

Improving Timing-Related Guarantees for Main Memory in Multicore Critical Embedded Systems

Asier Fernández De Lecea
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
asier.fernandezdelecea@bsc.es

Mohamed Hassan
Faculty of Engineering,
McMaster University
Hamilton, Canada
mohamed.hassan@mcmaster.ca

Enrico Mezzetti
Barcelona Supercomputing Center
Barcelona, Spain
enrico.mezzetti@bsc.es

Jaume Abella
Barcelona Supercomputing Center
Barcelona, Spain
jaume.abella@bsc.es

Francisco J. Cazorla
Barcelona Supercomputing Center
Barcelona, Spain
francisco.cazorla@bsc.es

Abstract—Main memory is one of the most complex resources to analyze in multicore-based embedded real-time systems, with contention in the memory controller and the timing constraints of the main memory device as the main contributors to that complexity. One of the main challenges in multicore real-time systems is producing the required evidence on the management of contention delay for the certification. This stems from the fact that current MPSoCs barely provide any event monitors on how tasks interact and delay each other in memory. Besides, even if hardware and software mechanisms are in place to mitigate contention in the memory system, it is hard – if at all possible – to provide evidence about their correctness. In this work, we cover this gap by proposing a lightweight hardware mechanism that tightly tracks inter-core contention in memory. The proposed hardware mechanism, which we evaluate in detail, improves the quality of timing-related evidence that must be provided on how contention in main memory of multicore real-time systems is handled in adherence to applicable safety standards.

Index Terms—Multicore, main memory, performance monitoring counters, event monitors, contention monitoring

I. INTRODUCTION

The use of multiprocessor systems-on-chip (MPSoCs) has widely spread in domains like avionics [1] and automotive [2], [3]. This responds to the need of providing the unprecedented levels of performance to support performance-hungry software control applications [4]–[7]. Contention in shared resources is one of the main open challenges brought by MPSoCs, with a plethora of software techniques for commercial-off-the-shelf (COTS) MPSoCs [8]–[10] and hardware designs proposed to address it [11]. The quantification of multicore contention entails defining the analysis and testing approach to produce the required evidence to support verification objectives in

the applicable safety standard (e.g. A(M)C 20-193 [12] for avionics) for the timely execution of multicore software.

Hardware solutions aim to achieve predictability by design simplifying the overall safety argumentation about software timing as they heavily limit or even remove contention [13]–[17]. Naturally, chip vendors have some resistance to adopt these designs since isolation and segregation cause a reduction of applications’ average performance, and the necessary hardware support is not demanded by mainstream domains in which a high volume of chips is sold.

Software solutions build on event monitors (EMs) that are read via the performance monitoring counters (PMCs). EMs provide insightful information on how corunning tasks affect each other. In fact, EMs have been shown to be a key stepping stone to produce the required safety information for certification [18]–[21]. Some shared resources like the last level cache come equipped with dozens of counters that can be useful to that end. For instance, in the NXP T2080 [22], the L2 cache, which is shared among corunning cores, provides global counters (e.g. accesses, hits, misses, data accesses, instruction accesses, ...) as well as per-thread (core) EMs. EMs allow detecting, for example, the increase in the number of misses tasks suffer due to contention, which helps understanding how much a task is affected in the L2 by tasks in other cores.

For the memory subsystem, i.e. the memory controller (MC) and memory device, trackable events are unsatisfactory with only few MPSoCs supporting a restricted number of events. For instance, in the NXP T2080, the event processing unit collects information on page hits, page misses, page reads, and page writes [22], while the NXP iMX6 provides a MC profiling module that collects global statistics on memory requests [23]. These EMs are still insufficient to understand how tasks affect each other in memory, which hinders the overall safety argumentation, even when software techniques are deployed to contain contention (e.g. splitting main memory banks per core). Evidence can build on a costly combination of memory simulators, which need to undergo qualification [24],

This work has been supported by Grant PID2019-107255GB-C21 funded by MCIN/AEI/ 10.13039/501100011033, the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No.772773), and the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors also thank the support of the Department of Research and Universities of the Government of Catalonia to the Smart and Safe Autonomous Systems (SSAS) Research Group, (Code: 2021 SGR 00637).

and code inspection to show, for instance, that a stressing application accesses all main memory banks.

In this work, we address the management of Double Data Rate (DDR) synchronous dynamic random access memory (SDRAM) contention in the overall safety certification via a low-overhead hardware proposal (MCTrack) aimed to increase observability of contention. In particular, we propose a new set of *contention monitors* for the memory system that track the contention among tasks rather than the events that are normally correlated to contention (e.g. read/writes to memory generated by each task). Recent success stories show how some chip providers are open to implement new monitoring support as it does not affect cycle time and processor performance [25]–[27]. More in detail, we first analyze the main sources of contention in memory that we split into the front-end (request queues) and the back-end (command queues). We develop on how MCTrack ascribes contention to each core depending on how their requests affect each other, and propose a possible low-complexity implementation. Our results show that MCTrack tightly tracks contention among cores in the memory system being of great help for timing validation in multicores.

The rest of this work is organized as follows. Section II covers the benefits of event and contention tracking. Section III introduces our reference MC. Section IV and Section V, introduce and present a possible implementation of MCTrack, respectively. Section VI evaluates the benefits of contention tracking and the hardware complexity of MCTrack. Section VII presents the most related works. Section VIII draws with the main conclusions and take away messages.

II. EM BENEFITS IN MULTICORE CRITICAL SYSTEMS

A. Exploiting EMs in Timing V&V

Timing characterization is hampered by *multicore timing (contention) interference* arising from hardware resource sharing in MPSoCs [28]–[30], whose impact on software timing has proven to be considerable [31], [32]. EMs have been increasingly considered as the cornerstone of diverse approaches from timing verification to hardware characterization.

1) *Timing Verification*: Interference diminishes the predictive value of end-to-end timing characterization as the observed timing is the cumulative result of several factors, including how applications contend for the shared resources [33], [34]. In this scenario, EMs have been increasingly considered as critical source of information, both to complement end-to-end timing and to enable alternative methods for timing characterization [19]–[21].

2) *Resource Sharing Control*: Despite the increasing hardware [25], [26] and software support [18], [35], [36] to control the way hardware resources are shared, no silver bullet solution exists to completely remove all sources of timing interference in complex platforms [32], [37], [38]. Those solutions build on the capability of the underlying hardware to provide EMs that are related to core activity and usage of shared resources, and promptly react to usage quota overruns in a way that is transparent to the application itself.

3) *Hardware and Timing Characterization*: EMs are also critical in providing empirical evidence to corroborate verification arguments in a consistent manner with domain-specific regulatory and support documents [12], [39], [40]. Certification authorities make software certification contingent upon the provision of *extensive evidence* on the process and outcomes of the characterization of software timing behavior [41], [42]. It is therefore necessary to support the conclusions of the software timing characterization effort with a well-structured strategy to cope with the sources of timing interference (*interference channels* in the avionics terminology) and the collection of the necessary certification evidence. EMs are instrumental for attaching supportive evidence on the effectiveness of static resource partitioning approaches. We illustrate this by considering how A(M)C 20-193 deals with shared L2 cache partitioning (either hardware [43] or software [44]): even if timing interference arising from inter-core cache evictions in the shared L2 is ruled out because of a specific partitioning scheme, it is still necessary to provide evidence that the partitioning mechanism works as expected. In this case, EMs can be exploited to provide evidence on the lack of interference.

B. Contention Tracking in the MC

Monitoring units in modern MPSoCs provide extensive coverage for core-level events and activity, but lack comparable coverage for memory-related events. Support for EMs tracking activity in the MC is only available in a limited set of platforms [23], [43] and for limited events: it typically allows to track the global number of reads, writes, page hits and page misses [43] or overall MC bandwidth [36]. Also, it is not possible to map events (and hence contention) to a specific core. On the other hand, accountability is a relevant property when analyzing the root cause of contention delay: this is not only critical for design or performance optimization objectives, but also, from the perspective of this work, for better understanding multicore contention. The main limitation, however, is inherent to the type of events being tracked by EMs: access counts are just a proxy metric to measure the contention a task (or a group thereof) can generate on others. They are not necessarily a fair metric as they fail to consider the actual impact that each access of the contending task generates on the task under analysis [45]. Such impact depends on several factors like the type of request (which can be tracked to some extent), the status of the target shared resource, and how requests from different cores interleave.

In considerations of EM limitations, in this work we do not seek for extending EM support to memory controller events. Instead, we propose an efficient approach to monitor the actual contention by using a set of direct metrics. Direct contention tracking, as opposed to event tracking, improves characterization and monitoring of contention impact for certification evidence [19]–[21] and more accurate contention monitoring, which in turn enables the computation of less conservative bounds on bandwidth allocation [18], [35], [36].

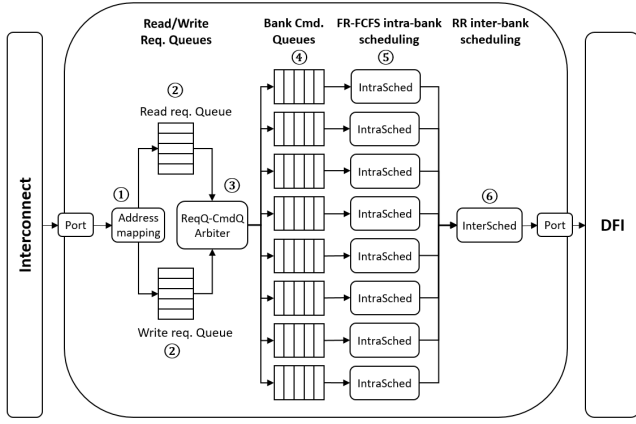


Fig. 1. Block diagram of the reference DDR memory controller

III. MC STRUCTURE AND ARBITRATION MECHANISMS

DRAM consists of parallel entities called banks, a matrix of memory cells composed of rows and columns. To increase parallelism, DRAM modules can combine several banks into bankgroups, bankgroups into ranks, and ranks into channels. Each DRAM bank provides a faster access to data in the currently-activated row, which is usually referred to as row buffer hit and requires only a CAS RD/WR command to read/write the target columns in that row. On the other hand, accesses targeting a row different to the one in the row buffer (known as row conflicts) suffer a slower access since they need to precharge the row first (through a PRE command), activate the new row (through an ACT command), and then perform the read/write (CAS RD/WR) access.

