 L4STAT unit. Our PMCs just require the wiring of the AMBA signals, which corresponds to 8 1-bit signals, that is 4 HBUSREQ and 4 HGRANT signals. The cost of the wiring depends on the target technology, synthesis tool and design size.

As TUA, we use bare-metal resource stressing kernels that put high load on the bus using either *read* or *write* bus requests, called *r-bsk* or *w-bsk* respectively.

Evaluation. To evaluate the design we use different *bsk* contenders with different type of requests, *read* or *write* that either *hit* (*read-hit*, *write-hit*) or *miss* (*read-miss*, *write-miss*) on the L2. Each type of request causes a different bus contention due to the different behavior of requests and L2 latencies.

Figure 8, shows the CCS when taking as TUA *r-bsk* against different workloads consisting of different types of *bsk*, shown on the x-axis. We observe that the worst effect is caused by the *read-miss* workload, followed by the *read-hit*. This happens because read requests that miss on the L2 hold the bus while accessing memory. On the other hand, read requests that hit on the L2 do not access memory, thus, requiring less time on the bus. We also see that *write-hit* and *write-miss* workloads have the same effect on the bus, which is smaller than both *read* workloads. This happens because write requests only require an acknowledge and are immediately responded, even if they miss on the L2.

The rightmost bar in Figure 8 shows a mixed scenario, in which Task2 uses *write misses*, Task3 *read hits* and Task4 *read misses*. The obtained CCS effectively demonstrates the capabilities to identify the contender with the highest interference in a workload in which contenders have different resource usage profiles, Task4 in this case.

Figure 9 shows the same contention scenarios using *w-bsk* as TUA. As shown, the bus working cycles reduce in comparison with Figure 8, because *write* bus requests from *w-bsk* take much shorter time in the bus than *read* requests due to the longer L2 cache read hit latency. We observe the same contender behavior with similar amount of contention, even though *w-bsk* requires less time on the bus than *r-bsk*. This happens because contention mostly depends on the amount of interfered requests and not on their duration, since once a request accesses the bus, it cannot be preempted.

Overall, CCS provides valuable and accurate information to build and validate timing models and WCD bounds.

VI. PMC Survey

In this section we provide further details on the PMC support of different architectures that can be useful to measure contention in shared resources to some extent.

IBM. The IBM POWER7 [25] is an 8-core multicore in which each core is 4-thread Simultaneous Multithreaded. Each core is divided into two clustered execution pipelines, with each one supporting two threads. Program performance analysis in such an aggressive core architecture with resources shared among different threads is complex. The POWER7 processor comprises a Performance Monitoring Unit (PMU) with six thread-level PMCs. Four of these are programmable from software to monitor the desired (four) events at the same time. There are more than 500 possible performance events

that can be read.

Based on our analysis of the POWER7 we identified counters for cycle utilization, busy and idle cycles, occurrences of a number of events, instructions of each type executed, and the amount of data transferred. PMCs in the *threshold* and *current events* were not found.

Intel. Usually Intel processors feature superscalar execution, complex branch predictors, out of order execution, and several levels of cache memories. Provided PMCs focus on providing performance metrics for a single process with counters for analyzing branch predictor effectiveness, cache misses due to speculative execution, coherence protocol metrics, etc. Most counters can be configured to measure events for either from a core or all the cores, an agent or all agents, and other kinds of specific qualifications (such as detection of all events/exclude prefetching events, or counts for different states for the coherence protocol used). In this work we have analyzed the PMCs available in the following Intel architectures: Haswell (Xeon E3-1200 v3), Ivy Bridge (Xeon E3-1200 v2), Sandy Bridge (Core i7-2xxx, Core i5-2xxx, Core i3-2xxx, Xeon E3-1200), Nehalem (Core i7, Xeon 5500 Series), and Westmere (Xeon E7-xxxx).

Intel's PMCs are to some extent similar to those of the IBM POWER7 since both processors are general-purpose high-performance ones. Still, some relevant differences exist. For instance, Intel processors don't have explicit support to measure data transferred. However, Intel's PMCs include the following counters:

– *Threshold exceeding count*. These counters measure the number of times a threshold specified in number of cycles has been exceeded for a given event. The threshold value is configured by the user.

– *Outstanding Requests*. This counter measures events such as cache requests, all offcore requests, etc. in the moment of reading the counter.

ARM v7. The ARMv7-A architecture [20] provides 6 different 32-bit counters, which can count any event available. This architecture is used, among other by the Cortex-A7, Cortex-A9, Cortex-A15 [7] and the big.LITTLE [19] system. ARMv7-A provides a Performance Monitoring Unit (PMU) with 6 performance counters. The events counted by ARM architectures are a subset of those available in high-performance processors such as Intel and IBM ones described before as shown in Table I. Still this architecture is rich in counters for a number of events such as miss-predicted branches, number of exceptions, L1 write backs, number of L1/L2 refills, L1 accesses, bus accesses and data memory accesses among others.

