Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds

Reza Mirosanlou
University of Waterloo, Canada

Mohamed Hassan
McMaster University, Hamilton, Canada

Rodolfo Pellizzoni
University of Waterloo, Canada

Abstract

In Commercial-Off-The-Shelf (COTS) systems-on-chip, processing elements communicate data through a shared memory hierarchy, and a coherent high-performance interconnect, where the de facto standard to handle shared data is through a coherence protocol. Driven by the extraordinary demands from modern real-time embedded system applications to generate, process, and communicate massive amounts of data, recent efforts aim to ensure timing predictability while integrating cache coherence in multi-core real-time systems. However, we observe that most of these efforts compromise system average performance upon offering predictability guarantees. Motivated by this observation, this work proposes an arbiter aimed at providing a predictable, coherent shared cache hierarchy solution, yet with a negligible performance degradation compared to COTS solutions. We achieve this goal by adopting a high-performance-driven architecture including a split-transaction bus and bankized shared cache. In addition, all accesses are arbitrated through a global ordering mechanism. Our proposed arbiter operates alongside conventional coherence protocols without requiring any protocol modifications. Furthermore, we leverage the Duetto reference model by pairing the proposed arbiter and a high-performance arbiter. We evaluate our solution based on both synthetic and SPLASH-3 benchmarks, showing that we can significantly outperform the state-of-the-art in predictable cache coherence, while offering a COTS-level performance.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture; Computer systems organization → Embedded hardware

Keywords and phrases Predictability, Cache, COTS, Arbitration, Real-time system

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.16

Acknowledgements We would like to thank the anonymous reviewers for their valuable feedback, and our shepherd for helping to significantly improve this paper. This work has been supported in part by NSERC, CMC Microsystems, and TII. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

1 Introduction

Enabling data sharing is imperative in modern embedded systems—automotive, Unmanned Air Vehicles (UAVs), and Internet-of-Things (IoT) to name a few. In these systems, massive amounts of data have to be collected (sensor fusion, cameras, etc), communicated through interconnect(s), and processed by various processing elements. As a result, recent efforts have been proposed to shift away from the independent task model, where tasks do not share data to a more-practical model that embraces data sharing and enables inter-core communication [5, 4, 21, 10, 33, 6, 34, 17]. Among these solutions, we find those leveraging cache-coherent interconnects to be promising due to their performance benefits as well as transparency to the software stack. In addition, cache coherence is already the standard de facto in Commercial-Off-The-Shelf (COTS) multi-core platforms.

However, real-time embedded systems impose their own unique challenges, which do not exist in these performance-oriented COTS platforms. Predictability comes at the top of the list of these challenges; the architecture has to be predictable by-design to facilitate the timing analysis, which is necessary to provide timing guarantees to tasks running on the system. Originally designed with performance as the main goal, COTS cache coherent interconnects deploy several re-orderings and optimizations that hinder their predictability. It has been shown that even deploying a simple COTS coherence protocol such as the Modified-Shared-Invalid (MSI) protocol on top of a Time Division Multiplexing (TDM)-based interconnect revokes the system predictability [10, 16].

1.1 Related Work: Predictable Cache Coherence

To address this problem, recently the community has proposed several works that aim at implementing predictable cache coherence solutions. However, most existing solutions impose both coherence protocol as well as architectural modifications [10, 37, 18, 16, 17] or at the very least require specific hardware support [9]. These changes have led in early works to a quadratic increase in the worst-case memory latency (WCL) [10, 37, 18] (***Problem*** ❶). Moreover, mandating coherence protocol modifications discourages a real adoption of these solutions from the industry since adoption and verification of a new coherence protocol is known to be one of the most complex architectural tasks [36, 30] *(**Problem** ❷)*. PISCOT [15] addresses these problems by deploying of COTS coherence protocols on split-transaction interconnects. It also reduces the quadratic worst-case coherence latency to be linear in the number of cores. Nonetheless, PISCOT achieves this tight WCL by deploying two techniques that limit the overall memory performance compared to COTS solutions. The first is a TDM-based request bus, which is needed to enforce predictability, and the second is limiting the number of requests that each core can issue to the interconnect to one, which is needed to achieve the aforementioned tight latency *(**Problem** ❸)*. Additionally, similar to all existing work, PISCOT models the data bus and the last-level cache (LLC) as a single shared resource, and hence, no parallelism is possible in accessing the LLC *(**Problem** ❹)*. In COTS platforms, since bank processing times are much longer than the data transfer on the bus, LLC is usually a bankized memory, where different banks can process requests in parallel to improve system's performance [2].

1.2 Contributions

Motivated by these limitations, this paper makes the following contributions: 1) We propose DUEPCO: a novel real-time arbitration scheme for managing memory accesses in the cache hierarchy. This arbiter models the cache hierarchy as independent and parallel resources: the request (control) bus, the response (data) bus, while each LLC's bank is a resource of its own. This is key to leverage parallelism among these components to improve average performance, while tightening memory latency bounds (addressing ❶ and ❹*)*. More details about this arbiter are in Section 4. 2) Our proposed arbiter operates alongside conventional coherence protocols without requiring any protocol modifications (addressing ❷*)*. 3) In Section 5, we provide a timing analysis that ensures predictability by statically bounding the worst-case latency suffered by any memory request. Unlike the solutions in [10, 16, 18, 37], and similar to [15, 9, 17], this bound is linear in the number of cores (addressing ❶). 4) To further address the performance-predictability trade-off, in Section 6 we show how to extend the Duetto reference model [25, 26] to our cache architecture. This is achieved by integrating two arbiters: a High-performance Arbiter (HPA) offers the system a COTS-level performance

most of the time, while the proposed Real-time Arbiter (RTA) runs in parallel and is only utilized when necessary to meet timing guarantees (addressing ❸). 5) Finally, in Section 7 we evaluate the proposed arbiter against the state-of-the-art predictable coherency solution as well as a baseline COTS solution using both synthetic and SPLASH-3 [31] benchmarks. Our evaluation shows that our arbiter outperforms state-of-the-art predictable solutions in terms of average memory latency by an average of 1.74×, while providing comparable worst-case latency bounds to the best predictable mechanism. Employing Duetto can further improve system throughput by up to 6.4× at the cost of some degradation in latency bounds.

2 Background

2.1 Hardware Cache Coherence

Cache coherence is ubiquitous in shared memory multiprocessors since it enables shared data communication between cores while ensuring the system maintains data correctness. Data correctness is achieved when all cores consume (upon request by load/store instructions) the most recent copy of data. Coherence protocols employed in current Commercial-off-the-Shelf (COTS) systems are extensions of the basic Modified-Shared-Invalid (MSI) protocol [36]. MSI consists of three fundamental stable states: 1) **Modified (M)**: corresponds to memory blocks that have been modified (dirty) by a write in a private/shared cache; hence, the core/LLC is the *owner* of this cache line; 2) **Shared (S)**: corresponds to the blocks that are unmodified (clean) and held by one or more cores; 3) **Invalid (I)**: contains potentially stale and incoherent datum and both loads and stores will miss when accessing invalid blocks. Note that the protocol allows multiple cores to have a cache line in the **S** state, while only one core could have a cache line in the **M** state. According to the observed memory activity, the state of the cache line copies will be changed in the cache controllers. There exist two types of hardware cache coherence mechanisms based on how cores and the shared memory perceive the memory activity: 1) snooping bus-based cache coherence [36] in which all cores broadcast all the memory activities on the bus; 2) directory-based cache coherence [38] in which every activity will be communicated through a centralized directory that tracks the information regarding the cache lines among all cores. In this work, we employ snooping bus-based coherence as they are normally deployed in multi-core systems with up to 16 cores [32, 3, 27] and also deployed in state-of-the-art efforts [15, 10, 17, 18].