These commands must adhere to timing constraints according to the DRAM JEDEC standard [46]. For instance, in the example in Figure 6, once an ACT command ($A1$) is issued to a bank, the following RD command ($R1$) has to wait t_{RCD} cycles before it can be issued to the same bank. An ACT to another bank in same bankgroup ($A3$) has to wait for t_{RRD_L} cycles due to the previous ACT command as well. A second RD command to the same bank ($R2$) can only be issued once t_{CCD_L} cycles have elapsed since the previous RD command. The same happens to $R3$, which is also constrained by t_{CCD_L} cycles since the previous RD to a bank in the same group. t_{CAS} cycles are required between the issuing of a RD command and the requested data being transmitted through the bus, and t_{BURST} are the number of cycles the data bus is occupied when a burst of data occurs.

A. Request and Command Queues

Figure 1 shows a diagram of the reference DRAM MC used in this paper. On arrival to the MC, the target address of a request is split into bits for channel, rank, bankgroup (for DDR4), bank, row and column by the address mapping logic ①. This information is stored in the read/write request queues along with the rest of the information of a request.

1) *RReqQ and WReqQ*: MCs usually implement separate read (RReqQ) and write (WReqQ) request queues for read and write operations ②. Having split queues allows implementing

write batching, which allows prioritizing read operations while write operations are issued in batches. Issuing write operations in batches reduces the number (and timing penalty) of read-to-write and write-to-read transitions between DRAM commands. Prioritizing reads over writes is common in commodity platforms since writes to DRAM are usually due to last-level cache evictions and do not stall the processor pipelines. For that reason, they do not contribute to the task's execution time critical path. Therefore, and similarly to existing works [47]–[49], we only consider the delays suffered by read requests. Write-batching is triggered when the WReqQ reaches a maximum capacity threshold or when there are no pending reads in the memory controller (specifically, in the BCmdQs) and the number of write requests in the WReqQ is above a minimum capacity threshold. In order to ensure memory consistency, the memory controller respects the order among reads and writes to the same DRAM location. To that end, read requests are allowed to hit in the WReqQ if there are pending writes to the same location, and writes arriving after a pending read to the same location in the RReqQ cannot be serviced until such read is serviced.

2) *BCmdQs*: Requests are issued from the RReqQ or WReqQ to per-bank command queues or BCmdQs ④. Read and write requests are then translated into different DRAM commands. Usually, MCs implement as many command queues as banks in the DRAM device (8 or 16, for instance).

B. Arbitration

1) *XReqQ to BCmdQ arbitration* ③: Every cycle the arbiter picks at most one request from the RReqQ or WReqQ. Arbitration takes into account the occupancy of each BCmdQ so that only requests targeting banks whose BCmdQ is not full are *issuable*. Request arbitration actually entails two levels of arbitration. First, *interXReqQ* decides whether reads or writes should be scheduled based on the aforementioned write-batching approach. Second, *intraRReqQ* (*intraWReqQ*) selects the oldest issuable request in the RReqQ (WReqQ).

2) *BCmdQs to memory arbitration*: Memory requests require different DRAM commands to be executed depending on the *state* of the bank and the type of the request.

IntraSched. Each intra-bank scheduler (IntraSched) ⑤ determines the DRAM command that can be issued from a BCmdQ at any cycle. A command is *issuable* (*ready*) when it fulfills its JEDEC timing constraints. To track latencies among commands, each IntraSched keeps a *timing table* that holds the time when each command type (PRE, RD, WR, ACT, etc.) can be issued, see Figure 2. A command is issuable when the cycle value present in the entry for its command type in the timing table is smaller or equal to the current cycle. When a command is issued, the timing table is updated to capture the new timing constraints on every command type due to the issued command. In particular, an entry in the timing table is updated when its previous cycle value, k , is smaller than the current cycle, n , plus the timing constraint imposed by the command being issued. For instance, if at cycle n an *ACT*

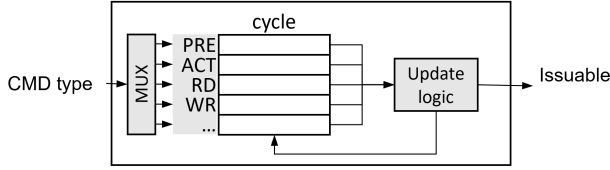


Fig. 2. Block diagram of the per-bank timing table. We assume 4 basic commands (ACT, RD, WR, PRE), and 2 refresh commands. We use no read/write-precharge or self-refresh commands.

command is issued to the bank, the RD entry in the timing table is updated if its value k is smaller than $n + tRCD$.

Note that, in a given cycle, several timing constraints can be simultaneously active. For instance, in Figure 6, after $R2$ is issued in cycle 14, both timing constraints $tCCD_L$ and $tRCD$ are active in cycle 15. This means that the $R3$ command has to wait because of both timing constraints in cycle 15, with $tCCD_L$ finishing at cycle 17. The timing table is updated in cycle 14 with the new timing constraint (18) when $R2$ is issued. This can complicate the tracking of contention when timing constraints are due to commands issued by different cores, as discussed in the next section.

Each IntraSched implements a first-ready first-come-first-served (FR-FCFS) policy. To that end, the IntraSched prioritizes ready younger row hits (RD/WR commands to the currently-open row) over ready older commands requiring a full row cycle. A *maximum row-hit threshold* limits the maximum number of younger requests that can bypass an older request, hence preventing starvation of requests that need to go through a full row cycle. Note that, if a requests requiring a PRE becomes *ready* but (1) there are pending row hits present in the BCcmdQ (can be ready or unready), and (2) the row hit limit has not been reached, the IntraSched halts the PRE command in favor of the pending row hits.

InterSched. InterSched arbitrates among all commands from different banks that are scheduled by the IntraScheds in round-robin fashion, sending one DRAM command at most to the DDR Physical Interface (DFI) per cycle ⑥. When a DRAM command is sent to memory, the timing tables must be updated to comply with JEDEC timing constraints. Also, the page state in the bank status information is updated to *OPEN*¹. Notably, it is possible to implement per-bank timing tables with intra-bank timing information along with a global timing table (in the InterSched) holding inter-bank timing constraints shared by all banks. However, for simplicity of the logic managing the timing table(s), commodity MCs use per-bank timing tables holding information on both intra-bank and inter-bank timing constraints, thus removing the global timing table [50], [51]. This work assumes the latter implementation.

IV. MEMORY CONTENTION TRACKING WITH MCTrack

We track memory contention for every running task, and classify each execution cycle of a task into one of the categories of the Execution Time Components (ETC). The ETC

¹The bank status also contains information like the cycle in which the bank needs to be refreshed and the id of the core that opened the current page.

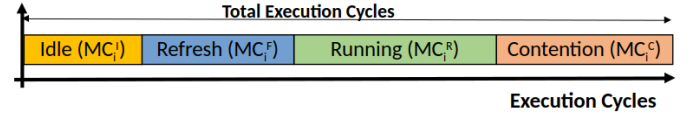


Fig. 3. MC ETC of a given task τ_i in the MC. Note that time is shown in the X axis, while in the following figures it is shown on the Y axis.

provides a clear understanding of how the task in each core is behaving in the MC: progressing in the MC (**Running**), with no requests in the MC (**Idle**), with requests blocked due to contention (**Contention**), or stalled due to reFreshes. These categories are complete, so that their addition matches the execution time of the task. That is, for a given task τ_i running in core c_i , its execution time t_i can be broken down as $t_i = MC_i^T = MC_i^I + MC_i^R + MC_i^C + MC_i^F$, where $MC_i^T = t_i$ is the total execution time of τ_i , see Figure 3.

In order to derive each of these components, we follow a bottom up approach in which we focus on the main contention points in the MC, which relate to the queues and arbiters introduced in Section III. In particular, we focus on the RReqQ (Section IV-A), the BCcmdQ (Section IV-B), and the arbiters around them. We derive the (local) ETC for each of them, and then we combine local ETCs to produce the ETC for the MC (Section IV-C).

A. RReqQ Execution Time Components

Figure 4 shows the proposed breakdown for the contention that requests suffer in the RReqQ. The vertical axis represents total cycles that are broken down into 3 main components.

Idle cycles. If in a given cycle (core) c_i has no requests in the RReqQ, in that cycle we set $RReqQ_i^I$ to one.

Running cycles. If in a given cycle c_i issues a read request to its target BCcmdQ, c_i 's RReqQ cycle is recorded as *running*, setting $RReqQ_i^R$ to one. Note that it is irrelevant whether c_i has other requests in the RReqQ, and whether those are issuable or not (e.g. they could have their target BCcmdQ full). Since c_i can issue a request, we consider it was able to progress in its execution in the RReqQ.

Contention cycles. The rest of the cycles in which c_i is not idle or running are recorded under the contention ETC ($RReqQ_i^C$). In particular, a read request can be contended-on due to (i) not being the first issuable request in the queue, (ii) its target BCcmdQ being full, or (iii) write batching being activated. If c_i 's requests suffer multiple contention types, only the contention type of the highest-priority request according to the ReqQ-BCcmdQ arbiter is considered.

Issue Contention (IC). c_i has a request ready to be issued, i.e. its target BCcmdQ is not full. However, it cannot be issued because the intraRReqQ arbiter selects another request from a different core c_j (i.e. it gives c_j 's request higher priority). That cycle is accounted as contention from c_j on c_i due to issue contention (IC), hence setting $RReqQ_i^{CIC_j}$ to one.

