Tracking Coherence-Related Contention Delays in Real-Time Multicore Systems

Roger Pujol*†, Mohamed Hassan‡, Hamid Tabani*†, Jaume Abella*†, and Francisco J. Cazorla*†

* Universitat Politècnica de Catalunya, Spain † Barcelona Supercomputing Center, Spain
‡ McMaster University, Canada

***Abstract*—The prevailing use of multicores in Embedded Critical Systems (ECS) is multi-application workloads in which independent applications run in different cores with data sharing restricted to the communication between applications and the real-time operating system. However, thread-level parallelism is increasingly used, e.g., OpenMP, in ECS to improve individual applications' performance. At the hardware level, we are witnessing increased research efforts to master and improve multicore cache coherence that plays a key role enabling efficient data sharing among threads. Despite these efforts, the limited information provided by performance monitoring counters on cache coherence limits the understanding of coherence's impact on tasks execution time and hence, poses severe constraints to estimate tight worst-case execution time bounds. In this line, this work contributes with an analysis of the impact that cache coherence can have on application timing behavior, and a new set of low-overhead performance monitoring counters that can be used to track the coherence-related contention that different threads can cause on each other when sharing data. Our results show that the proposed performance monitoring counters effectively capture all coherence-related contention that tasks can suffer and hence are key for parallel software timing validation and verification in ECS. Furthermore, they help application optimization by providing key information about data sharing among the application threads.**

***Index Terms*—Cache coherence, multicore real-time systems, contention**

I. Introduction

Multicores are acknowledged as the main hardware technology used to provide the required performance levels in ECS. Multi-application workloads are the preferred approach to exploit multicores so that independent applications can be run simultaneously to increase hardware resource utilization. Multi-application workloads exploit task-level parallelism: applications (tasks) are single-threaded and do not share data with each other while they run simultaneously. Instead, data sharing is restricted to communication between applications through the real-time operating system at software partition boundaries, i.e., the producer sends the data, finishes its execution, and some time later, the consumer is executed and reads the data [1]. However, the need to increase single-application performance calls for exploiting thread-level parallelism (TLP): applications are parallelized into different threads that run concurrently sharing data, which effectively reduces the execution time of the application. As an illustrative example, TLP has been exploited for 3D path planning and stereo navigation across other safety-critical real-time functionalities [31].

In this work, we focus on cache coherence hardware support as the main building block to speed up data sharing. Several works study the main challenges brought by data sharing in ECS. While the original works advocate for limiting data sharing [6], more recent works advocate for allowing data sharing by modifying the cache coherence protocols [14] to ensure that all accesses have predictable timing behavior. Unlike previous proposals, we do not aim to change the coherence protocol or limit data sharing to improve predictability. Designing and validating efficient cache coherence protocol implementations have overwhelming costs [21], and hence, changing those protocols instead of keeping existing implementations can introduce onerous costs. Instead, our work focuses on a cache coherence monitoring counter infrastructure, called Remote Protocol-Contention Tracking (RPCT), around those protocols.

We contend that the ability to understand the coherence-related contention within the threads of an application and across different applications provides a two-fold benefit. First, it delivers key information about how coherence affects software time predictability since tracking the end-to-end timing of an application just reports whether a task violates its assigned timing budget but does not allow to single out the actual source (i.e., the particular other task(s)) behind the software timing violation or whether it is related to coherence contention at all. And second, the values reported by RPCT provides insightful information about how threads of a given application share data, which effectively can be leveraged by the programmer to optimize parallel applications, through finding (and reducing) coherence-related bottlenecks. In contrast to RPCT, current event coherence-related monitors in cache track at most coherence state transitions, which neither allows diagnosing overrun causes, since they fail to track messages with sufficient detail to estimate the actual contention caused properly and its source [26], [22].

Our proposal, RPCT, tracks the contention experienced in the buses for all coherence messages, as well as the contention experienced by the coherence protocol itself, i.e., the responses from remote cores (e.g., when a core requests data owned by another core, the response is tracked as contention for the originator).

The rest of this work is organized as follows: Section II covers the main works on cache coherence in real-time sys-

© Owner/Author | ACM 2023. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *SAC '23: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing,* March 2023, pages 461–470.
http://dx.doi.org/10.1145/3555776.3577631

tems. Section III describes our system model and motivates the need for tracking coherence-related cache contention. Section IV describes our performance counter proposal to track cache contention. Section V shows the results of our proposal on various scenarios. Finally, Section VI presents the main conclusions of this work.

II. Background

While multicores can provide the required performance of ECS, they bring several challenges for software timing validation and verification including managing the access to hardware shared resources among different cores ensuring high performance and predictability. Cache memories are acknowledged as one of the most complex resources to master, with cache coherence being a major contributing factor to cache's complexity.

Multicores implement cache coherence protocols to preserve data correctness. Most commercial-off-the-shelf (COTS) architectures use cache coherence protocols based on the MSI (Modified-Shared-Invalid) protocol [27]. Those protocols are usually adaptations to optimize performance that inherit the three fundamental states: Modified (M), Shared (S), and Invalid (I); for instance, MOESI from AMD Opteron or MESIF from Intel Xenon [15].

Several research works adopt an independent-task model in which critical and non-critical tasks are not allowed to use shared data by using strict cache partitioning [32] in shared caches, or locking mechanisms [30]. These solutions come with some limitations, being the most obvious one that they disallow sharing data between threads, which consequently disables all communications between threads of parallel tasks running on different cores.

Recent works recognize these limitations and propose solutions for data sharing. We categorize these solutions as follows:

Cache bypassing [6], [3] simplifies enormously the data-sharing problem since it removes coherence issues. These benefits come with the cost of a deteriorated performance in the average case.

Data-aware scheduling [5], [11], [12], [25] adds data-awareness to the task scheduler to avoid data interference. This is done by either scheduling sharing-data tasks to the same core, scheduling so sharing-data tasks do not run in parallel (i.e., in line with ARINC 653 [1]), or using performance monitoring counters to take data-wise scheduling decisions that mitigate the data sharing effects.

Cache coherence [14], [28], [17], [29], [13], [19], [18], [16] solutions advocate for using cache coherence protocols that handle each data-sharing operation's correctness, resulting in better average performance than bypassing the cache. The main issue with cache coherence is the notably high worst-case memory latency and the unpredictability caused by coherence interference. The works in this area propose protocols that improve predictability (required for critical real-time systems). The first of these works is [14], which takes MSI protocol as a baseline, finds all possible unpredictable scenarios, and proposes modifications to fix them, creating PMSI. The other works are based on PMSI (or later versions), and extend it to more complex protocols or propose significant improvements. Still, coherence protocols are hard to validate and must undergo expensive validation processes [21], and thus, are not yet adopted by COTS processors. Another work, PISCOT [16], has been proposed to use conventional coherence protocols; however, it still requires modifications to the interconnect to ensure predictability.

Real Boards. Recently works have studied the coherence support in real boards undergoing avionics certification [23]. A first work [26] analyzes the coherence between the different e6500 clusters of the NXP T4240 processor (each cluster has the same architecture as that in the NXP T2080) and concludes that it actually implements MESIF instead of MESI, as specified in the e6500 technical reference manual [9]. A second work [22] analyzes the accuracy of cache-coherence related event monitors in the T2080.