2.2 Arbitration

Simultaneous access to physical shared resources such as shared bus have a significant effect on the execution time of the applications. Therefore, having an arbiter is necessary when multiple cores try to access the shared resource simultaneously. Similar to the other shared resources in the system, different arbitration schemes lead to different timing characterizations of the system [7, 28, 11, 40, 17, 15, 10, 8, 7, 24, 29, 23]. The memory bus in multicore systems is one of the primary sources of interference. Therefore, a predictable arbiter guarantees that each requesting core is granted the bus eventually in a defined upper-bound amount of time. In COTS platforms, a high-performance arbiter is commonly used to maximize the overall performance. First-Come-First-Serve (FCFS) [19] is one example in this context that does not provide any latency guarantee for the shared memory accesses (assuming that there is no bound on reordering requests) and favors the cores that generate more requests to the shared resource and operate faster than the other cores in the system. On the other hand,

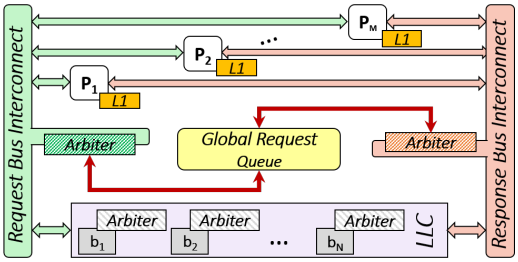


Figure 1 Architecture model.

this could potentially lead to a case where a request from a slower core takes a very long time to get service. Another approach is to assign a fixed priority to a certain processor but this type of arbitration policy cannot provide a guarantee to lower priority requests.

The level of complexity of the coherence protocols heavily relies on the underlying deployed interconnect network architecture. For instance, atomic/unified buses significantly simplify the protocol implementation; however, the performance of the system will be significantly degraded as all the cores are required to wait until the granted core finishes its interconnect usage; hence, serializing the accesses. In COTS platforms such as ARM Corelink CCI550 and Intel's QPI, the shared bus is usually implemented as a split-transaction interconnect in order to improve system performance. This is achieved by concurrently managing both coherent messages and data responses [35] such that the request bus and response bus will be separated. This architecture allows pipelining operation for the requests and responses on the bus and increases the flexibility in terms of the arbitration. In detail, a split-transaction bus can provide responses in an order different from the request bus depending on the arbitration on request and response buses.

3 System Model

In this section, we first detail the hardware architecture considered in this paper along with the coherency assumptions. Then, we explain how requests generated by the cores are processed by the proposed hardware architecture and how the latency for each request is constructed.

3.1 Architecture and Coherency

An overview of the proposed hardware architectural model is delineated in Figure 1. We consider a multi-core system with M Out-Of-Order (OOO) requestors[1] including processing cores, $P_1, ..., P_i, ..., P_M$ where each requestor has exclusive access to a private cache. All cores have also access to a shared memory that we assume is an on-chip Last-Level Cache (LLC). We assume that tasks running on the cores can share data among each other; hence, a coherency protocol must be employed in the system to allow coherent actions among cores and LLC. We consider both data transfers between a private cache and the LLC, as well as direct Cache-to-Cache (C2C) transfers between private caches, which exhibit improved average-case performance. In this paper, we adopt the MSI coherency protocol that includes three fundamental stable states as discussed in Section 2. Notice that we use the MSI as an

[1] We use cores and requestors interchangeably throughout the paper.

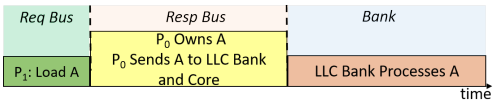
exemplar protocol; however, the proposed solution can work in tandem with other protocols. This is because we do not modify the coherency protocol by any means and all proposed elements of the design are independent of the details of the implemented protocol. This also simplifies the verification efforts compared to the approaches that alter protocols.

Instead of a unified interconnect commonly deployed in real-time architectures, we consider a split-transaction bus in which all communications between cores and LLC is done using two separate buses: 1) *request bus*, which is responsible for broadcasting the coherency messages; 2) *response bus*, which is a dedicated interconnect to transfer the data responses from/to cores. These two buses operate in parallel to improve the performance of the system. The *request bus* and *response bus* take a certain amount of time to transfer message packet and data response, which we represent with t_{REQ} and t_{RESP}, respectively. In order to maximize the parallelism in the system, we propose to bankize the LLC such that multiple requests can be processed simultaneously. Bankizing LLC is a common approach in COTS platforms to increase system's performance such as in Intel's architecture [2]. Therefore, we assume that the LLC consists of N independent banks, $b_1, ..., b_i, ..., b_N$ where each bank consumes a certain amount of time t_{BANK} to process writing data to (or retrieving data from) the cache data array inside each bank. In addition, LLC banks could be shared among all cores [12] or partially shared similar to [22]. In our model, we assume the former. Similar to existing related works [40, 17, 15, 10, 8], in this paper, we only focus on the interference suffered by the L1-LLC traffic due to coherence, shared cache(s), and shared interconnects and do not model the extra interference occurring in off-chip memory due to LLC misses. This latter can be bounded using other existing orthogonal approaches such as [39, 12, 13, 7].

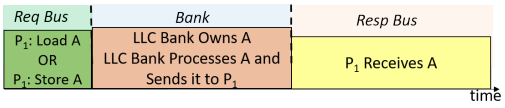
We assume that each cache entity (private and LLC) has its own set of interconnect buffers: `TxMsg`, `RxMsg`, `TxResp`, and `RxResp` to register the incoming/outgoing messages and data responses. `RxMsg` contains the incoming message packets from the request bus. We assume that every message will be decoded immediately in the private cache and each LLC bank even if the bank is busy writing/retrieving data from its data array. `TxMsg` contains the outgoing message packets from any core/bank that must snoop on the request bus. For instance, if a core asks to modify a cache line that it is not in possession of, the core must inform other cores by a coherency message (GetM) and push it in its own `TxMsg` buffer to be propagated on the request bus. This allows other cores/LLC to be aware of this action. `RxResp` contains the data responses coming from the response bus. `RxResp` at each core includes data response that the bank provides or data response due to a cache-to-cache transfer. `RxResp` at LLC bank includes the data response supplied by the cores in case of write-back. Notice that unlike `RxResp` buffers of each core, the data responses placed in LLC `RxResp` must be processed in the bank which takes t_{BANK} to process. Finally, `TxResp` includes the responses that need to be transferred on the response bus. The request bus, response bus, and LLC banks act as independent shared resources which conduct their own independent arbitration policies. In detail, the request bus arbiter is responsible to arbitrate the messages residing in `TxMsg` and the data responses inside `TxResp` buffers are arbitrated through the response bus arbiter. Similarly, each arbiter at LLC bank arbitrates the message/responses in `RxMsg` and `RxResp` buffers.

3.2 Request Processing and Order of Arbitration

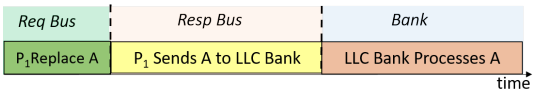
From the perspective of the coherency architecture, a requestor issues *requests* to the system based on the following activities in L1 cache: 1) load miss requests; 2) store miss requests, including stores to a cache line in **S** state; 3) replacement requests due to a write-back to shared memory or caused by an eviction. As mentioned earlier in this section, the proposed



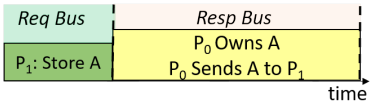
(a) P_1 executes load to A owned by P_0.