Full BCcmdQ (FU). c_i has a request in the RReqQ and it is not issuable, i.e. its target BCcmdQ is full. The BCcmdQ reports back to the RReqQ the core ID (c_j) of the highest-priority request preventing the BCcmdQ from draining. We

Contention ($RReqQ_i^C$)	Issue Contention ($RReqQ_i^{ICj}$)	Cycles in which c_i has issuable requests that are not issued because another older request from c_j is issued
	Full BCmdQ ($RReqQ_i^{CFUj}$)	Cycles in which c_i has reqs in the RReqQ but cannot be issued because the target BCmdQ being full due to core c_j
	Write Batching ($RReqQ_i^{CWBj}$)	Cycles in which c_i has reqs in the RReqQ but cannot be issued due to write batching
Running ($RReqQ_i^R$)		Cycles in which c_i issued a request to s BCmdQ from RReqQ
Idle ($RReqQ_i^I$)		Cycles in which c_i has no request in the RReqQ

Fig. 4. RReqQ contention breakdown for core c_i .

ascribe contention to c_j blocking the target BCmdQ of the read request of c_i and set $RReqQ_i^{CFUj}$ to 1.

Write Batching (WB). When in a given cycle core c_i has a read request in the RReqQ but write batching is activated, so read requests are not considered for issuing to the BCmdQs, in that cycle $RReqQ_i^{CWBj}$ is set to 1 (c_j is the core issuing the write request that cycle).

B. BCmdQ Execution Time Components

Typically, there are several instances of the BCmdQ in the MC (one per DRAM bank in our MC model). This affects BCmdQ contention tracking that is first performed at the individual BCmdQ instance level (Section IV-B.1) and then combining per-BCmdQ contention results into a single ETC (Section IV-B.2).

B.1. Per BCmdQ ETC

The ETC for a given $BCmdQ_k$, k is the BCmdQ ID and ranges from 1 to the number of DRAM banks (B), comprises three main top-level components, see Figure 5.

Idle cycles. Cycles in which c_i has no command in $BCmdQ_k$. $BCmdQ_{i,k}^I$ is set to one each of these cycles.

Running cycles. Cycles in which c_i issues a DRAM command to the InterSched and the InterSched selected it, effectively issuing the command to the DRAM device. $BCmdQ_{i,k}^R$ is set only in these cycles. Note that the case when c_i is able to issue a DRAM command to the IntraSched but InterSched selects another issuable command (from a different BCmdQ) is discussed as part of the contention cases below.

Contention cycles. A command in a given BCmdQ can suffer from contention due to one of the following reasons.

Intra-BCmdQ Issue Contention (NRA). c_i has a ready command that is not selected for issuing to the InterSched because another older command from core c_j in the same BCmdQ was instead selected by the IntraSched. $BCmdQ_{i,k}^{CNRAj}$ is set in these cycles.

Inter-BCmdQ Issue Contention (NER). c_i has a ready command that is selected by the IntraSched for issuing to the InterSched but another command from core c_j is selected by the InterSched. $BCmdQ_{i,k}^{CNERj}$ is set to 1 in these cycles.

Command Reordering (CR). A first situation in which we account CR delay is when request from c_i is a row conflict requiring a PRE command but (1) there is at least one pending younger row hit (i.e. RD/WR to the open row) from c_j present in the BCmdQ and (2) the row hit limit has not been reached.

Contention ($BCmdQ_{i,k}^C$)	Timing Constraints ($BCmdQ_{i,k}^{CTCj}$)	Cycles in which c_i has at least one cmd that is not ready due to compliance with JEDEC timing constraints caused by c_j
	Command Reordering ($BCmdQ_{i,k}^{CRj}$)	Req of c_i has ready PREcmd but c_j has pending rowhit to openrow (rowhitlimit not reached)
	Intra-CmdQ Issue Contention ($BCmdQ_{i,k}^{CNRAj}$)	Cycles c_i has ready cmds within a BCmdQ that are not issued due to a command from c_j was issued first to InterSched
	Inter-CmdQ Issue Contention ($BCmdQ_{i,k}^{CNERj}$)	Cycles in which c_i issued a DRAM cmd to InterSched but another cmd from c_j was picked instead by the InterSched
	Running ($BCmdQ_{i,k}^R$)	Cycles in which c_i issued a cmd to the InterSched
Idle ($BCmdQ_{i,k}^I$)		Cycles in which c_i has no requests in the BCmdQ

Fig. 5. BCmdQ contention breakdown for core c_i .

Hence, c_i 's PRE command is not issued but delayed to issue a younger request's command. Another situation we account as CR delay is when a request from c_i needing to issue its PRE command is the oldest ready command but it is not the oldest request in the BCmdQ, i.e. there are other older requests in the BCmdQ but they cannot issue their DRAM commands due to timing constraints. In this situation, the PRE command is not issued to prevent it from causing page misses on the older requests and contention is ascribed to the core of the oldest request in the BCmdQ, c_j . Note that, in the first case CR is allowed to favor page hits and in the second CR is prevented with the same aim. In both cases, $BCmdQ_{i,k}^{CCRj}$ is set to one.

Timing Constraints (TC). c_i has a command that is not ready due to compliance with JEDEC timing constraints. As introduced in Section III, per-bank timing tables are implemented to enforce timing constraints. In order to track TC contention among cores, MCTrack adds coreID tables associated to the per-bank timing tables to record the ID of the core (c_j) that prevents a command of every type (PRE, RD, WR, ACT, etc.) from being issued in the current cycle. When c_j issues a command, timing tables are updated along with the bank state of the IntraSched selected by the InterSched. The timing table is populated with the cycle in which a command of a given type can be issued, and the coreID table with the ID of the core, c_j , that issued the command causing these timing changes. Hence, if c_i later tries to issue a command and cannot do it because of a timing constraint caused by c_j , c_i records this as $BCmdQ_{i,k}^{CTCj}$.

With MCTrack, each contention cycle due to TC is effectively counted once. Ascribing TC cycles to the different cores is affected by MCTrack complexity. We consider an ideal MCTrack implementation and a more realistic one. We explain both in more detail in the examples below.

Example 1 (Overlapped Constraints). The first example captures the case of several constraints overlapping in time such that once the command satisfies the constraint causing the maximum delay, it is implicitly satisfying all the other previous timing constraints.

Our more realistic implementation of MCTrack builds on commodity MCs' implementation of the timing machinery that always updates the timing table with the latest cycle at which this command becomes ready. For instance, let's assume the sequence of commands in the top part of Figure 6 and consider

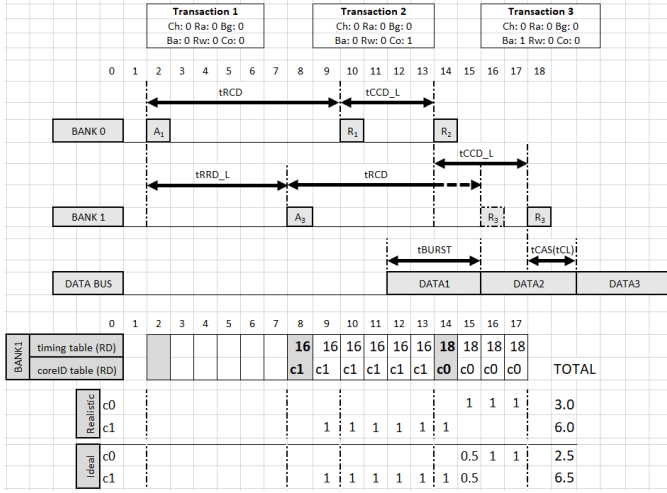


Fig. 6. RD entry in the timing Table of Bank 1 and contention tracking, both as MCTrack records contention and the ideal approach.

the entry for RD commands in the timing table and the coreID table of Bank B1, whose contents are shown in the bottom part of Figure 6. Further assume that A_1 , R_1 and R_2 are issued by core c_0 and A_3 and R_3 by c_1 .

- In cycle 8, c_1 issues A_3 and the RD entry of B1 is updated with $8 + tRCD = 16$ so as to capture that a RD command has to fulfill the $tRCD$ constraint. The RD command R_3 starts to be delayed by $tRCD$ from cycle 9 onwards.
- In cycle 10, c_0 issues R_1 to B0. B1's timing table is not updated with an inter-bank RD-to-RD dependency since $10 + tCCD_L = 14$, which is smaller than the current cycle value in B1's timing table of $8 + tRCD = 16$. Hence, cycles 10-13 are ascribed to c_1 .
- In cycle 14, c_0 issues R_2 to B0. B1's timing table is updated with an inter-bank RD-to-RD dependency, since $14 + tCCD_L = 18$ is greater than the current cycle value in B1's timing table of $8 + tRCD = 16$. Contention is ascribed to c_0 from cycle 15 onwards.

With the ideal MCTrack implementation, cycles 11-13 are ascribed to c_1 since the constraint $tCCD_L$ caused by R_1 issued by c_0 does not affect the timing of c_1 's R_3 command. That is, there is no contention. Instead, when c_0 issues R_2 , the corresponding $tCCD_L$ does affect R_3 's timing behavior. In that case, in cycle 15, due to the two active constraints, contention is split among both cores (each core is assigned 0.5 for that cycle).

In general, ideal MCTrack splits contention among cores better. However, it requires non-trivial extensions of the timing constraint tables to track multiple levels of overlapping constraints. If the additional hardware cost is affordable, the ideal solution is preferred. By default, and to reduce hardware complexity, MCTrack builds upon a timing table that keeps only one timing constraint, as usual in commodity MC. In the example above, MCTrack realistic produces very similar results to MCTrack ideal ascribing c_1 contention to c_0 (6 cycles vs 6.5 cycles) and c_1 (3 cycles vs 2.5 cycles)

Example 2 (Four-Bank Activation Window): This example

TABLE I
ETC DESCRIPTION FOR THE BCMDQ. 1+, <B and 1_{hpri} INDICATE THAT AT LEAST ONE QUEUE, NOT ALL QUEUES, OR THE HIGHEST PRIORITY COMMAND MATCH THAT CONDITION FOR CORE c_i RESPECTIVELY.

row	issued	issuable	idle	NER	NRA	CR	TC	ETC to set to 1
①	1	-	-	-	-	-	-	$BCmdQ_i^R$
②	0	0	B	-	-	-	-	$BCmdQ_i^I$
③	0	1+	<B	1+	1+	-	-	$BCmdQ_i^{CNER_j}$
④	0	1+	<B	0	1+	-	-	$BCmdQ_i^{CNRA_j}$
⑤	0	0	<B	0	0	1_{hpri}	-	$BCmdQ_i^{CCR_j}$
⑥	0	0	<B	0	0	0	1_{hpri}	$BCmdQ_i^{CTC_j}$

captures the four ACT window constraint ($tFAW$), which limits the number of issued ACT commands to DRAM to four every $tFAW$ cycles. Associating the extra delay due to $tFAW$ to the core c_j of the last ACT unfairly assigns more contention to c_j and less contention to the other cores causing the other three ACT to be sent. While MCTrack ideal would ascribe contention delay more fairly by assigning 0.25 of every cycle to the core issuing each ACT, it complicates the timing machinery and comes at the expense of complex logic and area overhead. Hence, we advocate for MCTrack realistic to track the latest ACT command to trigger an update to the timing table as the sole source of contention, as explained earlier. For completeness, in the evaluation Section we compare the realistic and ideal variants of MCTrack and show that the difference among them ascribing contention is minimal.