In this work, we propose adding monitoring counters to track coherence interference, improving the predictability without needing a modification in the cache coherence protocol nor the scheduling.

In line with the previous state of the art for coherence protocols in ECS, we focus on MSI, which includes the basics of most coherence protocols. In MSI, each cache line of a local cache can be in one of the three mentioned states. If a local cache holds a cache line in M state, it means that the cache owns the only valid copy of the cache line and, therefore, it can be read and modified without restrictions. When the cache line is in S state, it means that the local cache owns a valid copy of the data, but it is also found on the next level of cache and even might be in other same-level caches. While in this state, the cache line can be read but not modified. The last state is I, which means that the cache line is not valid, and therefore it can neither be read nor modified. To modify a cache line while in S or I state, the cache has first to send a $GetM$ coherence message to notify other elements having that same cache line, so they invalidate their copy (potentially copying it back if dirty). After that, the local cache receives a confirmation (if coming from S) or the cache line itself (if coming from I), and changes to M state where it can write on the cache line. Similarly, to read while in I state, the core has to send a $GetS$ coherence message to request a copy of the cache line; the current owner or next level cache will send the valid data, and upon receiving it, the state changes to S. While in M state, the cache line can be invalidated by either an eviction from the local cache or by an external $GetM$. In that case, the cache line is sent to the next level cache (if caused by an evict) or to the requestor (if caused by an external $GetM$). After that, the cache line is invalidated. Likewise, if the message received is an external $GetS$, the cache line is sent to the requestor, but after that, the state changes to S. Also, if a $GetM$ is received while in S state, the cache line is invalidated and changed to I state.

TABLE I: Coherence-related monitors in the T2080

Event Monitor	Description
L2SH	L2 snoop hits
L2SP	L2 cache snoop pushes
L2EX	L2 externally generated snoop requests
L2SM	L2 snoops causing MINT (Modified INTervention)
L2SS	L2 snoops causing SINT (Shared INTervention)
L2RE	L2 reloads from CoreNet
L2CN	L2 control requests to core (e.g., back invalidates)
L2DR	L2 data requests from L2 to core (data forwarding)

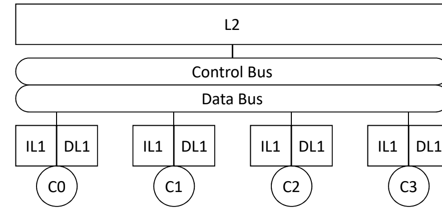


Fig. 1: Architecture model

III. Motivation

A. Monitoring Counters in Real Processors

In this section, without loss of generality, we focus on the T2080, given its traction in the avionics domain [23]. The NXP T2080 [10] comprises four e6500 cores [9], each with its private instruction and data cache, while the L2 – the main coherent point – is shared by the cores (see Figure 1).

The T2080 provides the event monitors related to coherence shown in Table I. All these monitors allow for deriving the coherence state transitions. However, they are not only aggregated monitors, rather than per-core, but also they fail to capture the time an application/thread is delayed due to coherence by other applications/threads. That is, after an application's execution, the user can only read the execution time of the application and the value of the coherence counters above – along with other 100+ counters. The value of the coherence counters is the result of aggregating the coherence activity by all cores. This information does not allow ascribing the coherence contention suffered by an application (or thread) to any of its co-runners.

B. Contention Prediction Models

Multicore-contention prediction models usually focus on the direct activity that the different contending cores generate each other [20], [7]. This includes the accesses generated by loads and stores and also the write-back activity the cache can generate. However, they do not take into account coherence-related contention.

The coherence-related activity affects threads of an application A_1 that share data. However, an independent application A_2 running in different cores to those used by A_1 and sharing no data with A_1 can also be affected by coherence among threads of A_1. We illustrate this with an example in which we focus on a 4-core multicore in which first-level data (DL1) and instruction (IL1) caches are private per core and the shared L2 cache from which each core receives a subset of the ways of the L2, see Figure 1. Section V provides further details about the evaluation framework.

We run a given single-thread task (application), referred to as analysis task or AT, in a given core and a two-threaded contender application (also referred to as contender task or CT) in two other cores. The CT always takes longer to execute than the AT, and the simulation ends when the AT finishes its execution.

We run the AT in isolation and in two multicore setups. In the first one, non-shared (NS), the threads of the CT do not share data. In the second one, shared (SH), the threads of the CT share data. In both cases, the CT threads execute exactly the same code, and the only difference is whether they access the same or different data.

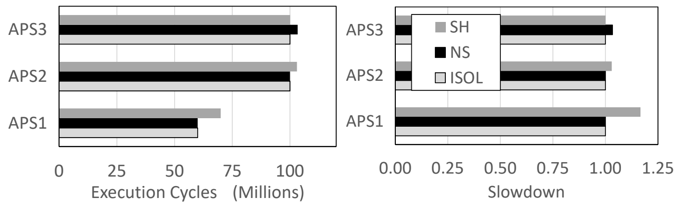
We create three application scenarios (APS1, APS2, and APS3) with different types of operations (loads and/or stores) and data sharing among the threads of the CT. Figures 2(a) and (b) respectively show the execution time of the AT and the slowdown it suffers w.r.t. its execution in isolation.

In APS1, the AT performs writes to a range of addresses that does not fit in its DL1, causing dirty evictions. Both threads of the CT also perform store accesses to a range fitting in their respective DL1 caches. For APS1, in Figure 2, we see a high slowdown when in the SH case caused by the increase in coherence messages that are exchanged between the two CT threads. With the messages from the CT using the buses constantly, the AT suffers contention when accessing the L2 because the interface (bus) connects all cores, and the L2 is busy sending data across both CT cores.

In APS2, the AT performs pairs of read and write operations to a range of addresses that does not fit in its DL1. In particular, the AT generates a load access followed by a store access to the same address, requesting the data to the L2. Also, since data is constantly being evicted from DL1 while dirty, each access has to write back the cache line evicted to the L2. Regarding the CT, the first thread performs load accesses and the second thread performs store operations. The data footprint of the CT threads fit in their DL1 caches. In APS2, we see a slowdown that is caused by reasons similars to those for APS1under the the SH scenario. The effect is less prominent than for APS1 because AT requests to the cores-to-L2 bus occur at a lower frequency, hence reducing both the absolute and relative impact of CT induced interference.

In APS3, the AT and the CTs behave as in APS2, with the only difference being that CT's footprint does not fit in DL1. In APS3, we see a counter-intuitive result where the NS version incurs a higher slowdown than the SH version. The main reason behind this behavior is that all CT accesses in APS3 result in a miss in the DL1, causing accesses to the L2, and on the SH version, these misses take longer to complete since the data might be on the other cores' DL1. This leads to CT accesses occurring at a lower frequency in the cores-to-L2 bus and hence, lower contention.

Overall, these simple examples show that the coherence-related activity of a given application affects other independent applications with which it shares no data. In all three APS



(a) AT's absolute execution time (b) AT's performance slowdown

Fig. 2: Motivation Examples

under the SH and NS setups, the CT threads perform the same number of accesses to DL1 and L2, so loads and stores hits/misses to DL1/L2 do not help in singling out the variability observed in Figure 2(a) and (b).

This calls for tracking coherence contention so that its impact can be properly tracked and attributed to the threads causing it.