(b) P_1 executes store/load to A owned by LLC bank.



(c) P_1 replace A to LLC bank.



(d) P_1 executes store to A owned by P_0.

Figure 2 The sequence of arbitration based on the request type.

architecture applies the arbitration schemes at all different resources including request bus, response bus, and each bank in LLC. Requests can experience different sequences of services on the arbitration resources. In detail, we consider three different *types* $\mathcal{T}$ of requests, depending on the sequence of arbitration: 1) `REQ:RESP:BANK` meaning that it first needs to broadcast on the request bus, then the data response will be propagated on the response bus and finally the data response should be processed at LLC bank; 2) `REQ:BANK:RESP` representing requests that need to first broadcast on the request bus, then the shared bank must process and fetch the data response and finally this data response must be propagated over the response bus; 3) `REQ:RESP`: the last category is related to cache-to-cache transfers. In such a scenario, after broadcasting the message on the request bus, the response will be supplied by the owner core on the response bus.

Figure 2 depicts all possible cases in which a request can be processed based on its type and coherency status. Figure 2a represents a scenario where core P_1 aims to load cache line A; therefore, it first needs to broadcast its action by sending the required coherency message on the request bus. Here, the owner of A is P_0 and the cache line A is in **M** coherency state; hence, according to **MSI** coherency protocol, P_0 must send A to both core P_1 and the LLC bank by pushing the response data into `RxResp` buffer of P_1 and the bank. Then, P_1 receives its data response; however, the data still needs to be processed inside the bank which requires t_{BANK} time. Note that all of these actions are eligible to execute after their corresponding arbitration issue them the grant to access the resource. Hence, the sequence of arbitration for this request follows `REQ:RESP:BANK`.

In Figure 2b a load/store request from P_1 targets cache line A which is owned by the shared bank. Therefore, the bank is responsible to process the request, and then it can be returned to the core through the response bus. Hence, the sequence of arbitration for this request follows `REQ:BANK:RESP`. For the replacement request shown in Figure 2c, after broadcasting the message, the core needs to transfer the data to the LLC bank by sending it to the `RxResp` buffer of bank. Hence, the sequence of arbitration for this request follows `REQ:RESP:BANK`. Finally, Figure 2d shows a scenario where P_1 tries to store to cache line A while the line is owned by P_0. According to the **MSI** coherency protocol, the LLC bank is not required to acknowledge this action; therefore, sending the response from P_0 to P_1 suffices the store request and the sequence of arbitration for this request follows `REQ:RESP`.

Finally, to maintain the correctness of execution, the service order for requests to the same cache line must respect the order in which the requests are issued on the request bus. Once a request starts being issued on the request bus, we say that it *depends* on previous requests to the same cache line which have already been issued on the request bus but not completed yet. Since requests are issued one at a time on the request bus, this means that requests

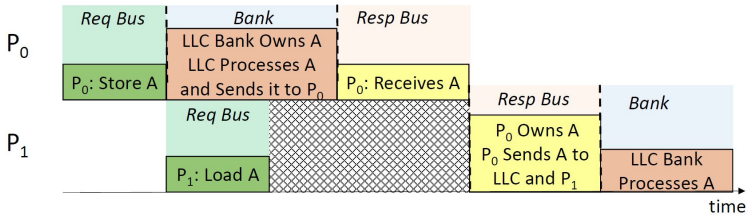


Figure 3 The lower priority request from P_1 *depends* on the higher priority request of P_0 to the same cache line A.

to the same cache line form a *chain of dependencies*, where requests are ordered based on when they are issued on the request bus. Due to dependencies, requests must complete in the same order in which they are issued on the request bus. In addition, some requests might not move to the next resource even though their process is finished at the current resource. Specifically, we say that a request is *ready* on a resource if it can be considered for arbitration at that resource. A request becomes ready on `REQ` when it arrives. For a `RESP` or `BANK` resource, a request becomes ready when it finishes processing at its previous resource, or when the previous request in the chain of dependencies finishes on that resource (if such previous request exists and uses the resource), whichever happens later. Figure 3 shows an example with two requests of different types targeting the same cache line A: the request of P_0 follows the scenario in Figure 2b while P_1 follows Figure 2a. Since the request of P_0 is issued on the request bus first, the request of P_1 depends on it and, to maintain consistency, it cannot start service on the response bus until P_0 finishes receiving the data from the response bus.

3.3 Latency Model

Now, we are able to precisely define the request latency from the core perspective. We assume that the requests of each core P_i can be ordered based on their arrival time. Hence, we can index them as $r_{i,1}, ..., r_{i,j}, ...$ Each request has a type $\mathcal{T}$ according to its sequence. Since OOO cores might issue multiple requests simultaneously and the LLC contains many independent banks, they can serve multiple requests simultaneously. It is thus important to formally define the finish time and processing time of a request.

▶ **Definition 1** (Arrival and Finish Time). *Let $t^a_{i,j}$ be the arrival time of request $r_{i,j}$, that is, the time at which $r_{i,j}$ is queued in `TxMsg`. The finish time $t^f_{i,j}$ of $r_{i,j}$ is the time at which the last action in its sequence is completed. This includes writing to the shared bank if the type of sequence is `REQ:RESP:BANK` or receiving the response from the response bus if the type of sequence is `REQ:BANK:RESP` and `REQ:RESP`. We say a request is outstanding if it has already arrived but is not finished yet. We say a request is pending if it is outstanding and it has already started or completed being issued on the request bus.*

Note that while the concept of a pending request is not used in this section, it will be relevant when defining the arbiter behavior in Section 4 because, as previously explained, requests to the same cache line become dependent on each other once they become pending.

▶ **Definition 2** (Processing Latency). *For any request $r_{i,j}$, let $prec_j$ be the index of the request $r_{i,prec_j}$ of P_i with latest finish time among those that arrived before $r_{i,j}$ (i.e., such that $prec_j < j$). Then the* processing latency *of $r_{i,j}$ is* $\max\left(0, t^f_{i,j} - \max(t^f_{i,prec_j}, t^a_{i,j})\right)$.

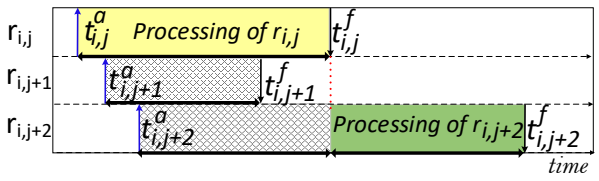


Figure 4 Processing latency example. Assume that all previous requests $r_{i,l}$ with $l < j$ finish before $t^a_{i,j}$. We use $\uparrow$ for the arrival time $t^a_{i,j}$ of each request and $\downarrow$ for its finish time $t^f_{i,j}$.

▶ **Definition 3** (Oldest Request). *At any time t, the oldest request of a requestor (if any) is the earliest arrived request of that requestor that is still outstanding at t.*

Figure 4 delineates a clarifying example, borrowed from [25], where three requests arrive from the same core such that $prec_{j+1} = prec_{j+2} = j$. $r_{i,j}$ becomes the oldest request as soon as it arrives at $t^a_{i,j}$. Initially, both $r_{i,j+1}$ and $r_{i,j+2}$ are non-oldest requests. Similar to related work, we are interested in bounding the processing latency of requests. This is due to the fact that when a requestor issues multiple requests and stalls until they finish, the stall time is upper bounded by the sum of the processing latencies of the requests. In detail, as discussed in [20, 39], the latency suffered by a real-time task running on a core accessing a shared resource can be bounded by the sum of the processing latency of the requests issued by the task. For this reason, the processing latency of $r_{i,j+1}$, which is covered by the processing time of $r_{i,j}$, is set to zero. Notice that a non-oldest request might or might not become the oldest request of its requestor. As shown in the example, $r_{i,j+2}$ becomes oldest once $r_{i,j}$ finishes, while $r_{i,j+1}$ never became oldest and its processing time is zero. For this reason, we only need to consider the processing latency of oldest requests. Finally, if a request becomes oldest, by definition it does so at time $\max(t^f_{i,prec_j}, t^a_{i,j})$ when its processing latency starts.