B.2. ETC for all BCMDQs

We combine the ETC for each $BCmdQ_k$ to produce a single ETC for all BCMDQs by following several principles, see Table I. In this section, we do not use the subindex k when referring to the combined ETC components for all BCMDQs.

A core c_i is progressing (running) in a given cycle if the InterSched selects a BCMDQ that issues a command from c_i . While there can be other commands in other BCMDQs in any other state (e.g. under contention) or idle for c_i , the fact that c_i issues a command to the DFI implies that c_i made progress in that cycle and hence, we count it as $BCmdQ_i^R$. In other words, if $BCmdQ_{i,k}^R$ is set for any k , then $BCmdQ_i^R$ is set to 1. This is captured in row ① of Table I that shows that there was a BCMDQ for which a command from c_i was issued.

If no $BCmdQ_k$ has a command from c_i , that cycle is accounted as idle for c_i , setting $BCmdQ_i^I$ to one. This scenario is depicted in row ② of Table I.

If there are 1 or more (1+) issuable commands from c_i in any $BCmdQ_k$, at least one of them is selected by the corresponding IntraSched, but the InterSched selects a command from c_j , the cycle is accounted as $BCmdQ_i^{CNER_j}$ (see row ③ of Table I). If none of the issuable commands is selected by any IntraSched, the cycle is accounted as $BCmdQ_i^{CNRA_j}$, where c_j is the core selected in the IntraSched not selecting the highest priority issuable command from c_i (see row ④).

If in all BCMDQs the requests of c_i are not issuable, i.e. their state is CR or TC, that cycle is accounted in the contention subcategory of the highest-priority command. That is, if the highest-priority c_i command is stalled due to CR then

R	I	F	C						
			RReqQ				BCmdQ		
			WB	FU	IC	TC	CR	NRA	NER

Fig. 7. MC ETC for core c_i . Contention (C) is tracked per each core.

$BCmdQ_i^{CCR}$ is set to 1, and if it is stalled due to TC then $BCmdQ_i^{CTC}$ is set to 1 (see the last 2 rows of Table I).

C. ETC for the MC

So far we have developed local ETCs for the RReqQ and BCmdQ. In order to produce a single ETC for the entire MC, we combine the information coming from the local ETCs. Note that we do not need contention counters to implement local ETCs, but just signals that are sent to the global ETC logic and counters used to implement the ETC of the MC.

The global ETC for the MC prioritizes the information coming from the ETC of the BCmdQs over those of the RReqQ. In particular, when for a core c_i the BCmdQs are not empty, i.e. there is at least one request from c_i in a BCmdQ, it determines the ETC of the MC.

- c_i is running (MC_i^R) those cycles when it effectively issues a command from any BCmdQ to the DRAM device.
- If in a given cycle c_i is blocked due to contention in a BCmdQ, we account the cycle under MC_i^C and, in particular, the corresponding subcategory of the highest-priority request of the BCmdQs.
- The cycles in which the MC is stalled due to a refresh are accounted as MC_i^F as an additive factor [52]–[54].
- When c_i is idle in the BCmdQs, we look at the status of c_i in the RReqQ to build the global ETC.
 - If c_i is idle in the RReqQ, i.e. it contains no request from c_i , the cycle is accounted as idle (MC_i^I).
 - If c_i issues a request from the RReqQ in a cycle, it is accounted as running (MC_i^R). Hence, c_i is running when (i) issuing from any BCmdQ to the DRAM device or (ii) the BCmdQs are idle and a request from c_i is issued from the RReqQ to a BCmdQ.
 - If requests from c_i are all blocked in the RReqQ, we take the contention category of the highest-priority request.

Figure 7 shows the contention categories for the ETC of the MC. Note that, for instance, a cycle in RReqQ's ETC can be accounted as idle for c_i when RReqQ holds no c_i requests. If in that cycle c_i issues a command from a BCmdQ, the ETC logic for the MC accounts the cycle as running for c_i .

V. IMPLEMENTATION

In this section, we develop a possible implementation of MCTrack. Section V-A provides a general view of MCTrack. Section V-B introduces the MCTrack logic that gathers the output information from the different sources in the MC. Section V-C shows the logic attached to the RReqQ. Likewise, Section V-D covers the logic to attach to the BCmdQ. Section V-E shows the logic of the contention counters.

MCTrack assumes that read/write requests are sent to the memory controller tagged with the coreID. Interestingly,

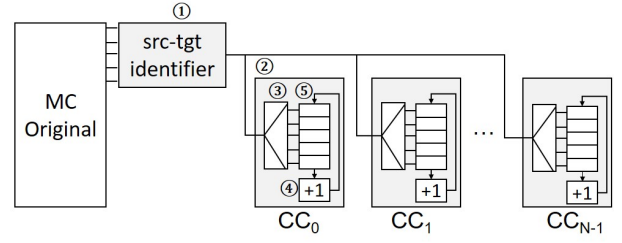


Fig. 8. General architecture of the MCTrack and its integration in the MC.

some cache proposals already include per-line owner (coreID) bits [55]. This allows to use the ID of the core causing an eviction in the last-level cache to tag the corresponding memory request. At the interconnect level, AMBA CHI provides a source ID field that encodes the ID of the core (source) generating the request [56].

A. MCTrack Implementation

In order to limit intrusiveness, MCTrack gathers information already present in the MC and routes it to the MCTrack logic. This does not alter the functionality of any of existing MC functional blocks, so the MC operation remains unaltered.

MCTrack, see Figure 8, comprises a src-tgt logic ① attached to the MC to identify the source core generating the contention and the target core suffering the contention, and hence, the particular contention counter to increment every cycle for each core. MCTrack also comprises a contention counters (CC_i) module per core ②. Each CC module is broken down into a multiplexor ③ that, based on the inputs provided every cycle by the src-tgt module, selects a particular contention counter ⑤ that is incremented by one ④ and written back. As each execution cycle of the task in core c_i is counted exactly once, the produced output is an execution time breakdown in the MC like the one illustrated in Figure 7.

B. MCTrack Per-Core State Determination

Cycles are classified in a given category hierarchically as presented in Section IV-C. Each queue and arbiter propagates its local ETC information to the global ETC that combines it as presented next.

Idle state. Only when no request from a given core exists in the RReqQ and each BCmdQ, the idle state signal is set and the rest (running and contention ones) are unset. Hence, the logic merging the idle signal coming from each module simply consist of an AND gate as shown in Figure 9(a).

Running state. We check whether a core c_i is running in a cycle by comparing c_i to the core ID of the command issued by the InterSched to the DFI (if any), and if c_i is idle in the BCmdQs, we also compare c_i with the core ID of the request issued by the ReqQ-BCmdQ arbiter to the BCmdQs (if any), see in Figure 9(b). If any of those comparisons results in a match, the running signal for c_i is set. Note that in Figure 9(b), $BCmdQ.idle.cID$ is the AND of all *idle* signals from all BCmdQs.

Refresh state. We simply assume a *refresh* signal that is set when the MC starts performing a refresh and it is unset

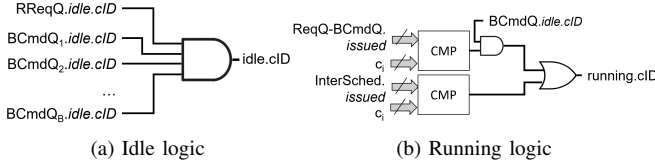


Fig. 9. MCTrack global logic for idle and running.

when the refresh period is finished. While this signal is set, we count for every core each cycle as refresh.

Contention states. When c_i is neither in idle nor in running state, one of the contention signals (that corresponds to a particular ETC Contention category) is set.

For the RReqQ, Figure 10(a), the contention information for c_i ($RReqQ.info.cID$) is passed to the contention counters module if (i) c_i is idle in all BCmdQs, (ii) it is not in running in the RReqQ, and (iii) it is not idle in the RReqQ.

Figure 10(b) shows the logic for BCmdQ contention information propagation, which requires that c_i is not running in the BCmdQ and it has at least one request in one BCmdQ. Note that $RReqQ.info.cID$ and $BCmdQ.info.cID$ contain the type of contention experienced in the corresponding queue, as well as the cID of the core causing contention on c_i .

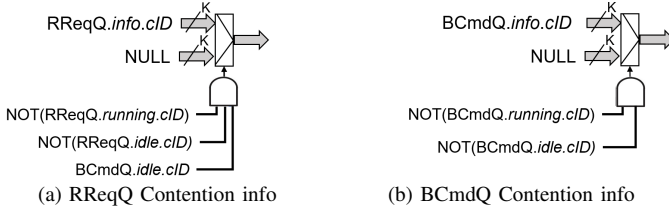


Fig. 10. MCTrack global logic for contention tracking.

C. RReqQ Contention

RReqQ idle, running and contention information per core is produced every cycle by the RReqQ MCTrack module.