IV. Proposal

Cache coherence contention can be categorized into two main components: *bus-access contention* and *protocol contention*. The *bus-access contention* might arise in the access to the control bus (cbus) and the data bus (dbus). The *protocol contention* captures coherence-processing, that is the delay suffered by a request to a cache line due to the fact that this cache line is shared with another core.

A differentiating feature of protocol contention compared to other contention sources is that protocol contention affecting a given core happens on the cache coherence messages it sends and the messages and data other cores send on its behalf. That is, a remote cache controller CCN_j can carry out some coherence activities to process an original request generated from a different CCN_i. To that end, CCN_j can generate messages that can suffer contention, which is to be ascribed to the original request generated by CCN_i and not to CCN_j. We refer to this as *indirect protocol contention*. Another source of indirect protocol contention arises when CCN_j answers to the CCN_i request, the response can create contention to other messages being sent from CCN_j, since it takes a slot that could be used by other messages. RPCT is designed to capture indirect protocol contention as shown in the following sections.

A. Tracking bus-access contention

Bus-access contention happens regardless of whether cores in the system share data or not. It occurs because these cores compete to access the shared DL1-L2 bus to issue memory requests, including control messages (control bus) and data (data bus). In order to prevent collisions and starvation, accesses to the bus are orchestrated using an arbiter. For example, under round-robin arbitration, each core is granted access to the bus in a fair manner. With round-robin contention delays, the worst-case bus access contention any core can suffer is the total latency to transmit one request from every other core in the system before it can issue its own request, having those contender requests the maximum request duration in the worst case. Hence, contention delay is bounded when there is a maximum duration to transmit a request [8].

B. Tracking protocol contention (RPCT)

We introduce RPCT in this section, which is illustrated via several examples in Section IV-C. A particular hardware implementation is presented in Section IV-D. RPCT assumes that dbus/cbus requests include the ID of the initiator core. Interestingly, interconnect specifications like the AMBA CHI used in the ARM CNM [2] capture that information in one of the mandatory request fields.

Under RPCT, the contention is tracked from the cores generating it instead of the cores suffering it. That is, the cache controller of the core owning the data (CCN_j) starts counting contention cycles suffered by a given core CCN_i when it receives a request ($GetM$ or $GetS$) from CCN_i. Cycles are counted until CCN_j sends the data to the L2 CCN (CCN_{L2}). Once the L2 gets the data, it starts a counter that tracks the contention between the contender (the core that sent the data, CCN_j) and the requestor (the core that sent the original request, CCN_i), and will finish counting when the data is sent to CCN_i. If there is more than one requestor, both the core that owns the data (CCN_j) and the L2 will keep track of the protocol contention suffered by all the requestors at the same time. In this way, the core and L2 count the protocol contention cycles for all the cores that are being delayed (e.g., CCN_i, CCN_k, etc.).

There might be multiple messages pending to be sent in a given core (CCN_j) to other cores (CCN_i, CCN_k, etc.), corresponding to different data. In that case, an additional source of contention is tracked being the contender, the core that created the request whose message is being served (e.g., CCN_i), and being the victim the core(s) whose message(s) is delayed (e.g., CCN_k). For instance, if CCN_0 owns two cache lines, and one is requested by CCN_1 and another by CCN_2, CCN_2's request will have to wait until CCN_1's request is solved. In this case, CCN_0 will count the indirect protocol contention caused by CCN_1 on CCN_2.

With RPCT, a core tracks not only the contention its requests suffer but also the coherence contention among any other pair of cores. Hence, it needs N^2 counters per core since each core can be tracking contention caused by any core on any other core. The total contention can be obtained simply by reading the counters in each core and aggregating them.

C. Illustrative Examples

In the figures used in this section, we draw an arrow from the core generating a request to the owner core of the requested data (i.e., holding the most recent version of the data for that request). This is just a representation to ease understanding since, in reality, messages are broadcasted to all cores and the L2 controller via the bus. In these examples, we assume no contention in the access to the data bus or the control bus for the sake of clarity. As discussed in Section IV-A, bus

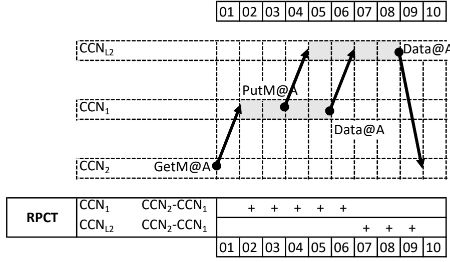


Fig. 3: One requesting data owned by another core.

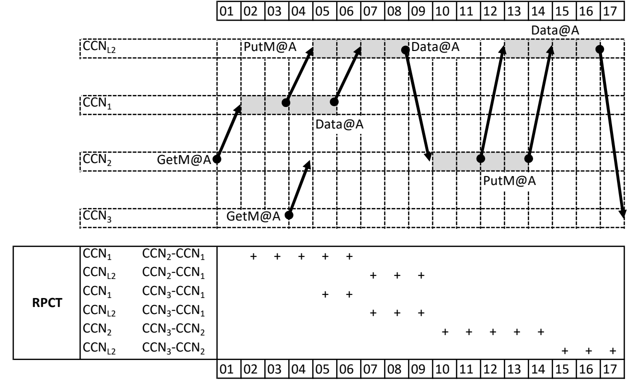


Fig. 4: Two requests to the same address from two cores.

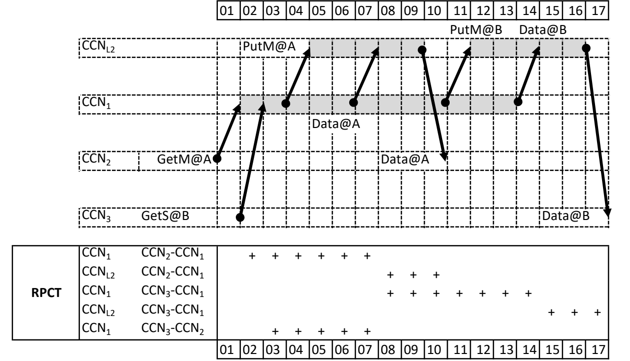


Fig. 5: Two requests to the different addresses.

arbitration contention simply adds to the protocol contention shown in this section.

1) Baseline case: Figure 3 shows a scenario in which CCN_2 requests some data owned by CCN_1. In cycle 1 CCN_2 sends $GetM$ on address $@A$. CCN_1 starts processing this request in cycle 2, sending a $PutM$ on $@A$ to the L2 controller at the end of cycle 3. We assume 2 cycles is the time it takes CCN_1 to process the request. CCN_1 follows in cycle 5 sending the data to the CCN_{L2}. The data arrives at the CCN_{L2} in cycle 7, which sends it to the CCN_1 in cycle 8 (we assume 2 cycles processing time in the CCN_{L2}).