Freescale. Freescale P4080 processor hosts eight e500mc [18] cores, which are superscalar processors that can issue and complete two instructions per clock cycle. Each core has a private L1 instruction and data cache. It also has a private L2 unified cache. The eight cores are connected through a proprietary CoreNet Fabric coherent interconnect with two shared 1MB L3 off-chip caches. Each L3 off-chip cache is connected to a separate DDR memory controller.

The PMC support in the P4080 offers dedicated core and SoC platform counters [15][16]. At the core level, the e500mc core allows monitoring 256 different hardware events, each core being able to monitor 4 different events at a given time in 4 dedicated 32-bit registers. At the SoC level the P4080 Event Processing Unit (EPU) allows counting SoC platform events of interest. Although some events across Intel and Freescale processors differ due to their different designs, in essence, the set of PMCs in both platforms is quite similar as shown in Table I.

GR740. The GR740 is a 4 core LEON4 processor developed at Cobham Gaisler under contract with the European Space Agency (ESA). It contains one or more LEON4 Statistical Units (L4STAT). The debug driver for L4STAT provides an interface for reading and configuring the 32-bit performance counters available in a L4STAT core. Each L4STAT allows configuring any available sixteen events we want to monitor. Each counter has an associated control register. Both the counters and the control registers are mapped to the peripheral address space.

The available events can be divided in three different categories, depending on the component counting the events. Processor events: events generated by the processor, e.g., pipeline or the L1 cache; bus events: events generated by the bus, e.g., busy cycles or number of read accesses; and Device specific events: events generated by other devices such as the L2 or the IOMMU.

Again, although the PMCs across chip vendors do not match, the type of counters one can find in the GR740 is quite similar to those in the Freescale and Intel architectures, with the exception of the idle cycle counters that the GR740 does include for the bus.

VII. Related Work

Performance Monitoring Counters (PMCs) have been traditionally used to measure average performance and power consumption [28]. One of the few works that addresses contention monitoring between tasks is [43], which uses cache scouts to monitor contention on shared caches. However, with few exceptions [37][8], cache partitioning is the common solution in the context of CRTES due to the complexity of estimating the WCET accurately on top of shared caches. Although our work could be ported to non-partitioned caches building on [43], we leave this analysis for future work.

The IBM POWER family, starting with the POWER5, have developed a Cycles-Per-Instruction (CPI) stack that covers the resources on which each task spends its cycles. The CPI stack reports the cycles spent in each core resource. For instance in the load/store Unit (LSU) [25]. This happens when a load/store operation is stalled. The CCS instead does not focus on the local resources getting clogged by contention (e.g. the LSU) but the off-core resources identifying where (and how much) contention occurs and the contending core producing it.

In [36] authors use custom PMC to derive WCD and WCET estimates with measurement-based timing analysis on a bus-based system. Authors assume that the WCD for the bus is known, which is not always the case in real implementations, as shown for the AMBA AHB bus [23]. Several works derive bounds, during the analysis phase of the system, to the WCD that a task may suffer in different processor resources assuming static or measurement-based timing-analysis [31][35][22][36][14][32][4][26]. We do not detail these works since we focus on measuring the ACD tasks suffer rather than on making a-priori predictions on WCD.

The ACD in multicore processors has been characterized mostly using resource-stressing kernels (rsk) [34][2], for instance in a previous implementation of the NGMP [14] or in the Freescale P4080 [29]. However, these approaches provide

neither a breakdown of the ACD nor means to measure it accurately online, while CCS provides both.

Authors in [30] propose a *runtime monitoring* to control the resource usage of tasks running on a multicore, preventing tasks from having resource usage limit violations. Authors make use of access count PMCs, such as bus access counts. However, it has been shown that bus latencies may differ accross different types of accesses and even for the same type [23]. As a result, access counts does not provide the actual impact of contention time which has to be estimated. The ACD instead provides in an exact manner contention delay for each task.

VIII. Conclusions

Obtaining accurate ACD breakdowns in MPSoCs provides evidence about the trustworthiness of contention bounds and increases confidence on derive execution time bounds; it is also of prominent importance to detect the reasons for overruns in a critical system once it has been deployed. It also helps optimizing system's performance and improving scheduling decisions based on the knowledge of the real impact of contenders in terms of execution time. Unfortunately, to the best of our knowledge no solution exists to measure and classify the ACD in CRTES.