Each of the R RReqQ entries contains the target address, which we assume to have 40 bits like in [57], [58], and the ID of the core generating the request (scID) which is $\log_2(N)$, where N is the number of cores, see Figure 11(a). Along with that information, a k -bit priority (prio) field is included to prioritize ready requests to schedule them to the BCmdQs. In some implementations, such priority field can be based on timestamps if requests are scheduled based on age.

MCTrack adds a *ContInfo* table with as many entries as the RReqQ (Figure 11(b)) with several fields per entry. First, *ISSA* (issuable) is a 1-bit field defining whether the request in this entry is issuable. It is updated when checking whether the target BCmdQ of the request in the entry is full (not issuable) or not full (issuable). The ReqQ-BCmdQ arbiter tracks this information to determine which request is issued (*ISSE*) each cycle. And third, *CSB* is a $\log_2(N)$ bit field containing the ID of the core stalling the target BCmdQ of the request in this entry (data forwarded from the target BCmdQ to the RReqQ). When the request is not issuable ($ISSA = 0$), it is because its target BCmdQ is full. Only in that case the contents of *CSB*

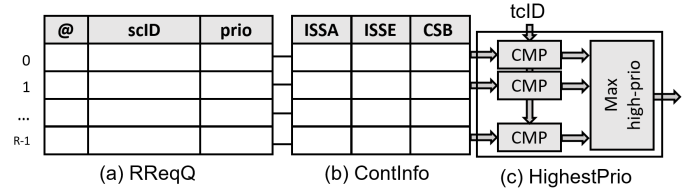


Fig. 11. Fields in the RReqQ (a), fields in the added *ContInfo* table (b), and contention logic (c). The logic to determine idle and running have not added in this figure for sake of clarity.

are valid. From all issuable entries ($ISSA = 1$) only one at most is actually issued ($ISSE = 1$).

Idle Logic. In order to implement the logic to produce the idle signal for the RReqQ for a given core ($c_i = tcID$), referred to as $RReqQ.idle.cID$ in Figure 9(a), the MCTrack RReqQ module adds logic to the *ContInfo* table. The logic checks whether all entries in the RReqQ have a coreID that is different from $tcID$. As soon as there is an entry belonging to $tcID$, the $RReqQ.idle.cID$ is unset this cycle. The logic we used to that end is instantiated in Figure 12(a) assuming 8 cores so $\log_2(8) = 3$ bits are used to encode $scID$ and $tcID$. $scID$ is ANDed at bit level with $tcID$. The core-ID-matching logic ($cIDmatch$) is replicated for each of the R entries of the RReqQ+*ContInfo* tables, Figure 12(b).

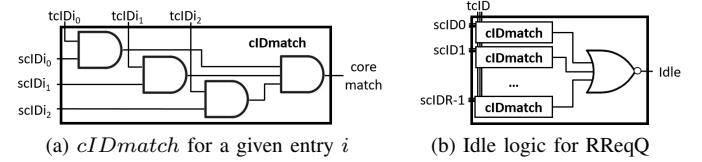


Fig. 12. RReqQ $cIDmatch$ and Idle logic for the RReqQ.

Running Logic. To produce the running bit for the RReqQ, referred to as $ReqQ-BCmdQ.issued$ in Figure 9(b), the needed logic simply checks whether any entry of $tcID$ has $ISSE = 1$. This is implemented with the logic in Figure 13.

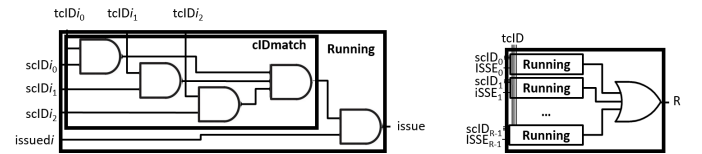


Fig. 13. RReqQ $cIDmatch$ and Running logic for the RReqQ.

Contention Logic. The logic to produce contention information in the RReqQ, $RReqQ.info.cID$ in Figure 10 is sketched in Figure 11(c). It entails selecting the highest-priority request of a given core. This is implemented with the CMP block that includes a $cIDmatch$ block illustrated in Figure 13 so only those entries belonging to the target core ($tcID$) are selected. The information of those entries is sent to a priority comparator that selects the one with the highest priority. For that entry (i) if $ISSA = 1 \& ISSE = 0$ the cID of the core selected by the ReqQ-BCmdQ arbiter is ascribed IC, (ii) and if write batching is active, the cID of the core performing the write is ascribed WB, and (iii) if $ISSA = 0 \& ISSE = 0$ the core in *CSB* is causing FU contention. Hence, overall the logic reduces to few gates.

TABLE II
CACHE AND MEMORY CONFIGURATION PARAMETERS WE USE.

Cache Parameters			Memory Structural Parameters	
Parameter	L1	L2	Parameter	DDR4
Cache Size	64KB	1024KB	Bankgroups	2
Sets	128	1024	Banks_per_group	4
Associativity	8	16	Ranks	2
Bank	1	1	Rows/Columns	65536/1024
Line size	64	64	BL	8

CSB generation. Requests in the RReqQ may experience contention due to the target BCcmdQ being full. To determine the CSB in that case, a.k.a. the core causing such contention, MCTrack builds on intermediate arbitration information from the BCcmdQs. In particular, if a command in $BCcmdQ_k$ is sent to the DFI by the InterSched arbiter, the owner of that command is the CSB for all RReqQ requests targeting $BCcmdQ_k$. For RReqQ requests targeting any different BCcmdQ, the CSB is the owner of the highest priority request in the corresponding BCcmdQ.

D. BCcmdQ Contention

The logic per BCcmdQ to produce the idle and running signals is similar to that of the RReqQ so it is omitted. Regarding the contention information, Figure 14 shows how it is propagated from the BCcmdQs for each core. For each command not sent to the DFI (so not running) the cause of its contention is obtained (TC, CR, NRA, NER) as well as the core causing such contention. We keep this information in a *ContInfo* table, similar to that in Figure 11(a), attached to the BCcmdQs along with the *cID* of the command, and the priority of the request originating that command (the priority is used to schedule the highest priority command first).

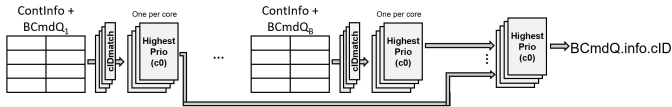


Fig. 14. BCcmdQ contention information propagation.

Regarding contention information propagation, for each BCcmdQ and core c_i , there is a *cIDmatch* (see Figure 12(a)) allowing only c_i commands to reach the corresponding *HighestPrio* module. Each *HighestPrio* module selects the highest priority command per BCcmdQ and core (if any) and propagates its contention information to the global *HighestPrio* modules, which select the contention information with highest priority across all BCcmdQs for each core. Command information includes the type of contention experienced by the command, and the core ID of the core causing such contention. That information is the input tagged as *BCcmdQ.info.cID* in Figure 10(b). Note that the arbitration used to select the command to send to the DFI uses *HighestPrio* modules, similar to the ones for contention information propagation but to select the highest priority command among all BCcmdQs.

TABLE III
JEDEC DRAM TIMING PARAMETERS (PARAM.) [46]. VALUES SHOWN IN CYCLES. S/L REFERS TO SHORT (DIFFERENT BANKGROUP) AND LONG (SAME BANKGROUP), RESPECTIVELY.

Param.	Description		Param.	Description	
$tRCD$	ACT to CAS delay	22	$tRTP$	Read to PRE Delay	9
tRL	RD to Data Start	17	$tCCD$	CAS to CAS delay	4/6
tWL	WR to Data Start	12	$tRRD$	ACT to ACT delay	4/6
tRP	PRE to ACT Delay	17	$tRTW$	RD to WR Delay	10
$tRAS$	ACT to PRE Delay	39	$tWTR$	WR to RD Delay	3/9
tWR	Data end of WR to PRE	18	$tFAW$	4 activation window	26

E. MCTrack monitors

The complexity of the counters that implement the EMs varies with the degree of detail wanted for the contention information collected. One may collect detailed information, as described in Section IV, which requires as many counters per core as categories considered for the cycles. For instance, if we want maximum detail, for each core we would need 1 counter for running, 1 for idle, 1 for refresh and N for each type of contention where N is the core count in the most detailed scenario. Alternatively, if we want the minimum detail possible, we could use just 4 counters: 1 for running, 1 for idle, 1 for refresh and 1 for contention (regardless of its cause). In any case, at every cycle one counter per core is read and incremented by 1, which incurs limited activity (e.g. orders of magnitude lower energy than an off-chip DRAM access).

VI. EVALUATION

A. Experimental framework

We use the gem5 [59] modular platform to model a multi-core system with local instruction and data caches connected to the shared L2 cache, see cache configuration in Table II. We also use DRAMsim3 [50], a reference memory simulator used for computer-system architecture research. We model a 8GB, single-channel DDR4-3200 DRAM module with the configuration in Table II and the timing constraints in Table III.

We build on *synthetic* and *representative benchmarks*. For the former, we have complete information about their memory access patterns, so that we can make informed guesses on the impact they may have on each other when accessing memory. We contrast our clear a priori contention breakdown with the breakdown generated by MCTrack to assess its correctness. Regarding *representative benchmarks*, we focus on two baseline operators in artificial intelligence and image related functionalities used in real-time systems for autonomous operation. In particular, we consider image-to-columns (I2C), used for tensor lowering to enable matrix convolutions needed by neural networks, and matrix multiplication (MM) that are pervasively used in functionalities like object detection or path planning (e.g., for matrix convolutions).

For the evaluation of MCTrack we cover two main metrics (i) its hardware complexity and (ii) its accuracy to single out the root causes of contention tasks suffer in the main memory.