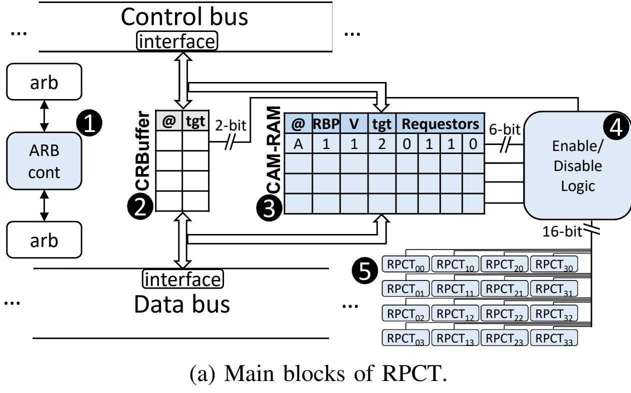
Under RPCT, CCN_1 starts counting contention on CCN_2 from the time it receives the request until it sends the data to the CCN_{L2}, from cycle 2 to cycle 6 (5). CCN_{L2} counts contention on CCN_2 from the time it receives the data until it sends it from cycle 7 to 9 (3). Hence, RPCT ascribes contention experienced by CCN_2 to CCN_1 (8 cycles), and is capable of splitting request delay between core contention and L2 protocol processing delay.

2) Two requests to the same address: Figure 4 shows a scenario in which CCN_2 requests a piece of data owned by CCN_1, and CCN_3 sends another request to the same address few cycles later. In cycle 1 CCN_2 sends $GetM$ on $@A$. CCN_1 starts processing this request in cycle 2, sending a $PutM$ on $@A$ to the L2 controller at the end of cycle 3. We assume 2 cycles is the time it takes CCN_1 to process the request, as before. CCN_1 follows in cycle 5 sending the data to the CCN_{L2}. In the meantime, CCN_3 sends a $GetM$ on the same address in cycle 3. The data arrives to the CCN_{L2} from CCN_1 in cycle 7 who sends it to the CCN_2 in cycle 9 (we assume 2 cycles processing time in the CCN_{L2}). CCN_2 continues sending the data to the CCN_{L2} with a $PutM$ message and the corresponding data in cycles 11 and 13, respectively. CCN_{L2} finally sends the data to CCN_3 in cycle 17.

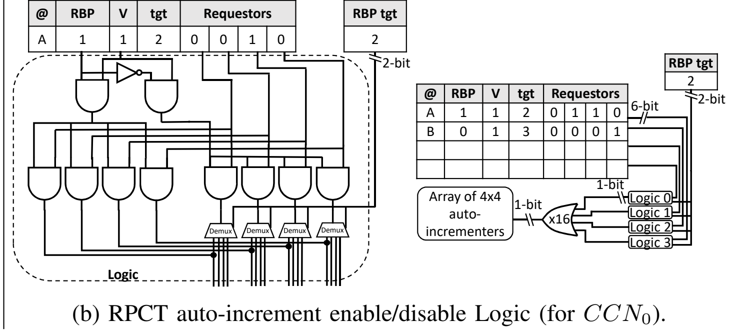
Under RPCT, in cycle 2 CCN_1 receives the $GetM$ for $@A$ from CCN_2 and starts counting contention until the data is sent at the start of cycle 6. In cycle 5 CCN_1 also receives the $GetM$ for $@A$ from CCN_3 so it also counts contention for CCN_3 until data is sent. CCN_3's $GetM$ is also received by CCN_2, which marks that the cache line has to be sent later to CCN_3. The L2 receives $@A$ data in cycle 6 and starts counting the contention caused by CCN_1 to both requestors (CCN_2 & CCN_3) waiting for the data $@A$ until it is sent to CCN_2. Upon CCN_2 receiving data in cycle 10, it starts counting contention to CCN_3, who had previously requested it, until CCN_2 sends the data $@A$ to the L2. Finally, L2 counts the processing time since it receives the data until it is sent to CCN_3 as contention caused by CCN_2. Overall, RPCT ascribes $5+3=8$ cycles (from cycle 2 to cycle 9) of CCN_2 contention to CCN_1 and $2+3=5$ (from cycle 5 to cycle 9) to CCN_3; and $5+3=8$ cycles (from cycle 10 to 17) of CCN_3 contention to CCN_2.

3) Two requests to different addresses: Figure 5 shows a scenario in which in cycle 1 CCN_2 sends a $GetM$ on $@A$ that is received by CCN_1 (who holds $@A$) and starts processing the request. In cycle 2 CCN_3 sends a $GetS$ on $@B$ that is snooped by CCN_1 who holds the address. In cycle 4, CCN_1 sends a $PutM$ for the write-back of $@A$, which happens on cycle 7. Then, L2 receives the data in cycle 8 and sends it to CCN_2 in cycle 10. In cycle 11, CCN_1 sends the $PutM$ to write-back $@B$, which occurs few cycles later in cycle 14. Finally, L2 sends $@B$ to CCN_3.

Under RPCT, CCN_1 starts counting delay to CCN_2 when it receives the $GetM$ for $@A$ in cycle 2. At the end of cycle 2, CCN_1 receives the $GetS$ from CCN_3. Since it is currently handling CCN_2's request, CCN_1 starts counting delay to CCN_3 on behalf of CCN_2. Once CCN_1 sends the $@A$ data to the L2, it stops counting CCN_2 contention on CCN_3 and starts counting CCN_1 contention on CCN_3. When the data $@A$ is written-back to L2, CCN_{L2} counts contention from



(a) Main blocks of RPCT.



(b) RPCT auto-increment enable/disable Logic (for CCN_0).

Fig. 6: RPCT block diagram.

CCN_1 (who sent the data) on CCN_2 (who requested the data) until the data is sent to the requestor (in cycle 10). This process happens again in cycle 14 when CCN_1 sends the data $@B$ to L2 and CCN_{L2} counts contention from CCN_1 to CCN_3 until the data is sent to CCN_3. CCN_1 stops counting contention to CCN_3 when the data is sent in cycle 14. Overall, with RPCT, CCN_1 is ascribed $6 + 3 = 9$ (from cycle 2 to cycle 10) contention cycles of CCN_2. CCN_3 contention is ascribed to CCN_1 ($7 + 3 = 10$ cycles, from cycle 8 to cycle 17) and CCN_2 (5 cycles, from cycle 3 to 7). CCN_3 contention caused by CCN_2 is indirect protocol contention.

D. Implementation

Figure 6a shows the main cache-coherent related elements of a cache controller. It comprises ① a simple logic to track arbitration contention; ② a coherent request FIFO buffer (CRBuffer) that is part of the cache coherence controller (CCN) and that keeps the coherence requests to be processed by the core; ③ a CAM-RAM to store requests whose contention is being tracked; ④ the control logic that enables/disables the update of ⑤ a set of auto-increment event monitors to track protocol contention among any pair of cores. Note that the arbiters in the cbus/dbus and the CRBuffer are part of the CCN, while RPCT adds the CAM-RAM, enable/disable logic, and RPCT counters (highlighted with light blue background). As described later, the additions to our proposal in terms of hardware cost are relatively minor. In particular, the CAM-RAM size and complexity are comparable to that of a Miss Status Holding Registers (MSHRs) in caches, but access only upon coherence requests, which occur far less often than cache misses and requires less than 100 bytes per core. The enable/disable logic has very low complexity, as highlighted in Figure 6b, and also switches occasionally. Finally, the RPCT counters require 64 bytes per core (in our 4-core multicore reference architecture) only and a small logic for the auto-incrementers (they are carry propagators, so far less complex than an adder). The counters are also active a small fraction of the time (e.g., 1 auto-incrementer active across all RPCTs of all cores every 5 cycles if coherence contention is 20%). Therefore, RPCTs incur less activity than the cycle counter of a core.