3.4 Task Analysis

In Section 5, we will derive a bound on the processing latency for each of the three types of request defined in Section 3.2. The total access latency for a task can then be determined by summing the product of the number of requests of each type issued by the task by the WCL for that type [14].

We assume that a portion of accesses by the task targets data shared with other cores, while some accesses are to non-shared data. For each case, we need to retrieve the number of load miss requests, store miss requests, and the number of replacements from the task. For non-shared data, approaches based on either profiling or static analysis can be used to extract the number of requests. For shared data, to the best of our knowledge, no general method exists to determine which cache lines exists in the cache of the other cores at any point of time. A safe assumption can be adopted where every load request on shared data is considered a load miss, and every store request on shared data is considered a store miss [15]. However, if better assumptions can be made based on code analysis, our framework can take advantage of them by deriving different latency bounds for each type of request.

Note that based on Figure 2, for shared data, load misses can be of type `REQ:RESP:BANK` or `REQ:BANK:RESP`, as shown in Figures 2a and 2b, while store misses can be either `REQ:BANK:RESP` or `REQ:RESP`. For non-shared data, load and store misses can only be of type `REQ:BANK:RESP`. Replacements can only follow `REQ:RESP:BANK` as shown in Figure 2c. If we cannot determine the specific type of a request based on task analysis, we simply consider the largest latency among the types to which the request might belong.

4 Proposed Arbiter

This section describes the behavioral details of the proposed arbiter. The proposed arbiter considers the realistic hardware architecture introduced in Section 3 and maintains predictability by design while maximizing average-case performance. Based on the hardware architecture, there exist three distinct types of resources in the system. Formally, we capture the behavior of the proposed arbiter by a set of rules. In order to predictably manage interference among different cores, the arbiter maintains a unified Global Round-Robin (GRR) order of requestors across all resources. A requestor is removed from the GRR queue after the oldest request of that requestor completes at its last resource, and it is inserted at the back of the queue either immediately when it has any other request or when its next request arrives. At any point in time, the *Global Request Queue* shown in Figure 1 contains all outstanding requests in the system as well as their state in terms of their next resource that they need to get processed on. In addition, a work-conserving approach is used at each resource to increase overall system performance. Specifically, the proposed arbiter deploys a two-level arbitration mechanism: 1) oldest requests over non-oldest per core; 2) GRR order among the oldest requests; if no oldest request is ready, GRR over non-oldest requests.

Rule 1. (Global Round-Robin Ordering) The arbiter maintains a Global Round-Robin order of requestors across all resources. Each request is associated with a *GRR priority* as follows: given two outstanding requests $r_{p,q}, r_{i,j}$, $r_{p,q}$ has higher GRR priority than $r_{i,j}$ if: (1) $r_{p,q}$ is oldest and $r_{i,j}$ is not oldest; or (2) $r_{p,q}$ and $r_{i,j}$ are both oldest or both non-oldest, and P_p is ahead of P_i in the GRR order of requestors.

Note that GRR priorities for oldest requests are static, in the sense that they never change while an oldest request is outstanding: this is because the relative requestor order in the GRR queue is fixed once the request becomes oldest. However, GRR priorities for non-oldest requests are not static: specifically, a non-oldest request $r_{p,q}$ might have higher GRR priority than a non-oldest request $r_{i,j}$ at time t, but its GRR priority might become lower than $r_{i,j}$ at some later time t' once an oldest request of P_p completes, forcing P_p to be enqueued at the back of the GRR queue (assuming that $r_{p,q}$ does not become oldest at t').

The arbiter manages each resource independently, selecting the highest priority request that is ready on each resource. Since requests that are ready on the request bus do not depend on other requests, the arbiter manages the request bus according to strict GRR priorities. However, requests that are ready on a bank or the response bus might depend on each other. To correctly arbitrate in the presence of dependant requests, we further introduce a priority inheritance mechanism, where a lower-priority pending request inherits the priority of a higher-priority request that depends on it. Note a further complexity: a lower-priority pending request might target the same cache line as a higher-priority request, but the higher-priority request does not depend on it if the higher-priority request has not yet started being issued on the request bus. However, the higher-priority request will eventually become pending and thus depend on the lower-priority one. To capture such behavior, we extend the concept of dependency to the one of *eventual dependency* to include both requests that are currently dependent, but also requests that will be dependent in the future once they become pending. Based on this concept, we can define dynamic priorities that are used to arbitrate on each bank and the response bus.

Rule 2. (Priority Inheritance) The dynamic priority of a request is equal to the highest priority between its own GRR priority and (if the request is pending) the GRR priority of any other request that eventually depends on it.

Rule 3. (Resource Arbitration) The arbiter manages all three resources, including request bus, response bus, and each LLC banks, independently. On the request bus, the arbiter selects the ready request with the highest GRR priority. On the response bus and each bank, the arbiter selects the request that is ready on the corresponding resource and has the highest dynamic priority.

As before, eventually dependent requests form an *eventual chain of dependencies*, based on the order in which they either already became or will become pending. Note that since the request bus follows strict GRR priorities, an oldest request that is not pending yet will be preceded in the eventual chain of dependency by all already pending requests to the same cache line, as well as by all other oldest requests to the same cache line that are not pending yet and have higher GRR priority.

The proposed arbiter supports out-of-order execution, allowing processing cores to issue multiple requests simultaneously. Based on Rule 3, it is clear that if the arbiter allows many non-oldest requests to the same cache line to be sent, then an oldest request could arrive and suffer priority inversion on all those non-oldest requests. Therefore, to limit the amount of priority inversion in the system, we set a parameter $k_{ceil} \geq 0$, that controls the possibility of sending non-oldest requests ahead of a possible oldest request to the same cache line.

Rule 4. (Request Blocking) When applying Rule 3 on the request bus, the arbiter does not consider a non-oldest request $r_{i,j}$ if there are already other k_{ceil} pending non-oldest requests to the same cache line.

5 Latency Analysis

In this section, we detail the latency analysis for the proposed arbiter. Specifically, consider an oldest request *u*nder *a*nalysis r_{ua} of type $\mathcal{T}$ targeting a bank b_k, and let $t^a_{ua}, t^f_{ua}, t^f_{prec_{ua}}$ be its arrival, finish time, and the finish time of the request with latest finish time among those that arrived before r_{ua} and belong to the same core. We first show how to compute an upper bound to the remaining latency (time to finish) $t^f_{ua} - t_{\text{now}}$ of r_{ua} at time t_{now}, based on the current state of the resource - following related work [25], we call this the *dynamic bound*. Then, we obtain the *static worst-case bound* $\Delta(\mathcal{T}, k_{ceil})$, i.e. an upper bound to the processing latency of any request of type $\mathcal{T}$ for a given value of k_{ceil}, by maximizing the dynamic bound over all possible states of the system at time $t_{\text{now}} = \max(t^f_{prec_{ua}}, t^a_{ua})$ when r_{ua} becomes oldest.