B. Complexity and size assessment

MCTrack incurs very limited overheads for (i) contention counters to track contention, (2) state tables to hold request-

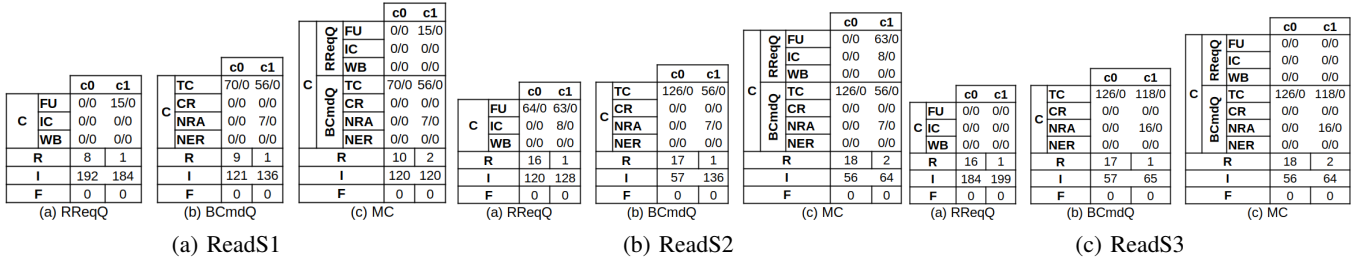


Fig. 15. RReqQ breakdown in scenarios Reads1, Reads2, and Reads3. Each cell contains a pair x/y , where x is the delay generated by $c0$ and y by $c1$.

and command-related information such as highest-priority information and core IDs, and (3) simple logic to decide the counters to update based on limited casuistic. Moreover, since MCTrack logic has no functional effect, it is out of the critical path and can be pipelined. This translates into a few cycles of delay in tracking contention since the cycle it is produced. However, the impact is negligible since EMs are read at a much higher granularity, e.g. micro- and milli-seconds.

Storage. For an 8-core system, MCTrack needs 59 counters per core to track running, idle, refresh, and the contention categories. For each core, three counters for running, idle and refresh, and 7 counters for contention: 3 for the RReqQ (WB, FU, IC) and 4 for the BCmdQs (TR, CR, NRA, NER). To track the cores causing contention we need 8 counters for each contention category. So $3+7 \cdot 8 = 59$ counters in total per core. Assuming 40-bit counters (5 bytes), the area in bytes amounts to $5 \cdot 8 \cdot 59 = 2360$ bytes. *ContInfo* in the RReqQ requires 5 bits per entry (1-bit *ISSA*, 1-bit *ISSE*, and 3-bit *CSB* for 8 cores). A 32-entry RReqQ amounts to 20 bytes. Each coreID table has 8 3-bit entries, and since there is a coreID table for all 16 BCmdQs, 48 bytes are required overall. Finally, *ContInfo* of a BCmdQ includes as many entries as the BCmdQ queues, 8 in total. Each entry keeps analogous information to that of the RReqQ's ones, 5 bits. Note that we assume the k -bit priority field to be part of the BCmdQ itself since it is needed for command selection. Overall, 5 bits per-entry times 8 entries per BCmdQ times 16 BCmdQs results in 80 bytes. Hence, MCTrack requires $2360 + 20 + 48 + 80 = 2508$ bytes.

Logic. MCTrack counters include one incrementer per core, each one implemented as a full-adder for the increment, and half-adders for the carry propagation. Assuming 8-entry BCmdQs, each *HighestPrio* module includes 8 3-bit core ID comparators, and a tree of 2-input comparators and 2-input multiplexers to compare priorities of each pair of entries and select the information of the entry with highest priority. The tree would have 7 instances of the logic to move from 8 to 4 entries (4 instances), from 4 to 2 (2 instances), and from 2 to 1 (1 instance). Since MCTrack requires as many *HighestPrio* modules as the number of BCmdQs (including the global *HighestPrio*) $\times$ number of cores + RReqQ's modules $\times$ (17 $\cdot$ 8+8), we can assume 144 such modules, so $7 \cdot 144 = 1008$ comparators and multiplexers. Each comparator operates on the k bits of the prio fields, and each multiplexer selects the 8 bits of the *ContInfo* entry plus the k bits of the corresponding prio. Assuming $k = 16$, which allows setting up to 65536

different priority levels, we have 1008 16-bit comparators and 1008 24-bit multiplexers. A multiplexer can be built using 4 2-input NAND gates per bit, so 1008 multiplexers require $1008 \cdot 24 \cdot 4 = 96768$ 2-input NAND gates. Each 2-input 16-bit magnitude comparator² requires 36 AND, 16 XNOR, and 8 OR gates if built using 4 4-bit cascading comparators, hence 60 gates per comparator, and $1008 \cdot 60 = 60480$ gates.

Transistors. If we account for the number of transistors of each storage bit and gates, we would need less than 8 transistors per storage bit on average, so $8 \cdot 2508 \cdot 8 = 160512$ transistors. Incrementers require less than 8K transistors, and 866K transistors for the *HighestPrio* modules leveraging the number of transistors per gate. Overall, the total transistor count is around 1136K. This compares to, for instance, an embedded Qualcomm Snapdragon 835 CPU with 3B transistors, which is less than a 0.04% transistor count increase.

C. Evaluation with Synthetic Benchmarks

For these experiments we show the local ETC for the RReqQ, BCmdQ, and the global MC ETC. Showing local ETC provides evidence that they correctly capture contention in the local resource (RReqQ and BCmdQ). Subsequently, in the case studies we just focus on MC (global) ETC.

Reads1. For this example, we assume a 32-entry RReqQ and an 8-entry BCmdQ. We further assume tasks are running in 2 cores ($c0$ and $c1$). After $c0$ sends 8 read requests to the same row and bank, $c1$ sends a read request to that bank. Requests are sent to the MC in consecutive cycles: $c0$ sends its requests in cycles 0-7 and $c1$ sends its request in cycle 8.

The ETC for the RReqQ is shown in Figure 15(a). The requests from $c0$ are issued to the RReqQ as they arrive and from there to the BCmdQ without delays. The ninth request (the one from $c1$) must wait until there is a free entry in the BCmdQ, so it suffers 15 cycles of FU contention that represents the cycles $c1$'s request was issuable in the RReqQ but could not be issued to the BCmdQ due to $c0$'s requests. $c0$ has 8 Running cycles and $c1$ has 1 Running cycle.

The ETC for the BCmdQ, shows high TC contention (70 cycles) for $c0$, caused by its own RDs, affected by ACT-to-RD and RD-to-RD dependencies. $c1$ incurs high TC contention (56 cycles) since its RDs suffer from RD-to-RD dependencies caused by $c0$'s older RDs. $c1$ shows 7 cycles of NRA contention caused by $c0$'s 7 older RDs since they also become

²Magnitude comparators provide which input is larger than the other, as opposed to equality comparators that only indicate whether inputs are equal.

ready after RD-to-RD dependencies elapse. c0 has no NRA contention since it is considered as progressing when one of its requests issues a command.

The ETC for the MC shows that the only contention suffered by c0 comes from TC. We also observe 10 running cycles for c0 since the first read request arriving to the MC can progress (no BCmdQs stalls) and contribute to the running global category, while the remaining 9 cycles correspond to the running cycles of the BCmdQ once read requests have entered it. For c1 we observe the same contention numbers as in the RReqQ and BCmdQ ETCs, and 2 running cycles, one for the RReqQ progression and the other for the BCmdQ.

Reads2. In this scenario we keep the same RReqQ/BCmdQs sizes but increase the number of c0 requests to 1. Figure 15(b) shows for the RReqQ how Running cycles increase accordingly with c0's 16 requests. In the BCmdqQ FU and TC cycles increase significantly for c0, in both cases due to c0 itself. c1 maintains the same numbers as in Reads1, since it can only be affected by the previous 7 requests from c0 present in the BCmdQ. FU cycles for c1 increase because of the higher occupancy of the BCmdQ and its IC cycles amount to 8, corresponding to the 8 requests of c0 that precede c1's.

Reads3. This scenario also considers 16 requests for c0, but the size of the BCmdQ has been increased to 128 positions. Figure 15(c) shows that, no core incurs any FU contention cycles, due to the increased capacity of the BCmdQ. Since all requests coexist together in the BCmdQ and, for this same reason, contend on each other, TC contention cycles are significant for both cores (always caused by c0) and c1's RD command is preempted by c0's older 16 requests.

D. CaseStudy1: Software memory mapping techniques

One of the most straightforward software-level approaches to mitigate contention consists in controlling memory mapping to ensure that tasks running in different cores go to a distinct set of banks and bankgroups [44]. However, the challenge for certification lies on the amount (and trustworthiness) of evidence that can be provided on the correctness of any proposed mechanism. While timing can be used as an indirect metric to capture different degrees of contention, it cannot be used directly for building a solid argument on bank mapping. In fact, the relation between timing and bank-related contention is nonlinear, with end-to-end timing observations potentially including different and nonuniform effects. In this scenario, the only practical way to provide the necessary evidence is by analysis: first, a description of the documentation analyzed to identify the bits used to select bank and bankgroup; and second, a review of the code forcing tasks to go to the desired bank (i.e. code qualification). This evidence requires time to be provided and, in general, is feeble.

MCTrack provides more solid evidence for the certification arguments without building on code review. In particular, we can use an analysis task performing only reads and pin it to c0 and we then run a real contending tasks pin to c1. In each experiment we change the memory mapping with the corresponding software tool so that c0 is mapped to a fixed

c1-on-c0							
RReqQ				BCmdQ			
	sameB	sameBG	sameR		sameB	sameBG	sameR
FU	0	0	0	TC	5600	4900	2100
IC	0	0	0	CR	0	0	0
WB	0	0	0	NRA	0	0	0
				NER	0	0	0

Fig. 16. MC ETCs for different placements of c1's requests

bank while c1 bank mapping varies and is set to: (1) the same bank (SameB) where c0 is mapped to; (2) a different bank of the same bankgroup (SameBG); and (3) a bank in a different bankgroup of the same rank (SameR).

Figure 16 confirms TC has diminishing values when the contender is shifted from executing in the same bank as the analysis task (5600) to the case when it is run on a bank in a different bankgroup (2100). More in detail, c0-on-c1 TC contention for sameBG reduces since c1 does not suffer from c0's tRCD, and the separation of each core's requests in different banks allows for greater interleaving of DRAM commands issued (as opposed to scenario sameB, where c1 must wait until c0's first request issues its ACT and RD commands). Also c0-on-c1 TC contention for sameR shows a big drop-off in inter-core TC contention, since the RD-to-RD constraint for requests targeting different bankgroups of the same rank is the lowest in value out of all scenarios.