The arbiter contention logic ① tracks dbus/cbus contention. Basically, when the CCN has a request/data ready to be issued, all the cycles it is delayed because the bus grant is given to another core are accounted as contention from that other core to the local core.

RPCT builds on a CRBuffer ② in the cache controller that stores all coherence requests that are pending to be processed (answered) by the core. Each entry of the CRBuffer stores the ID of the core requesting the data, that in fact is the target core to which the response is going to be sent (tgt), and the address that it is requested. The requests in the CRBuffer are processed in order of arrival; hence the top entry is referred to as the request being processed (RBP). The interface logic snoops the cbus/dbus, and if an address hits an entry in M state or in a transient state where it is transitioning to M (for example, IMD), the request is inserted into the CRBuffer with the address, and the target core. At the same time, this inserts a new entry to the CAM. Whenever a request reaches the top of the CRBuffer, it becomes the RBP. Once the data for the RBP is sent to the L2, the entry is removed from the CRBuffer (and the CAM-RAM).

The request CAM-RAM ③ stores those requests for which this core is in charge of tracking coherence contention. The CAM-RAM contains the requests in the CRBuffer and other requests since, with RPCT, a core needs to track not only the requests having it as the target (i.e., it needs to track indirect protocol contention). The fields of the CAM-RAM are address (key), 1 bit to indicate whether it is RBP, 1 bit to indicate if the data is valid, a requestor N-bit vector indicating the other cores that have requested the data, and a $log_2(NumCores)$ field indicating the target core of the address in that entry of the CAM-RAM. In terms of operation, every time that a cache line in M state or in a state that will eventually be in M (IMD, ...) receives a request (GetM or GetS), which adds an entry to the CRBuffer, the system adds it to the CAM-RAM, puts the requestor as the Target and sets its requestor bit to 1. If another request is received for an address that is already found on the CAM-RAM, the system sets the corresponding

requestor value to 1. When the cache line data is sent to the L2, the corresponding line is deleted from the CAM-RAM.

The operation of the CAM-RAM in the L2 cache controller varies a bit as it needs an additional field, called source, indicating the ID of the core sending the cache line, which will be used as the aggressor in the case where a core is redirected as the local core ID.

Note that a small CAM-RAM (e.g., 8-16 entries) can satisfy all the coherence requests generated by load/store requests in general since, for instance, the Arm A53 core – commonly used in processors in ECS that are undergoing a certification process, like the Zynq UltraScale+ [34], [33] – allows a maximum of 3 loads in flight (store-related information is not revealed).

RPCT uses a set of event counters ⑤ that can either be incremented (by 1) or remain unaltered every cycle [1]. Conceptually, they are incremented while some condition holds (i.e., a specific core causes coherence contention to another core). Since such condition may hold for a number of consecutive cycles, the event counters are implemented as auto-incrementer counters where they are incremented by 1 every cycle as long as a specific control signal is high and remain unaltered if such signal is low. Hence, such control signal is an extra bit stored along with each counter. Overall, there is an array of 16 ($NumCores \times NumCores$) 32-bit counters (64 bytes in total for 4 cores), each one with its auto-increment control bit. From these 16 counters, up to 6 can be turned on in the same cycle for cores and up to 8 for the L2 cache due to the different casuistic.

RPCT comprises a logic ④ that, upon a change of the RBP or the entries of the CAM-RAM, updates the $NumCores \times NumCores$ enable signal for the auto-incrementers. As shown in Figure 6b, such logic ④ is relatively simple and mainly combines the new RBP and current requestors to determine the auto-incrementers to be enabled. Gate-level description of the logic is omitted due to space constraints. The outputs of the logic follow that if one request is valid and is the RBP, the local core (which in Figure 6b is represented for C0 [2]) is delaying all the requestors; if the request is valid but it is not RBP, the delay of all requestors is assigned to the current RBP target (which in Figure 6b is the RBP target is C2). Note that in the left part of the figure each dot represents an one-bit OR operation. Also, as shown in the right side of Figure 6b, this logic is replicated for each entry in the CAM-RAM, and each of the 16 outputs from each logic is ORed with their correspondents from the other entries to generate a total of 16 outputs that go to each of the 16 auto-incrementers.

If a snoop of the bus (GetS/GetM) results in a hit to the CAM-RAM, then a new requestor might be added, and therefore, it can start a new auto-incrementer. This modifies at most one entry of the CAM-RAM and, at most, one auto-incrementer.

[1]RPCT event counters are simpler than that regular event counter as the latter can be increment every cycle from 0 to N where N relates to the event tracked like the number of instructions committed in a cycle, flushed on a branch misprediction, ...

[2]From now on, we refer to cores (C_x) instead of cache controller in the core (CCN_x) for the sake of simplicity.

TABLE II: WWWW sharing data fitting in DL1 (10^6 cycles).

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	0.0011	43.20	41.39	24.00	0.9002	7.19	7.30	7.79
C1	24.00	0.0001	43.20	41.39	7.80	0.9000	7.20	7.30
C2	41.39	24.00	0.0001	43.20	7.29	7.80	0.8999	7.20
C3	43.19	41.39	24.00	0.0001	7.19	7.30	7.80	0.90

Whenever the RBP in the coherence logic is fully processed and removed from the CRBuffer, the next entry becomes the RBP. This causes that all auto-incrementers have to be reset again since changing the RBP implies changing the victims or aggressors in the cases described above. This modifies at most two entries of the CAM-RAM (one entry is removed, and another one is set as RBP), but has the potential to modify several auto-incrementers (up to 12 on the corresponding core and all 16 in the L2).

It is worth noting that the main building blocks of RPCT are not impacted when other cache protocols are used. What really changes is the control logic to determine which control messages to allocate and when to allocate and deallocate entries on the CAM-RAM. Hence, even when protocols are not fully predictable [14], RPCT can still be used to help discern how cores affect each other in terms of cache coherence.

V. RESULTS

We use the Gem5 [4] open-source modular platform widely used for computer-system architecture research, encompassing system-level architecture as well as processor micro-architecture. We focus on Ruby, which implements a detailed simulation model for the memory subsystem of Gem5. We model a system with 4 cores with local instruction and data caches. Both are connected to the shared L2 cache. The interconnection is split into two buses, the control, and the data bus, which buses use round-robin arbitration (see Figure 1). For the coherence, we implemented a cache coherence snooping MSI protocol in Ruby with no core-to-core transfers.

We use benchmarks from Splash-3 [24], which is a suite of parallel applications to facilitate the study of centralized and distributed shared-address-space multiprocessors. In this work, Section V-B analyzes RPCT counters on the Splash-3 benchmarks first in isolation and later against synthetic contenders.

We also use synthetic benchmarks, Section V-A, that have a sustained behavior during their execution and for which we have information about their usage of cache and data sharing. This allows us to have an informed guess of how cache coherence contention affects each other and hence, the actual cache coherence contention breakdown. In particular we use the *Write (**W**)* and the *Read (**R**)* benchmark generate one million load and stores, respectively; and the *Both (**B**)* benchmark that generates one million load-store pairs to the same position. Each of these benchmarks can be set to share data or use a disjoint set of data. The data size of the benchmark can be varied to force it to fit or not fit in L1.

TABLE III: BB (C2&C3) sharing data and fitting in L1 (10^6 cycles).

(a) CT: WW (C0&C1) not sharing data fitting in DL1.