5.1 Dynamic Latency Analysis

Depending on its type $\mathcal{T}$ and its current state at time t_{now}, r_{ua} will need to be serviced on one or more resources; in analogy to processor scheduling, we say that r_{ua} must *execute* on those resources. As in Section 3.2, we use `REQ` to denote the request bus, `RESP` for the response bus, and `BANK` for bank b_k.

We start by proving a fundamental property of the proposed arbitration scheme; namely, the fact that, despite the priority inheritance mechanism in arbitration Rule 2, a lower-priority request $r_{i,j}$ cannot increase its dynamic priority above r_{ua} after time t_{now}.

▶ **Lemma 4.** *Consider any request $r_{i,j}$ other than r_{ua}. If $r_{i,j}$ has lower dynamic priority than r_{ua} at t_{now}, or has not arrived in the system yet, then its dynamic priority cannot become higher than r_{ua} at any time $t > t_{\text{now}}$ while both requests are outstanding.*

Proof. We first show that the dynamic priority of r_{ua} cannot decrease after t_{now}. By arbitration Rule 2, the dynamic priority of r_{ua} at t_{now} is equal to either its GRR priority, or the GRR priority of a higher-priority request $r_{p,q}$ that eventually depends on r_{ua}. Since r_{ua} is oldest, such GRR priorities are static and cannot decrease. Furthermore, as noted in Section 3.2, the coherency protocol forces requests to finish in dependency order. Hence, if r_{ua} inherits the GRR priority of $r_{p,q}$ at t_{now}, then $r_{p,q}$ cannot finish before r_{ua}, and thus the dynamic priority of r_{ua} cannot decrease below the GRR priority of $r_{p,q}$.

Next, we show that the GRR priority of $r_{i,j}$ cannot become higher than the dynamic priority of r_{ua}: (1a) if $r_{i,j}$ is already oldest at time t_{now}, then its GRR priority is static and thus cannot increase; (1b) otherwise, the GRR priority of $r_{i,j}$ increases when it becomes oldest after t_{now}, but it must still be lower than the one of r_{ua} since GRR priorities for oldest requests are based on when they become oldest (which is when their core is pushed to the back of the GRR queue).

In summary, we have shown that the dynamic priority of r_{ua} cannot decrease after t_{now}, and the GRR priority of $r_{i,j}$ cannot increase past it. Therefore, $r_{i,j}$ can only acquire a higher dynamic priority than r_{ua} based on Rule 2 if it inherits the GRR priority of an oldest request $r_{p,q}$ such that: (A) $r_{p,q}$ is either r_{ua} or has a higher GRR priority than r_{ua}; (B) at time t_{now}, $r_{i,j}$ has either not arrived yet, or does not inherit the priority of $r_{p,q}$; (C) at some time t after t_{now}, $r_{i,j}$ is inheriting the priority of $r_{p,q}$. We now show that this is impossible.

Since by (A) the static GRR priority of $r_{p,q}$ must be equal to or higher than r_{ua}, it follows that $r_{p,q}$ must already be outstanding and oldest at t_{now}. We next consider two possible cases: (2a) $r_{p,q}$ is pending at t_{now}; (2b) not pending. In case (2a), note that the set of requests that $r_{p,q}$ depends upon are fixed when $r_{p,q}$ becomes pending; hence (B) and (C) cannot simultaneously hold. In case (2b), for (B) and (C) to be satisfied, $r_{i,j}$ must target the same cache line as $r_{p,q}$ and become pending after t_{now} and before $r_{p,q}$. However, this is again impossible: since the GRR priority of $r_{i,j}$ is lower than r_{ua} and thus $r_{p,q}$, it follows that by Rule 3, $r_{i,j}$ cannot be serviced on the request bus and become pending before $r_{p,q}$. ◀

Based on Lemma 4, we can evaluate at time t_{now} which requests have higher priority than r_{ua} and can thus interfere with it. In details, let $\mathcal{R}_{REQ}$ ($\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$) be the set of outstanding requests with GRR (respectively, dynamic) priority higher than or equal to r_{ua} at time t_{now}, including r_{ua} itself, and which have not yet started executing on `REQ` (respectively, `BANK` or `RESP`) [2]. Since GRR priorities for oldest requests are static, no request that is not included in $\mathcal{R}_{REQ}$ can become higher GRR priority than r_{ua}, and thus interfere with it on `REQ`, at any time $t > t_{\text{now}}$; and by Lemma 4, the same holds for $\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$ on `BANK` and `RESP`.

Next, we formalize the dependencies among requests that target the same cache line as r_{ua}. We do so by constructing a DAG, where each node represents the execution of one request in the eventual chain of dependencies on a resource.

▶ **Definition 5** (Dependency DAG). *The dependency DAG for r_{ua} at time t_{now} is a directed acyclic graph $G = (V, E)$ where: (1) V is a set of nodes; each node is of the form $\{r_{i,j}, res\}$, where $r_{i,j}$ is either r_{ua} or one of the requests that targets the same cache line as r_{ua} and precedes it in the eventual chain of dependencies, while res is one of the resources on which $r_{i,j}$ has not yet finished executing at t_{now}; (2) E is a set of edges of two types: (2a) for*

[2] Note that the sets comprise only requests that have yet to start executing on a resource because in Lemma 7 we will account for the interference of requests that have already started executing on each resource in a different manner

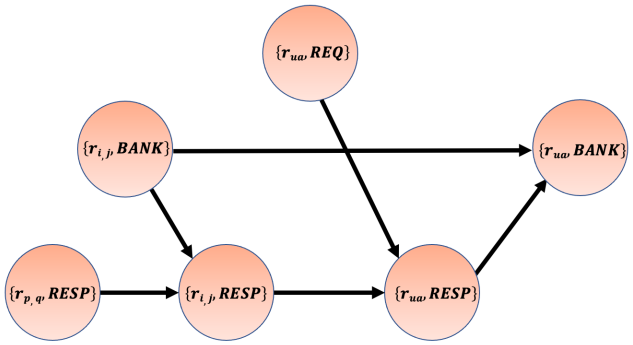


Figure 5 Example dependency DAG. The eventual chain of dependencies comprises requests $(r_{p,q}, r_{i,j}, r_{ua})$.

each request $r_{i,j}$, E includes an edge $\{r_{i,j}, res'\} \to \{r_{i,j}, res\}$ if the two nodes exist in the graph and $r_{i,j}$ executes on res' before res; (2b) for each pair of consecutive requests $r_{p,q}, r_{i,j}$ in the eventual chain of dependencies, E includes edges $\{r_{p,q}, \mathit{RESP}\} \to \{r_{i,j}, \mathit{RESP}\}$ and $\{r_{p,q}, \mathit{BANK}\} \to \{r_{i,j}, \mathit{BANK}\}$ if the corresponding nodes exist in the graph.

Example: assume that at time t_{now}, there are three outstanding requests targeting the same cache line: $r_{p,q}$ of type `REQ:BANK:RESP` became pending first, and has already executed on `BANK` but not yet finished on `RESP`; $r_{i,j}$, also of type `REQ:BANK:RESP`, became pending after $r_{p,q}$ and has completed executing on `REQ` but not yet finished on `BANK`; and r_{ua} is of type $\mathcal{T}$ = `REQ:RESP:BANK` and not yet pending. Then, the eventual chain of dependencies is $(r_{p,q}, r_{i,j}, r_{ua})$, and the dependency DAG is depicted in Figure 5.