Overall, the hardware support provided by MCTrack allows to design specifically crafted experiments to assess that the memory mapping library works properly, i.e. forcing the requests of certain cores to go to specific banks.

E. CaseStudy2: Bandwidth regulation

Bandwidth regulation schemes [18] build on memory access counts to constrain and regulate accesses to shared memory. Tasks execution time is divided into intervals, each one with a quota of memory accesses, derived either empirically or theoretically, which can be broken down per read and write, per core [10], [18], [35], [36]. The operating system or hypervisor periodically checks whether a core has exhausted its quota, throttling it until its quota is replenished.

In the following, we assess the benefits of using MCTrack as direct contention metric for bandwidth regulation methods, instead of access counts. To that end, we consider a set of contention scenarios in which we run variants of a synthetic R/W task (v1 to v9) with different interarrival times and clustering of requests, both for the task under analysis and contender task. Without loss of generality, we focus on a single monitoring interval and, for bandwidth regulation, we set an access count (AC) threshold of 100 accesses. We compute the corresponding contention impact threshold for MCTrack (MCT) by considering the average latency incurred by a conflicting access to memory (empirically determined in 110 cycles), thus obtaining the cumulative contention threshold of 110×100 cycles. Note that in reality, such bound could be more conservative taking the maximum or a high quantile, which will provide further benefits to MCTrack.

In Figure 17(a) the Y axis shows the progress made by the contending task by the time it is stalled by AC or MCT. We see

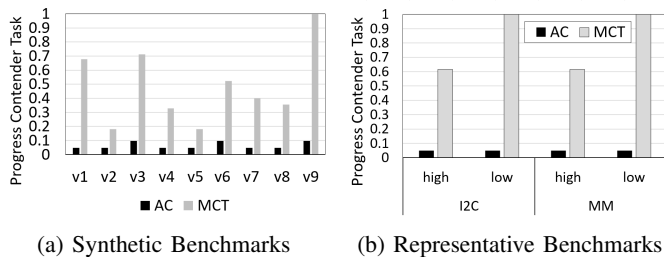


Fig. 17. Progress of the contending task when throttled by AC and MCT.

that for all 9 variants AC stalls the contending task prematurely, which does not reach 10% progress in any experiment. Instead, MCT stalls the contending task only when the contention it generates on the task under analysis reaches the desired cycle threshold: the minimum progress the contender task does is 18% and the maximum 100% (i.e. the task is not stalled), 48% on average compared to the 7% of AC. This occurs because the use of access counts as indirect metric implies that overly-conservative assumptions must be taken in order to capture and bound the contention impact as not all requests are actually causing contention at run time. With MCTrack we provide a direct metric for contention tracking that can be exploited by existing bandwidth regulation schemes. Our results show that building on MCTrack measured contention favors the contending task by guaranteeing a (7x average) shorter latency compared to AC methods.

Figure 17(b) shows results for the representative benchmarks (I2C and MM). In this case each benchmark is run against a R/W benchmark that has two variants. In both variants it makes the same number of accesses. In the first variant the contender task has a high-density of requests, while in the second the density is low. We set an AC threshold of 100 accesses and the corresponding MCT threshold of 110×100 .

For both benchmarks, the low-density variant causes less impact on each benchmark, as it happens to have its accesses more spread in time. MCT captures this and does not require to stall the contender task, allowing it to make full progress. For the high-density variant, the impact on each benchmark is greater, requiring to throttle the contending task when it has made a progress of around 60%. Instead, AC systematically stalls the contending task when it performs 100 out of its 2048 accesses, no matter the impact it causes on the analysis task (i.e. the representative benchmark).

For completeness, we assessed an implementation of MCTrack ideal on the I2M and MM setups. The resulting difference in accounting between MCTrack ideal and realistic is less than 1% when considering the high-density contender variant, and less than 500 cycles for the low-density one.

VII. RELATED WORKS

1) *Predictable DRAM subsystems*: A plethora of recent solutions target at providing predictability for DRAM subsystems due to their large and variable delays [60]. We categorize these solutions into two main directions. The first one's goal is to analyze the COTS DRAM subsystem to provide

latency bounds [47], [49], [61], [62] without imposing any hardware modifications. This analysis depends on collecting few performance monitoring counters to represent the running applications including number of reads, writes, row hits, and row misses [49]. We believe the approach we propose in this paper by tracking contention components and not only high-level metrics can benefit these works to provide tighter bounds instead of accounting for the worst-possible scenarios. The second direction takes a more invasive approach by re-designing the memory controller to ensure predictability [63]–[77]. We find that these works tighten latency bounds at the expense of degrading performance by disabling most of the optimizations found in COTS systems.

2) *Contention tracking*: Few works have proposed specific hardware support for contention tracking, but focusing on the interconnects rather than on the memory controller [45]. This is the case of the Contention Cycle Stack (CCS) unit [78], targeting AMBA AHB interfaces, and realized in a commercial space multicore [27]. In general, interconnect-specific mechanisms tend to be simple owing to the full request serialization upon experiencing contention.

3) *Software support*: Several proposals build on access counts to monitor at run-time the contention incurred by a task on its co-runners and possibly take corrective actions (i.e. throttling) in case tasks exceed the allocated bandwidth [10], [18], [35], [36], [79]. The different variants of bandwidth validation schemes, either building on bespoke implementation or specialized hardware support [25], [26], can effectively provide guaranteed performance for critical tasks. They have been successfully deployed on representative COTS targets in the critical domain (e.g. NXP T2080, Zynq US+, etc.). However, the fact that access counts is an indirect metric for contention can lead to overall performance degradation [36]. The combination of access counts and memory controller bandwidth information can be exploited to partially prevent the enforcement of overly conservative bandwidth thresholds [36].

VIII. CONCLUSIONS

The quantification of the impact of contention on execution time is mandatory for the use of MPSoCs for real-time systems. State of the art approaches have often built on access counts for contention tracking and analytical upperbounding. MCTrack goes beyond access counts and focuses on direct contention metrics, which allows to accurately and fairly track the contention caused by a task on its co-runners. In this respect, MCTrack can be exploited by existing bandwidth regulation approaches. Moreover, MCTrack also enables a paradigm shift for contention tracking and regulation approaches by allowing monitoring the contention suffered by a task rather than just estimating the contention caused on others. MCTrack simplifies the analysis and testing approach to produce the required evidence according to the objective of the applicable safety standard (e.g. A(M)C 20-193 for avionics) of the timely execution of multicore software.