	RPCT				Arb. Cont			
	Delayed Core							
	C0	C1	C2	C3	C0	C1	C2	C3
C0	0	0	0	0	0	0	0	0
C1	0	0	0	0	0	0	0	0
C2	0	0	24.99	53.98	0	0	4.99	13.99
C3	0	0	53.99	25.00	0	0	17.99	8.99

(b) CT: WW (C0&C1) sharing data fitting in DL1.

	RPCT				Arb. Cont.			
	Delayed Core							
	C0	C1	C2	C3	C0	C1	C2	C3
C0	0	96.31	0	0	3.66	8.99	6.33	11.33
C1	93.31	0	0	0	11.99	0.33	7.66	15.99
C2	0	0	30.99	60.32	6.33	9.99	7.66	9.99
C3	0	0	63.64	24.99	3.99	4.00	7.99	8.99

In the result tables in this section, we report the cycles of protocol contention each core in the column ascribes to the core in the row as produced by RPCT. We also provide the arbitration contention cycles. All values are reported in millions of cycles.

A. *Detailed Analysis of specific scenarios*

1) WWWW: All tasks write to the same memory address, which is shared among all cores. Since we model round-robin arbitration and all cores are aggressively trying to access the same cache line, the requests always get sent in the same order (as if we had TDM arbitration) [8]. In this experiment, when a core receives the data (e.g., C_1), one or two other cores will already be waiting to receive that same piece of data (e.g., C_2 and C_3), so the current core (C_1) carries out the store operation and sends the data to the L2 for the next requestor to take it. During this process, the core might receive another request from the core that owned the data previously (e.g., C_0). Hence, in terms of coherence contention, each core will be delaying mainly two cores (the ones that will always be waiting for the data once the core receives it) (e.g., C_2 and C_3), and then it will also delay the third core (e.g., C_0), which is the last one to get the data because it will send the request during the store process.

With RPCT, protocol contention is distributed, matching the real contention from these sequential accesses. For instance, in Table II, we can see that C_0 (column) is delayed by C_3 (row), but it assigns only around 40% of the contention ($40\% \approx 43.19/(43.19 + 41.39 + 24)$). C_2 also causes almost 40% of the delay, and C_1 causes the 20% remaining. Note that 40% matches quite well what we expected since C_0 will always request the data while C_1 is holding it and C_2 and C_3 are already waiting for it, i.e., C_0 will be delayed partially by the access from C_1 and completely by the accesses from C_2 and C_3.

Arbitration Contention is shown in the right sub-table. We see here that the requests from each core are mostly equally suffering contention from all cores (except themselves), which is expected in a homogeneous scenario like this one.

2) WWBB: In this case, the analyzed task (AT) is running in two cores (C_2 and C_3) and sharing data, while the contender task (CT) runs two threads in C_0 and C_1 respectively that share data among them but not with the AT. This scenario, in line with Section III, aims at illustrating the impact that coherence can have on each other's cache with different applications, even if they share no data.

In this experiment, one CT thread reads data fitting its L1, and the second thread writes data fitting its L1. If data is non-shared, no coherence traffic is generated by the CT. If data is shared, then coherence traffic is generated and is expected to impact AT due to bus Arbitration Contention. Both cores from the AT (C_2 and C_3) run a loop with read-write pairs sharing data and have some contention between them, meaning that they suffer contention from the other core when they send the $GetS$ request for the read. The other core is the current owner of the data, which causes the cache line to move to S state. Then the following write operation has to request the ownership again to the L2 to switch to M state.

With RPCT, the delay to switch to M state is counted as internal contention (since data is being accessed directly from L2), which explains why in Table III we see C_2 and C_3 suffering self contention. Table IIIb shows how, when the contenders are sharing data, contention between C_0 and C_1 changes from 0 cycles in Table IIIa to more than 90M cycles in both ways. This data sharing between CT makes AT delays increase between 10% and 20%.

The arbitration contention is captured by the bus arbitration counters (rightmost sub-table), but the intrinsic message transmission and data read activities are not included in RPCT counters. With this, we observe two effects when moving from non-shared to shared. First, the overall contention suffered by cores C_2 and C_3 increases (e.g., from 13.99+8.99 to 11.33+15.99+9.99+8.99 for C_3) due to the increased bus activity due to CT coherence messages and data exchange. Second, contention experienced by the AT is now caused by all cores rather than by AT ones only (e.g., C_3 is affected by C_2 and C_3 only in the non-shared CT case, and by all cores in the shared CT case).

B. *Splash-3 Isolation Results*

We start by analyzing the benefits of RPCT for performance analysis and optimization. To that end, we run each Splash-3 benchmark as a multi-threaded application using the four cores of our reference architecture (each core runs one thread). We show the insights about the application behavior provided by RPCT, which we correlate to the known behavior of Splash-3 applications. We group benchmarks that have very similar behavior and show results only for one representative of each group.

Intra: In this group we identify BARNES (Table IVa), FMM, and RADIX. RPCT shows that all 3 benchmarks suffer high intra-core protocol contention (i.e., from core C_i to core C_i, see the top-left bottom-right diagonal) while the inter-core

TABLE IV: Protocol and aribtration contention for Splash-3 Benchmarks (10^6 cycles).

(a) BARNES

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	241.13	4.93	4.84	6.57	20.59	15.72	15.19	14.54
C1	4.56	227.30	4.50	3.49	15.53	17.96	15.06	14.42
C2	5.42	5.38	226.19	4.16	15.10	15.08	18.43	14.59
C3	6.53	3.36	4.20	217.03	14.45	14.48	14.55	16.83

(b) OCEAN-NC (base)

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	510.32	4.55	4.84	2.77	19.30	50.91	45.42	53.32
C1	3.05	508.77	1.06	2.71	53.01	19.06	49.60	43.68
C2	0.9551	4.40	507.39	3.09	45.81	50.07	19.77	51.27
C3	4.87	3.22	3.75	508.70	48.54	46.19	52.45	17.97

(c) OCEAN-C (Optimized)

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	164.57	24.96	3.06	21.14	12.81	15.05	16.18	15.84
C1	9.68	166.45	1.98	4.95	15.30	12.93	16.28	16.06
C2	1.54	5.93	164.85	6.79	15.88	16.20	12.23	15.20
C3	4.67	2.16	9.52	164.64	15.98	16.04	14.97	12.65

(d) RAYTRACE

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	836.21	4.66	0.81	0.78	167.35	6.27	6.48	6.64
C1	0.89	97.49	0.95	0.78	6.21	6.82	6.28	6.35
C2	0.83	3.01	100.75	0.80	6.42	6.29	6.91	6.59
C3	0.84	0.82	0.77	103.42	6.57	6.35	6.60	7.28

(e) WATER-NSQUARED (base)

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	37.55	0.03	0.02	0.01	6.70	1.02	1.01	1.02
C1	0.03	25.93	0.05	0.05	0.98	3.45	1.07	1.17
C2	0.10	0.11	25.72	0.00	0.97	1.09	3.43	1.10
C3	0.01	0.23	0.05	26.05	0.98	1.19	1.09	3.49

(f) WATER-SPATIAL (optimized)

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	67.96	2.71	0.03	0.01	8.61	2.41	2.41	2.48
C1	0.03	59.06	0.01	0.01	2.38	6.57	2.73	2.35
C2	0.05	0.04	58.46	0.00	2.38	2.75	6.40	2.43
C3	0.03	0.05	0.02	59.54	2.43	2.32	2.40	6.97

protocol contention from C_i to C_j is limited and evenly distributed. For BARNES, RPCT shows how only a 6% $((total - diagonal)/total = 57/968 \approx 6\%)$ of the coherence contention to each core is coming from external cores, i.e., around that 6% of the accesses to memory are answered by other cores instead of coming directly from the next level of cache. This occurs because BARNES divides a volume into volumetric cells via octree, being cells treated individually (in isolation in one core), with only the uppermost cell levels requiring sharing data across cores. On the arbitration side, we see that mostly all cores delay equally to each other (between 20.59 and 14.45) because memory accesses from cores are highly homogeneous in number and distribution over time across cores. Overall, RPCT provides key information to conclude that protocol contention is low and homogeneous, so there is little margin for improvement.