In this paper we propose the Contention Cycle Stack (CCS), which provides an effective representation of the ACD that classifies contention per resource and contending task by means of measurements. The CCS relies on some existing Performance Monitoring Counters (PMCs) and introduces its own-set of low-overhead PMCs to track the events that allow building ACD per task and shared resource. Our evaluations for the Cobham Gaisler GR740 (NGMP) show the benefits of the CCS. In particular CCS enables understanding the source and magnitude of the actual contention delay (ACD) caused by contending tasks in a MPSoC, which is crucial to adopt MPSoC in CRTES.

IX. Acknowledgements

The research leading to these results has received funding from the European Space Agency under contracts 4000109680, 4000110157 and NPI 4000102880, and the Ministry of Science and Technology of Spain under contract TIN-2015-65316-P. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

[1] Rapitime. https://www.rapitasystems.com/products/rvs.

[2] A. Abel et al. Impact of resource sharing on performance and performance prediction: A survey. In *CONCUR*, 2013.

[3] J. Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 2015.

[4] B. Akesson et al. Predator: a predictable SDRAM memory controller. In *CODES+ISSS*, 2007.

[5] ARM. *Cortex-R5. Technical Reference Manual. Revision: r1p2*, 2011.

[6] ARM Ltd. AMBA specification (rev. 2), 1999.

[7] ARM Ltd. The ARM Cortex-A9 processors. ARM white paper, 2009.

[8] S. Chattopadhyay et al. A unified WCET analysis framework for multicore platforms. *ACM Trans. Embed. Comput. Syst.*, 13(4), Apr. 2014.

[9] Cobham Gaisler. *Dual-Core LEON3-FT SPARC V8 Processor - GR712RC-UM - Data Sheet and Users Manual Draft, 2014*.

[10] Cobham Gaisler. *GRMON2-User Manual Version 2.0.62, March 2015*.

[11] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - GR740-UM-DS-D1 - Data Sheet and Users Manual Draft, 2015*.

[12] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-N2X Data Sheet and Users Manual Draft*, 2013.

[13] G. Fernandez, J. Jalle, J. Abella, E. Quinones, T. Vardanega, and F. Cazorla. Increasing confidence on measurement-based contention bounds for real-time round-robin buses. In *The 52st Annual Design Automation Conference 2015, DAC '15, San Francisco, CA, USA, June 7-11, 2015*, 2015.

[14] M. Fernandez et al. Assessing the suitability of the NGMP multi-core processor in the space domain. In *EMSOFT*, 2012.

[15] Freescale. *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture Processors*.

[16] Freescale. *On-Chip Debugging of Multicore Systems*.

[17] FreeScale. *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual. Rev 1*, 2012.

[18] FreeScale. *e500mc Core Reference Manual. Rev 3*, 2013.

[19] P. Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. ARM white paper, 2011.

[20] http://www.arm.com. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*.

[21] Infineon. *XMC4500 Microcontroller Series for Industrial Applications Reference Manual. Rev 1*, 2012.

[22] J. Jalle et al. Deconstructing bus access control policies for real-time multicores. In *SIES*, 2013.

[23] J. Jalle et al. AHRB: A high-performance time-composable amba ahb bus. In *RTAS*, 2014.

[24] A. Jung et al. The H2RG infrared detector: introduction and results of data processing on different platforms. Technical report, European Space Agency, 2012.

[25] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. POWER7: IBM's next-generation server processor. In *IEEE Micro*, 2010.

[26] 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 *RTAS*, 2014.

[27] Kingston. *KVR667D2S5/2G Datasheet*, 2011.

[28] Q. Liu et al. Hardware support for accurate per-task energy metering in multicore systems. *ACM Trans. Archit. Code Optim.*, 2013.

[29] J. Nowotsch et al. Leveraging multi-core computing architectures in avionics. In *EDCC*, 2012.

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

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

[32] M. Paolieri et al. Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1s), 2013.

[33] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.

[34] P. Radojković et al. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 2012.

[35] J. Rosen et al. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, 2007.

[36] H. Shah et al. Measurement based WCET analysis for multi-core architectures. In *RTNS*, 2014.

[37] M. Slijepcevic et al. Time-analysable non-partitioned shared caches for real-time multicore systems. In *DAC*, 2014.

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

[39] Special Committee of RTCA. DO-178C, Software Considerations in Airborne Systems and Equipment Certification, 2011.

[40] D. Wang et al. DRAMsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 2005.

[41] A. West. NASA Study on Flight Software Complexity. Final Report. Technical report, NASA, 2009.

[42] Wilhelm R. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.

[43] L. Zhao et al. Cachescouts: Fine-grain monitoring of shared caches in CMP platforms. In *PACT*, 2007.