Note that by definition and based on Section 3.2, the dependency DAG contains an edge between two nodes whenever the second node cannot become ready before the first one finishes executing. Therefore, if a node $\{r_{i,j}, res\}$ in the dependency DAG has no predecessors, then it must be ready on res at t_{now}. Otherwise, it becomes ready once all predecessor nodes finish executing. We can then define the latency for a node as follows. The latency window for a node $\{r_{i,j}, res\}$ with one or two immediate predecessors spans from the time it becomes ready on res, until the time it finishes executing on res, and its latency is the difference between the two times. If $\{r_{i,j}, res\}$ has no predecessor, then its latency window spans from t_{now} until the time it finishes executing given that we do not want to account for latency before t_{now}. The latency for a (directed) path through G is simply the sum of the latencies of the constituent nodes. Finally, note that as pointed out in Section 4, requests in the eventual chain of dependencies for r_{ua}, and thus the dependency graph, are either already pending or are oldest. Therefore, arbitration Rule 4 cannot block any such request, and thus we do not need to consider it when determining the latency of a path.

We can now derive the remaining latency for r_{ua}. In details, in Lemma 6 we first show that its remaining latency must be equal to the latency of a critical path in the dependency DAG, that is, a path that starts from a node with no precedessors and finishes with the last node of r_{ua}. Then, in Lemma 7, we show that the latency of a critical path is upper bounded by Equation 1. Therefore, the remaining latency of r_{ua} can be computed by taking the maximum of Equation 1 over every potential critical path in the DAG.

▶ **Lemma 6.** *The remaining latency $t^f_{ua} - t_{\text{now}}$ for r_{ua} at time t_{now} is equal to the latency of some path $\mathcal{P}$ in its dependency DAG whose last node is $\{r_{ua}, res\}$ and res is the last resource on which r_{ua} executes. In such* critical path*, each node becomes ready when the previous one finishes executing, except for the first node which has no predecessor and is ready at t_{now}.*

Proof. We iteratively construct the critical path $\mathcal{P}$ as follows. We first start from the path that contains only node $\{r_{ua}, res\}$. Note that by definition, such node finishes executing at t^f_{ua}. We then have two possible cases: (1) the node has no precedessor; (2) it has one or two immediate precedessors. In case (1), r_{ua} is ready on res at t_{now}. Therefore, the latency of the node, which is equal to the latency of $\mathcal{P}$, is $t^f_{ua} - t_{\text{now}}$ and the lemma follows. In case (2), r_{ua} becomes ready on res, and thus the latency window for the node starts, when one of its immediate predecessor nodes finishes executing. Let $\{r_{i,j}, res'\}$ be such node. We can then add it to the beginning of $\mathcal{P}$ and repeat the same reasoning: if the node has no predecessor, then $r_{i,j}$ must be ready on res' at t_{now} and thus its latency window spans from t_{now} to the beginning of the latency window for $\{r_{ua}, res\}$. Therefore, again the latency of $\mathcal{P}$ is equal to $t^f_{ua} - t_{\text{now}}$. If instead $\{r_{i,j}, res'\}$ has one or two immediate precedessors, we continue the iteration by adding one such node to the path. But since the graph is by definition a DAG (hence has no cycles) and the number of nodes is finite, the iteration must eventually terminate. The lemma follows. ◀

▶ **Lemma 7.** *The latency of a critical path $\mathcal{P}$ is upper bounded by:*

$$c_{\overline{res}} + \sum_{res \in \mathcal{S}_\mathcal{P}} |\mathcal{R}_{res}| \cdot t_{res} + K_{BANK}(\mathcal{P}) \cdot (t_{BANK} - 1) + K_{RESP}(\mathcal{P}) \cdot (t_{RESP} - 1), \quad (1)$$

where $\overline{res}$ is the resource on which the first node in $\mathcal{P}$ executes, $c_{\overline{res}}$ is the remaining time to finish executing the current executing request on $\overline{res}$ at t_{now} (if any, otherwise 0), $\mathcal{S}_\mathcal{P}$ is the set of all resources on which nodes in $\mathcal{P}$ execute (with the exception of the first note if it already started executing on $\overline{res}$ at t_{now}), and $K_{BANK}(\mathcal{P})$ ($K_{RESP}(\mathcal{P})$) is the number of nodes in $\mathcal{P}$ that executes on `BANK` (respectively, `RESP`) and are preceded in $\mathcal{P}$ by a node that does not execute on `BANK` (respectively, `RESP`).

Proof. We first consider the case in which the first node in $\mathcal{P}$ has not yet started executing on $\overline{res}$. For each $res \in \mathcal{S}_\mathcal{P}$, we consider the total latency of nodes in $\mathcal{P}$ that execute on res. Since latencies are computed during time windows when a request in $\mathcal{P}$ is ready on res, it follows that the only executions that can contribute to the total latency are: (1) for each node $\{r_{i,j}, res\}$, at most one request that is already executing on res at the beginning of its window. For the first node in $\mathcal{P}$, by definition the length of such execution is $c_{\overline{res}}$. For every other node, since the request is already executing, it can be bound as $t_{res} - 1$. (2) Requests in $\mathcal{P}$ that have not yet started executing on res (since otherwise they would be included in the previous category). (3) Other requests that have not yet started executing on res and have higher GRR priority (if res is `REQ`) or higher dynamic priority (if res is `BANK` or `RESP`) than some request in $\mathcal{P}$ that has not yet started executing on res. Note that lower priority requests can only be included in category (1). Also, as noted in Section 4, requests that precede r_{ua} in the extended chain of dependencies, and can thus be included in $\mathcal{P}$, must either be pending (and hence have higher or equal dynamic priority than r_{ua}) or be non-pending and have higher GRR priority than r_{ua}. Hence, requests in category (2) are included in $\mathcal{R}_{res}$; and since no request that is not included in $\mathcal{R}_{res}$ can acquire higher priority than r_{ua} after t_{now}, requests in category (3) must also be included in $\mathcal{R}_{res}$. Therefore, $|\mathcal{R}_{res}| \cdot t_{res}$ upper bounds the contributions on requests in categories (2) and (3).

It remains to determine the number of requests in (1). Note that if a node executing on res is preceded in $\mathcal{P}$ by a node executing on the same resource (either `RESP` or `BANK`), then no request can be executing at the beginning of its window, since it corresponds to the time at which the preceding node finishes executing on res. Hence, the number of requests can be bounded by the number of nodes that are preceded by another node executing on a

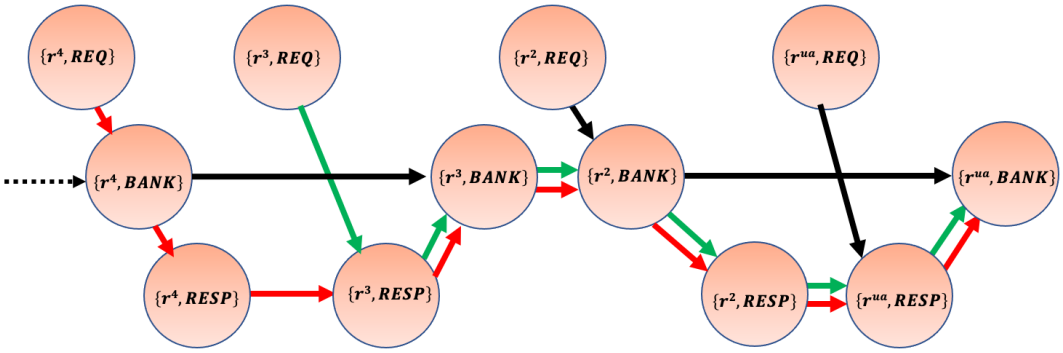


Figure 6 Example paths with maximal number of RESP → BANK and BANK → RESP edges for a r_{ua} of type $\mathcal{T}$ = REQ:RESP:BANK. The eventual chain of dependencies comprises requests $(..., r^4, r^3, r^2, r_{ua})$.

different resource, plus possibly the first node in the path. By definition, this adds a latency of $c_{\overline{res}} + K_{BANK}(\mathcal{P}) \cdot (t_{BANK} - 1) + K_{RESP}(\mathcal{P}) \cdot (t_{RESP} - 1)$. Adding the contribution of categories (2) and (3) to the one of (1) yields Equation 1.