C0: This group covers RAYTRACE (Table IVd), CHOLESKY, FFT, and VOLREND. For these benchmarks, RPCT shows C_0 suffers high protocol contention (see C_0-C_0 cells) due to high initialization costs and/or serialized parts where other cores stall. C_0-C_0 protocol contention is 3x to 20x higher than for any other core (e.g., 8x for RAYTRACE). Similar trends are observed in arbitration contention. Hence, RPCT also allows us to conclude that the other cores almost do not share data or share only clean (read-only) data, meaning that most of the contention is intra-core, and we see almost no inter-core contention. Overall, RPCT provides valuable information that indicates that C_0 is the bottleneck, not because of coherence but because of poor load balancing across cores. A careful analysis of the application behavior confirms our findings. RAYTRACE generates rays from the viewport that bounce on the objects in the scene. Each pixel in the viewport can be parallelized since the only shared data is the scene, which is used as read-only. Hence, data is mostly shared in S (Shared) state, and therefore not causing high delays. Also, C0 has a way higher intra-core contention, but this is caused by the initialization of the data, which does not fit in the cache.

Optimized and Non-optimized groups: We find two benchmarks with optimized and non-optimized (base) versions respectively: OCEAN-C (Table IVc) and OCEAN-NC (Table IVb) on the one hand, and WATER-SPATIAL (Table IVf) and WATER-NSQUARED (Table IVe) on the other. For OCEAN-NC, RPCT shows that only intra-core protocol contention is high, whereas inter-core one is very low and highly homogeneous, as for the **Intra** category. However, upon optimization (see OCEAN-C), RPCT shows that intra-core protocol contention roughly drops to 1/3 of the original one, but the inter-core one increases noticeably. For instance, C_1 inter-core protocol contention grows by 3x in absolute terms and moves from being 2% to becoming 16% of the protocol contention. Moreover, such contention is highly heterogeneous across cores (e.g., C_0 contention on C_1 and C_3 is particularly high). Overall, RPCT provides accurate diagnostics that relates contention to how the OCEAN-C benchmark data grids are split and shared across cores and enable further optimizations.

Regarding WATER-NSQUARED (Table IVe) and WATER-SPATIAL (Table IVf), RPCT shows that there is almost no shared data between cores in both cases, causing a very low inter-core contention. In fact, the algorithm calculates the interactions between water molecules; since there is no communication between the intramolecular computations, except in the small number of accumulations to a global sum

TABLE V: Splash-3 Benchmarks against contenders (10^6 cycles).

(a) CHOLESKY-ISOL2

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	308.56	0.03	0	0	79.62	20.83	0	0
C1	0.05	223.45	0	0	20.72	52.09	0	0
C2	0	0	0	0	0	0	0	0
C3	0	0	0	0	0	0	0	0

(b) CHOLESKY-CT

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	333.33	0.02	0.00	0.00	49.51	6.84	21.99	22.01
C1	0.06	225.73	0.00	0	6.86	13.20	21.07	21.06
C2	0	0	0.01	1449.81	34.17	31.87	25.81	216.19
C3	0	0	1449.85	0.00	34.47	32.05	216.18	25.64

(c) LU-NCB-ISOL2

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	555.24	0.01	0	0	94.98	82.91	0	0
C1	0.01	576.05	0	0	82.81	101.17	0	0
C2	0	0	0	0	0	0	0	0
C3	0	0	0	0	0	0	0	0

(d) LU-NCB-CT

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	636.97	0.007	0.00	0.00	31.92	47.89	57.66	58.23
C1	0.01	602.90	0.00	0	48.25	28.88	53.90	54.81
C2	0	0	0.007	1114.62	83.60	80.22	12.07	119.22
C3	0	0	1114.01	0.00	85.96	81.71	120.15	10.55

(e) LU-CB-ISOL2

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	76.03	0.0252	0	0	22.54	3.25	0	0
C1	0.0372	72.12	0	0	3.27	21.36	0	0
C2	0	0	0	0	0	0	0	0
C3	0	0	0	0	0	0	0	0

(f) LU-CB-CT

	RPCT				Arb. Contention			
	Delayed Core							
	C0	**C1**	**C2**	**C3**	**C0**	**C1**	**C2**	**C3**
C0	80.78	0.02	0.00	0.00	5.50	1.14	7.58	7.60
C1	0.04	78.82	0.00	0	1.15	4.75	7.71	7.71
C2	0	0	0.01	591.49	12.23	12.51	9.46	89.99
C3	0	0	591.47	0.00	12.42	12.49	90.00	9.41

every time step, the problem size reduces the communications. Although the WATER-SPATIAL is the optimized version, it suffers approximately 2x intra-core contention w.r.t. WATER-NSQUARED. The cause is that in WATER-SPATIAL, the data is stored in a different and more complex data structure, which reduces the computations needed in large cases. Still, in small cases (such as the one in this execution), the data structure has an overhead resulting in approximately double the contention (both in RPCT intra-core and arbitration) caused by a poorer data locality. This effect, while not increasing the data accesses, decreases the L1 hits, consequently increasing the L2 accesses. It is worth noticing that RPCT allows us to see that, in the WATER-SPATIAL, differently to WATER-NSQUARED, C_0 generates non-negligible contention on C_1 (around 4.5% of its total contention suffered).

C. Splash-3 vs. Contenders Results

In this experiment, we run a 2-threaded Splash-3 benchmark using two cores alongside a 2-threaded contender application in the other two cores with heavy data sharing among its two threads. We compare the Splash-3 benchmark execution in isolation using two cores (ISOL2) and then with the contender application running on the other two cores (CT). The purpose of this experiment is to show how RPCT captures contention even among co-running applications that do not share data among each other.

For CHOLESKY (Tables Va and Vb), we see an intra-core contention increase. For instance, in the case of C_0 it increases from 308.56 to 333.33 (8% increase). This reflects the increased latency to process coherence requests in the L2 cache since, despite different applications do not delay each other's requests explicitly, they do it implicitly due to resource hazards other than bus arbitration contention. Arbitration contention globally increases due to the large number of coherence requests of the contenders (i.e., accumulated contention grows from around 100 to 125). However, intra-core arbitration contention across threads of the application under analysis decreases. The cause is that the arbitration assigns slots to all cores, and if a core is attempting to send a message during a given slot, it will wait for the arbitration for the next slot, and this delay will be attributed to the last core that used the bus. Since now contenders are constantly using the bus, it is more likely that the contenders will use the bus between accesses from the benchmarks, and hence, they are the ones delaying benchmarks' messages. Apart from this, the arbitration contention complements the data from the RPCT results showing where the increase in time comes from. For example, C_0 experiences around 25M (333.39 - 308.61) extra cycles of delay when run against contenders, and looking into the arbitration contention, we see a similar increase on the C0 column from 100 to 125. If we further analyze the overall execution times of the benchmarks, we see that it increases by almost 31M cycles, which means that RPCT allows explaining where 80% of that contention comes from. Note that without RPCT, end users would lack the means to determine whether contention is dominated by coherence contention or any other source.