REFERENCES

- [1] D. Radack, H. Tiedeman, and P. Parkinson, "Civil Certification of Multi-core Processing Systems in Commercial Avionics," 2018.
- [2] E. Francis, "Autonomous cars: no longer just science fiction," *Automotive Industries*, vol. 193, 2014.
- [3] NVIDIA Corporation, "NVIDIA DRIVE PX. Scalable supercomputer for autonomous driving," <http://www.nvidia.com/object/drive-px.html>.
- [4] P. Burgio, M. Bertogna, N. Capodici, R. Cavicchioli, M. Sojka, P. Houdek, A. Marongiu, P. Gai, C. Scordino, and B. Morelli, "A software stack for next-generation automotive systems on many-core heterogeneous platforms," *Microprocessors and Microsystems*, vol. 52, pp. 299–311, 2017.
- [5] I. S. Olmedo, N. Capodici, and R. Cavicchioli, "A Perspective on Safety and Real-Time Issues for GPU Accelerated ADAS," in *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, 2018, pp. 4071–4077.
- [6] J. Fernández, J. Perez-Cerrolaza, I. Agirre, A. J. Calderon, J. Abella, and F. J. Cazorla, "On the safe deployment of matrix multiplication in massively parallel safety-related systems," *Applied Sciences*, vol. 12, no. 8, 2022.
- [7] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, "Timing of autonomous driving software: Problem analysis and prospects for future solutions," in *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020, Sydney, Australia, April 21-24, 2020*.
- [8] A. Saeed, D. Hoornaert, D. Dasari, D. Ziegenbein, D. Mueller-Gritschneider, U. Schlichtmann, A. Gerstlauer, and R. Mancuso, "Memory latency distribution-driven regulation for temporal isolation in mpocs," in *35th Euromicro Conference on Real-Time Systems, ECRTS 2023, July 11-14, 2023, Vienna, Austria*, ser. LIPIcs, A. V. Papadopoulos, Ed., vol. 262. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 4:1–4:23.
- [9] A. Serrano-Cases, J. M. Reina, J. Abella, E. Mezzetti, and F. J. Cazorla, "Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC," in *33rd Euromicro Conference on Real-Time Systems, ECRTS 2021, July 5-9, 2021, Virtual Conference*, ser. LIPIcs, vol. 196. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 3:1–3:26.
- [10] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, "Profile-driven memory bandwidth management for accelerators and CPUs in QoS-enabled platforms," *Real Time Syst.*, vol. 58, no. 3, pp. 235–274, 2022.
- [11] J. P. Cerrolaza, R. Obermaier, J. Abella, F. J. Cazorla, K. Grütner, I. Agirre, H. Ahmadian, and I. Allende, "Multi-Core Devices for Safety-Critical Systems: A Survey," *ACM Comput. Surv.*, vol. 53, no. 4, aug 2020.
- [12] FAA/EASA, "Acceptable Means of Compliance/Advisory Circular on Use of multi-core processors," Tech. Rep., 2021.
- [13] K. Goossens, J. Dielissen, and A. Radulescu, "Aethereal network on chip: concepts, architectures, and implementations," *IEEE Design & Test of Computers*, vol. 22, no. 5, pp. 414–421, 2005.
- [14] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, G. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.
- [15] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A Template for Composable and Predictable Multi-Processor System on Chips," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, jan 2009.
- [16] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "Flexpret: A processor platform for mixed-criticality systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 101–110.
- [17] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. Cassé, S. Uhrig, I. Guliasvili, M. Houston, F. Kluge, S. Metzlauff, and J. Mische, "Merasa: Multicore execution of hard real-time applications supporting analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, 2010.
- [18] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*. IEEE Computer Society, 2013, pp. 55–64.
- [19] Rapita Systems, *Multicore Avionics Certification for High-integrity DO-178C projects*, 2017. [Online]. Available: <https://www.rapitasystems.com/products/mach178>
- [20] S. H. VanderLeest and S. R. Thompson, "Measuring the impact of interference channels on multicore avionics," in *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, 2020, pp. 1–8.
- [21] F. Boniol, J. Brunel, K. Delmas, C. Pagetti, and V. Jegu, "Modelling and analyzing multi-core COTS processors," in *11th European Congress on Embedded Real Time Software and Systems (ERTS 2022)*, Toulouse, France, Jun. 2022.
- [22] Freescale semiconductor, "QorIQ T2080 Reference Manual," 2016, Also supports T2081. Doc. No.: T2080RM. Rev. 3, 11/2016.
- [23] NXP, "MMDC & NoC Configuration for Optimal DDR3 Performance on the i.MX 6DualPlus/6QuadPlus, Document Number: EB828 Rev. 0, 08/2016," 2016.
- [24] RTCA and EUROCAE, *RTCA DO-330 - Software Tool Qualification Considerations*, RTCA and EUROCAE, 2011.
- [25] Intel, "Intel Resource Director Technology (Intel RDT) on 2nd Generation Intel Xeon Scalable Processors Reference Manual, Rev. 1.0," 2019.
- [26] ARM, "Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture," 2022, ARM DDI 0598D.a (ID042622).
- [27] N.-J. Wessman, F. Malatesta, J. Andersson, P. Gomez, M. Masmano, V. Nicolau, J. L. Rhun, G. Cabo, F. Bas, R. Lorenzo, O. Sala, D. Trilla, and J. Abella, "De-RISC: the First RISC-V Space-Grade Platform for Safety-Critical Systems," in *2021 IEEE Space Computing Conference (SCC)*, 2021, pp. 17–26.
- [28] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2019, pp. 357–367.
- [29] A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Hauptenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, and R. Wilhelm, "Impact of resource sharing on performance and performance prediction: A survey," in *CONCUR 2013 - Concurrency Theory*, P. R. D'Argenio and H. Melgratti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 25–43.
- [30] R. Wilhelm, "Mixed Feelings About Mixed Criticality (Invited Paper)," in *18th Int. Workshop on Worst-Case Execution Time Analysis (WCET)*, ser. OpenAccess Series in Informatics (OASICS), F. Brandner, Ed., vol. 63. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018, pp. 1:1–1:9.
- [31] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla, "Assessing the suitability of the NGMP multi-core processor in the space domain," in *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012*, A. Jerraya, L. P. Carloni, F. Maraninchi, and J. Regehr, Eds. ACM, 2012, pp. 175–184.
- [32] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Vienna, Austria, April 11-14, 2016. IEEE Computer Society, 2016, pp. 161–172.
- [33] R. Pellizzoni and M. Caccamo, "Impact of peripheral-processor interference on WCET analysis of real-time embedded systems," *IEEE Transactions on Computers*, vol. 59, no. 3, pp. 400–415, 2009.
- [34] D. Dasari, B. Akesson, V. Nélis, M. A. Awan, and S. M. Petters, "Identifying the sources of unpredictability in COTS-based multicore systems," in *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013, pp. 39–48.
- [35] GreenHills, "Bandwidth Allocation Management (BAM)," 2023, https://www.ghs.com/news/20200701_INTEGRITY_tuMP_multicore_interference_arm.html.
- [36] A. Saeed, D. Dasari, D. Ziegenbein, V. Rajasekaran, F. Rehm, M. Pressler, A. Hamann, D. Mueller-Gritschneider, A. Gerstlauer, and U. Schlichtmann, "Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 133–145.
- [37] F. J. Cazorla, J. Abella, and E. Mezzetti, "Dissecting Robust Resource Partitioning, Robust Time Partitioning, and Robust Partitioning in CAST-32A," in *Automotive Technical Papers*. SAE International, nov 2021. [Online]. Available: <https://doi.org/10.4271/2021-01-5101>

- [38] I. Agirre, J. Abella, M. Azkarate-askasua, and F. J. Cazorla, "On the tailoring of CAST-32A certification guidance to real COTS multicore architectures," in *12th IEEE International Symposium on Industrial Embedded Systems, SIES 2017, Toulouse, France, June 14-16, 2017*. IEEE, 2017, pp. 1–8.
- [39] ISO, *ISO 21448:2022. Road vehicles — Safety of the intended functionality*, 2022.
- [40] CAST, "Multi-core Processors - Position Paper," CAST-32A, Tech. Rep., November 2016.
- [41] RTCA and EUROCAE, *DO-178C / ED-12C, Software Considerations in Airborne Systems and Equipment Certification*, 2011.
- [42] "ISO 26262(-1/11) road vehicles - functional safety," ISO, 2018.
- [43] NXP, "QorIQ T2080 Reference Manual," 2016, Also supports T2081. Doc. No.: T2080RM. Rev. 3, 11/2016.
- [44] DDC-I, "Deos, a Time & Space Partitioned, Multi-core Enabled, RTOS Verified to DO-178C/ED-12C DAL A," 2023, <https://www.ddci.com/category/deos/>.
- [45] J. Cardona, C. Hernández, J. Abella, and F. J. Cazorla, "Maximum-contention control unit (MCCU): resource access count and contention time enforcement," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, J. Teich and F. Fummi, Eds. IEEE, 2019, pp. 710–715.
- [46] D. S. Standard, "Jedec jesd79-3," 2007.
- [47] H. Yun, R. Pellizzoni, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2015, pp. 184–195.
- [48] M. Hassan and R. Pellizzoni, "Bounding DRAM interference in COTS heterogeneous mpsoes for mixed criticality systems," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2323–2336, 2018.
- [49] —, "Analysis of memory-contention in heterogeneous cots mpsoes," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- [50] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: a Cycle-accurate, Thermal-Capable DRAM Simulator," in *IEEE Computer Architect. Letters*, 2020.
- [51] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.
- [52] X. Pan and F. Mueller, "Hiding DRAM refresh overhead in real-time cyclic executives," in *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*. IEEE, 2019, pp. 1–13.
- [53] B. Bhat and F. Mueller, "Making dram refresh predictable," *Real-Time Systems*, vol. 47, no. 5, pp. 430–453, 2011.
- [54] Z. P. Wu, R. Pellizzoni, and D. Guo, "A composable worst case latency analysis for multi-rank dram devices under open row policy," *Real-Time Systems*, vol. 52, no. 6, pp. 761–807, 2016.
- [55] L. Zhao, R. R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell, "Cachescouts: Fine-grain monitoring of shared caches in CMP platforms," in *16th International Conference on Parallel Architectures and Compilation Techniques (PACT 2007)*, Brasov, Romania, September 15-19, 2007. IEEE Computer Society, 2007, pp. 339–352.
- [56] Arm Ltd., *AMBA @ 5 CHI Architecture Specification*, 2020.
- [57] XILINX, *Zynq UltraScale+ Device. Technical Reference Manual. UG1085 (v2.1)*, 2019.
- [58] freescale semiconductor, "e6500 Core Reference Manual," <https://www.nxp.com/docs/en/reference-manual/E6500RM.pdf>, 2014, E6500RM. Rev 0. 06/2014.
- [59] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
- [60] M. Hassan, "On the off-chip memory latency of real-time systems: Is ddr dram really the best option?" in *IEEE Real-Time Systems Symposium*, 2018.
- [61] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsoes for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.
- [62] 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 *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 145–154.
- [63] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, Mar. 2011, pp. 1–6.
- [64] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A Predictable SDRAM Memory Controller," in *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM Press New York, NY, USA, Sep. 2007, pp. 251–256.
- [65] B. Akesson and K. Goossens, *Memory controllers for real-time embedded systems*. Springer, 2011.
- [66] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst, "A mixed critical memory controller using bank privatization and fixed priority scheduling," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2014, pp. 1–10.
- [67] L. Ecco and R. Ernst, "Improved DRAM Timing Bounds for Real-Time DRAM Controllers Read/Write Bundling," in *Real-Time Systems Symposium*, 2015.
- [68] L. Ecco, A. Kostrzewa, and R. Ernst, "Minimizing DRAM Rank Switching Overhead for Improved Timing Bounds and Performance," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.
- [69] D. Guo and R. Pellizzoni, "A requests bundling DRAM controller for mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*. IEEE, 2017, pp. 247–258.
- [70] J. Jalle, E. Quiñones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla, "A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation of a Space Case Study," in *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*. IEEE Computer Society, 2014, pp. 207–217.
- [71] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni, "A rank-switching, open-row dram controller for time-predictable systems," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 27–38.
- [72] Y. Li, B. Akesson, and K. Goossens, "Dynamic command scheduling for real-time memory controllers," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 3–14.
- [73] R. Miroslou, M. Hassan, and R. Pellizzoni, "Drambulism: Balancing performance and predictability through dynamic pipelining," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.
- [74] M. Paolieri, E. Quiñones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *IEEE Embedded Systems Letters*, vol. 1, no. 4, pp. 86–90, 2009.
- [75] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: bank privatization for predictability and temporal isolation," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '11. New York, NY, USA: ACM, 2011, pp. 99–108.
- [76] P. K. Valsan and H. Yun, "MEDUSA: a predictable and high-performance DRAM controller for multicore based embedded systems," in *Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2015, pp. 86–93.
- [77] R. Miroslou, M. Hassan, and R. Pellizzoni, "DuoMC: Tight DRAM Latency Bounds with Shared Banks and Near-COTS Performance," in *MEMSYS*. ACM, 2021, pp. 4:1–4:16.
- [78] J. Jalle, M. Fernandez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello, and F. J. Cazorla, "Contention-aware performance monitoring counter support for real-time mpsoes," in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016, pp. 1–10.
- [79] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement," in *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*. IEEE Computer Society, 2014, pp. 109–118.