Finally, we consider the case in which the first node in $\mathcal{P}$ is already executing on $\overline{res}$ at time t_{now}. In this case, the latency of the first node is simply $c_{\overline{res}}$, and no other request can execute in its latency window. The same reasoning as above can then be applied to the other nodes in $\mathcal{P}$, given that $\mathcal{S}_{\mathcal{P}}$ does not account for the first node. This again results in a latency of $|\mathcal{R}_{res}| \cdot t_{res}$ for (2) and (3) and a latency of $K_{BANK}(\mathcal{P}) \cdot (t_{BANK} - 1) + K_{RESP}(\mathcal{P}) \cdot (t_{RESP} - 1)$ for (1), which added to $c_{\overline{res}}$ for the first node yields Equation 1. ◀

Example. consider the example in Figure 5. The three possible critical paths are $\mathcal{P}' = \{r_{p,q}, \texttt{RESP}\} \to \{r_{i,j}, \texttt{RESP}\} \to \{r_{ua}, \texttt{RESP}\} \to \{r_{ua}, \texttt{BANK}\}$, $\mathcal{P}'' = \{r_{i,j}, \texttt{BANK}\} \to \{r_{i,j}, \texttt{RESP}\} \to \{r_{ua}, \texttt{RESP}\} \to \{r_{ua}, \texttt{BANK}\}$, and $\mathcal{P}''' = \{r_{ua}, \texttt{REQ}\} \to \{r_{ua}, \texttt{RESP}\} \to \{r_{i,j}, \texttt{BANK}\}$. Note that no critical path can include edge $\{r_{i,j}, \texttt{BANK}\} \to \{r_{ua}, \texttt{BANK}\}$, as the DAG includes the longer path $\mathcal{P}''$ between $\{r_{i,j}, \texttt{BANK}\}$ and $\{r_{ua}, \texttt{BANK}\}$, and thus $\{r_{ua}, \texttt{BANK}\}$ must become ready once $\{r_{ua}, \texttt{RESP}\}$ finishes. Further note that for $\mathcal{P}''$ we have $\overline{res} = \texttt{BANK}, \mathcal{S}_{\mathcal{P}} = \{\texttt{BANK}, \texttt{RESP}\}$, $K_{BANK}(\mathcal{P}'') = 1, K_{RESP}(\mathcal{P}'') = 1$. Its latency can be upper bounded by summing: the remaining time c_{BANK} to finish executing the current executing request on BANK (if any); plus the maximum time $t_{RESP} - 1$ to finish executing a request once $r_{i,j}$ becomes ready on RESP; plus the maximum time $t_{BANK} - 1$ to finish executing a request once r_{ua} becomes ready on BANK; plus the time $|\mathcal{R}_{BANK}| \cdot t_{BANK}$ to execute requests with higher or equal priority (including the ones in the DAG) that have not yet started executing on BANK; plus the time $|\mathcal{R}_{RESP}| \cdot t_{RESP}$ to execute requests with higher or equal priority that have not yet started executing on RESP.

Lemmas 6 and 7 provide a way to estimate the remaining latency for r_{ua}. However, they require constructing the dependency DAG and all possible critical paths. As it will become clear in Section 6, to apply the Duetto reference model we need to estimate the remaining latency online in hardware at every clock cycle. Therefore, we next derive a simpler, albeit conservative, way to determine the remaining latency. Specifically, we replace $K_{BANK}(\mathcal{P})$ and $K_{RESP}(\mathcal{P})$ with upper bounds $\overline{K_{BANK}}(C)$ and $\overline{K_{RESP}}(C)$ which depend only on the number C of requests in the dependency DAG; this ensures that at run-time, we only need to maintain the list of requests in the eventual chain of dependencies and which resources they need to execute upon, but not the detailed DAG structure.

▶ **Lemma 8.** *For $C \geq 1$, define:*

$$\overline{K_{BANK}}(\mathcal{T}, C) = \begin{cases} \lfloor (C+1)/2 \rfloor & \text{if } \mathcal{T} = \texttt{REQ:BANK:RESP} \\ \lceil (C+1)/2 \rceil & \text{if } \mathcal{T} = \texttt{REQ:RESP:BANK} \\ \lceil (C-1)/2 \rceil & \text{if } \mathcal{T} = \texttt{REQ:RESP} \end{cases} \quad (2)$$

$$\overline{K_{RESP}}(\mathcal{T}, C) = \begin{cases} \lceil (C+1)/2 \rceil & \text{if } \mathcal{T} = \texttt{REQ:BANK:RESP} \\ \lfloor (C+1)/2 \rfloor & \text{if } \mathcal{T} = \texttt{REQ:RESP:BANK} \\ \lfloor (C+1)/2 \rfloor & \text{if } \mathcal{T} = \texttt{REQ:RESP} \end{cases} \quad (3)$$

For a r_{ua} of type $\mathcal{T}$ and any critical path $\mathcal{P}$ comprising nodes of C requests, it holds: (1) $K_{BANK}(\mathcal{P}) \leq \overline{K_{BANK}}(C) - 1$ if the first node in $\mathcal{P}$ executes on BANK*, and $K_{BANK}(\mathcal{P}) \leq \overline{K_{BANK}}(C)$ otherwise; (2) $K_{RESP}(\mathcal{P}) \leq \overline{K_{RESP}}(C) - 1$ if the first node in $\mathcal{P}$ executes on* RESP*, and $K_{RESP}(\mathcal{P}) \leq \overline{K_{RESP}}(C)$ otherwise.*

Proof. First note that by construction, only the first node in $\mathcal{P}$ can execute on REQ; all other nodes must execute on either BANK or RESP. Therefore, $K_{BANK}(\mathcal{P})$ and $K_{RESP}(\mathcal{P})$ are maximized when $\mathcal{P}$ comprises the maximum number of edges from a node executing on RESP to a node on BANK (an edge RESP $\rightarrow$ BANK), and from a node on BANK to a node on RESP (an edge BANK $\rightarrow$ RESP). Note that if two consecutive requests in the eventual chain of dependencies are of the same type (for example, REQ:RESP:BANK), then any path over BANK and RESP nodes belonging to those two requests can have only one such edge (in this example, the two possible paths are RESP $\rightarrow$ BANK $\rightarrow$ BANK and RESP $\rightarrow$ RESP $\rightarrow$ BANK); on the other hand, if two consecutive requests are one of type REQ:RESP:BANK and the other of type REQ:BANK:RESP, then the path can have one edge RESP $\rightarrow$ BANK and one BANK $\rightarrow$ RESP. Hence, $K_{BANK}(\mathcal{P}), K_{RESP}(\mathcal{P})$ are maximized when requests in the eventual chain of dependencies switch between the two types.