Finally, for both LU versions (Tables Vc, Vd, Ve and Vf) we see again that RPCT values increase when the contenders are running. Similar to CHOLESKY, we see that the arbitration contention gets distributed, but the summation of each column

increases with the contenders, and the difference between these values is very similar to the difference we see in the RPCT counters, meaning that the arbitration contention justifies the cycles increase in RPCT. In this particular case, it is interesting that the counters allow us to see that the non-optimized version of LU (LU-NCB) is more prone to suffer arbitration contention than the optimized version (LU-CB) since the relative contention increase between isolation and against contenders is higher for the non-optimized version. This is so because, for the non-optimized version, data blocks are accessed in a non-contiguous manner, hence with higher miss rates and additional cache coherence requests.

D. Benefits on Testing and Validation

Several techniques have been proposed to estimate the worst-conten-tion delays in the accesses to hardware shared resources, which are used as building blocks for multicore WCET estimation. Tests can be built to add high load on a resource (e.g. the bus), checking with the proposed performance monitoring counters whether the contention delay observed for any request goes beyond the estimate made. The absence of this scenario, together with an explanation of the experiment carried out to cause high load on the bus, serves as additional evidence on the correctness of the estimation to the worst-case contention delay. The proposed performance monitoring counters can also track the longest contention delay a request from a given core can cause on others, which is fundamental for the validation of WCET estimates.

VI. Conclusions

Embracing parallel applications in embedded critical systems requires providing some light on how cache coherence protocols affect co-running applications (co-runners). Building on the end-to-end execution time of co-runners and the use of event counters related to coherence activity, we can track the source of coherence contention. In this line, and unlike previous works that either advocate for parallelism-limiting solutions or propose new coherence protocols, we have shown that cache-coherence related contention spans beyond the threads actually sharing data, and we have proposed a new cache-coherence specific performance monitoring counter infrastructure. We have shown how with low overhead our proposal can track cache coherence, ascribing how threads, either from the same or different applications, affect each other. This helps to optimize parallel applications and is required for multithreaded application timing validation and verification.

VII. Acknowledgments

The research leading to these results has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773). This work has also been partially supported by Grant PID2019-107255GB-C21 funded by MCIN/AEI/ 10.13039/501100011033.

References

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

[2] Arm Ltd. *Arm CoreLink CMN-600 Coherent Mesh Network – Technical Reference Manual*, 2018.

[3] Ayoosh Bansal et al. Cache where you want! reconciling predictability and coherent caching. *CoRR*, abs/1909.05349, 2019.

[4] Nathan L. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011.

[5] John M. Calandrino and James H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Dublin, Ireland, July 1-3, 2009*, pages 194–204. IEEE Computer Society, 2009.

[6] Micaiah Chisholm et al. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 - December 2, 2016*, pages 57–68. IEEE Computer Society, 2016.

[7] Enrique Díaz et al. Mc2: Multicore and cache analysis via deterministic and probabilistic jitter bounding. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 102–118, 2017.

[8] Gabriel Fernandez et al. Computing safe contention bounds for multicore resources with round-robin and FIFO arbitration. *IEEE Trans. Computers*, 66(4):586–600, 2017.

[9] freescale semicondutor. e6500 Core Reference Manual. https://www.nxp.com/docs/en/reference-manual/E6500RM.pdf, 2014. E6500RM. Rev 0. 06/2014.

[10] Freescale semicondutor. QorIQ T2080 Reference Manual, 2016. Also supports T2081. Doc. No.: T2080RM. Rev. 3, 11/2016.

[11] Giovani Gracioli and Antônio Augusto Fröhlich. On the design and evaluation of a real-time operating system for cache-coherent multicore architectures. *ACM SIGOPS Oper. Syst. Rev.*, 49(2):2–16, 2015.

[12] Giovani Gracioli and Antônio Augusto Fröhlich. Two-phase colour-aware multicore real-time scheduler. *IET Comput. Digit. Tech.*, 11(4):133–139, 2017.

[13] Mohamed Hassan. Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems. In *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPIcs*, pages 16:1–16:24, 2020.

[14] Mohamed Hassan et al. Predictable cache coherence for multi-core real-time systems. In Gabriel Parmer, editor, *2017 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2017, Pittsburg, PA, USA, April 18-21, 2017*, pages 235–246. IEEE Computer Society, 2017.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[16] Salah Hessien and Mohamed Hassan. The best of all worlds: Improving predictability at the performance of conventional coherence with no protocol modifications. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 218–230. IEEE, 2020.

[17] Anirudh M. Kaushik et al. CARP: A data communication mechanism for multi-core mixed-criticality systems. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 419–432. IEEE, 2019.

[18] Anirudh M. Kaushik et al. Designing predictable cache coherence protocols for multi-core real-time systems. *IEEE Transactions on Computers*, 2020.

[19] Anirudh M. Kaushik and Hiren D. Patel. A systematic approach to achieving tight worst-case latency and high-performance under predictable cache coherence. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 105–117. IEEE, 2021.

[20] Hyoseung Kim et al. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014.

[21] Yangdi Lyu et al. Directed test generation for validation of cache coherence protocols. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(1):163–176, 2019.

[22] Roger Pujol et al. Empirical evidence for mpsocs in critical systems: The case of nxp's T2080 cache coherence. In *DATE*, 2021.

[23] David Radack et al. Civil Certification of Multi-core Processing Systems in Commercial Avionics, 2018.

[24] Christos Sakalis et al. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.

[25] Gero Schwäricke et al. Fixed-priority memory-centric scheduler for cots-based multiprocessors. In *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPIcs*, pages 1:1–1:24, 2020.

[26] Nathanaël Sensfelder et al. On how to identify cache coherence: Case of the NXP qoriq T4240. In *ECRTS*, 2020.

[27] Daniel J. Sorin et al. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[28] Nivedita Sritharan et al. Hourglass: Predictable time-based cache coherence protocol for dual-critical multi-core systems. *CoRR*, abs/1706.07568, 2017.

[29] Nivedita Sritharan et al. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 433–445. IEEE, 2019.

[30] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In Limor Fix, editor, *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 300–303. ACM, 2008.

[31] Theo Ungerer et al. Parallelizing industrial hard real-time applications for the parmerasa multicore. *ACM Trans. Embed. Comput. Syst.*, 15(3), may 2016.

[32] Bryan C. Ward et al. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 157–167. IEEE Computer Society, 2013.

[33] XILINX. Rockwell Collins Uses Zynq UltraScale+ RFSoC Devices in Revolutionizing How Arrays are Produced and Fielded: Powered by Xilinx, 2018.

[34] XILINX. *Zynq UltraScale+ Device. Technical Reference Manual. UG1085 (v2.1)*, 2019.