Next consider $\mathcal{T}$ = REQ:RESP:BANK. Figure 6 shows the resulting DAG when requests in the chain switch types, together with two possible critical paths with $C = 4$ (even) and $C = 3$ (odd) requests, where the first node executes on REQ. By construction, the path comprises an edge RESP $\rightarrow$ BANK for r_{ua}, and for every second request before it; hence, the number of such edges is $\lceil C/2 \rceil$. In addition, if C is even, there is a REQ $\rightarrow$ BANK edge for the first request. Hence, the maximum value of $K_{BANK}(\mathcal{P})$ can be computed as $\overline{K_{BANK}}$(REQ:RESP:BANK, C) $= \lceil (C+1)/2 \rceil$; except that if $\mathcal{P}$ starts with a node on BANK, then that node is not preceded by any other node, hence the maximum value of $K_{BANK}(\mathcal{P})$ is $\overline{K_{BANK}}$(REQ:RESP:BANK, C) $- 1$. Similarly for $K_{RESP}(\mathcal{P})$, we note that the path comprises an edge BANK $\rightarrow$ RESP for the request before r_{ua}, and for every second request before it; plus an edge REQ $\rightarrow$ RESP if C is odd. Hence, the maximum value of $K_{RESP}(\mathcal{P})$ can be computed as $\overline{K_{RESP}}$(REQ:RESP:BANK, C) $= \lfloor (C+1)/2 \rfloor$, or $\overline{K_{RESP}}$(REQ:RESP:BANK, C) $- 1$ if the first node in $\mathcal{P}$ is on RESP. This concludes the proof for $\mathcal{T}$ = REQ:RESP:BANK.

For brevity, we omit the proof for $\mathcal{T}$ = REQ:RESP and $\mathcal{T}$ = REQ:BANK:RESP, since the derivation is equivalent; in particular, note that $\overline{K_{RESP}}$(REQ:RESP, C) $= \overline{K_{RESP}}$(REQ:RESP:BANK, C) but $\overline{K_{BANK}}$(REQ:RESP, C) $= \overline{K_{BANK}}$(REQ:RESP:BANK, C) $- 1$, since with $\mathcal{T}$ = REQ:RESP we miss the RESP $\rightarrow$ BANK edge for r_{ua}; while for $\mathcal{T}$ = REQ:BANK:RESP we have $\overline{K_{RESP}}$(REQ:BANK:RESP, C) $= \overline{K_{BANK}}$(REQ:RESP:BANK, C) and $\overline{K_{BANK}}$(REQ:BANK:RESP, C) $= \overline{K_{RESP}}$(REQ:RESP:BANK, C) since the two cases are specular. ◀

▶ **Theorem 9.** *The remaining latency $t^f_{ua} - t_{\text{now}}$ for a r_{ua} of type $\mathcal{T}$ at time t_{now} is upper bounded by:*

$$c_{init} + \sum_{res \in \mathcal{S}_{ua}} |\mathcal{R}_{res}| \cdot t_{res} + \overline{K_{BANK}}(\mathcal{T}, C) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, C) \cdot (t_{RESP} - 1), \quad (4)$$

where $\mathcal{S}_{ua}$ is the set of all resources on which the C requests for nodes in the dependency DAG have not yet started executing, and $c_{init} = c_{REQ}$ if any node in the DAG executes on `REQ`*, $c_{init} = 0$ otherwise.*

Proof. By Lemma 6, the remaining latency of r_{ua} is equal to the latency of a critical path $\mathcal{P}$, which is upper bounded by Equation 1. Note that by definition, only the first node in $\mathcal{P}$ might have already started executing at t_{now}. Hence, it holds: $\mathcal{S}_{\mathcal{P}} \subseteq \mathcal{S}_{ua}$, which implies $\sum_{res \in \mathcal{S}_{\mathcal{P}}} |\mathcal{R}_{res}| \cdot t_{res} \leq \sum_{res \in \mathcal{S}_{ua}} |\mathcal{R}_{res}| \cdot t_{res}$. Let C' be the number of requests for nodes in $\mathcal{P}$; by definition, $C' \leq C$. By cases on the resource on which the first node in $\mathcal{P}$ executes.

`REQ`: by definition we have $c_{init} = c_{REQ} = c_{\overline{res}}$ and by Lemma 8 and since $\overline{K_{BANK}}(\mathcal{T}, C)$, $\overline{K_{RESP}}(\mathcal{T}, C)$ are monothonic in C it holds $K_{BANK}(\mathcal{P}) \leq \overline{K_{BANK}}(\mathcal{T}, C') \leq \overline{K_{BANK}}(\mathcal{T}, C)$, $K_{RESP}(\mathcal{P}) \leq \overline{K_{RESP}}(\mathcal{T}, C') \leq \overline{K_{RESP}}(\mathcal{T}, C)$. Hence, the latency of $\mathcal{P}$ in Equation 1 is upper bounded by Equation 4.

`RESP`: we have $c_{init} = 0, c_{\overline{res}} = c_{RESP}$ and by Lemma 8 and monotonicity it holds $K_{RESP}(\mathcal{P}) \leq \overline{K_{RESP}}(\mathcal{T}, C) - 1, K_{BANK}(\mathcal{P}) \leq \overline{K_{BANK}}(\mathcal{T}, C)$. Hence starting from Equation 1 and noting that by definition it must hold $c_{RESP} \leq t_{RESP} - 1$, we obtain:

$$
\begin{aligned}
& c_{RESP} + \sum_{res \in \mathcal{S}_{\mathcal{P}}} |\mathcal{R}_{res}| \cdot t_{res} + K_{BANK}(\mathcal{P}) \cdot (t_{BANK} - 1) + K_{RESP}(\mathcal{P}) \cdot (t_{RESP} - 1) \\
\leq\ & c_{RESP} - (t_{RESP} - 1) + \sum_{res \in \mathcal{S}_{\mathcal{P}}} |\mathcal{R}_{res}| \cdot t_{res} + \overline{K_{BANK}}(\mathcal{T}, C) \cdot (t_{BANK} - 1) + \\
& + \overline{K_{RESP}}(\mathcal{T}, C) \cdot (t_{RESP} - 1) \\
\leq\ & \sum_{res \in \mathcal{S}_{ua}} |\mathcal{R}_{res}| \cdot t_{res} + \overline{K_{BANK}}(\mathcal{T}, C) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, C) \cdot (t_{RESP} - 1), \quad (5)
\end{aligned}
$$

hence the latency is again upper bounded by Equation 4.

`BANK`: we have $c_{init} = 0, c_{\overline{res}} = c_{BANK}$ and by Lemma 8 and monotonicity it holds $K_{BANK}(\mathcal{P}) \leq \overline{K_{BANK}}(\mathcal{T}, C) - 1, K_{RESP}(\mathcal{P}) \leq \overline{K_{RESP}}(\mathcal{T}, C)$; repeating the same derivation as for the `RESP` case, we again find that Equation 1 is upper bounded by Equation 4. ◀

5.2 Static Analysis

We compute the static worst-case latency $\Delta(\mathcal{T}, k_{ceil})$ by maximizing Equation 4 over all possible values of the parameters. The resulting bounds in Theorem 10 depend on k_{ceil} and M: by definition, there are at most M oldest request at any point in time, meaning that at most $M - 1$ requests can have higher GRR priority than r_{ua}. The theorem computes two separate bounds for $k_{ceil} = 0$ and $k_{ceil} > 0$: for the former, in the worst-case all M oldest requests target the same cache line, while for the latter, each oldest request targets a different cache line and suffers interference from k_{ceil} non-oldest requests based on arbitration Rule 4.

▶ **Theorem 10.** *The static latency for any request of type $\mathcal{T}$ is upper bounded by:*

$$
\begin{aligned}
& \Delta(\mathcal{T}, 0) = t_{REQ} - 1 + M \cdot t_{REQ} + M \cdot t_{BANK} + M \cdot t_{RESP} \\
& + \overline{K_{BANK}}(\mathcal{T}, M) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, M) \cdot (t_{RESP} - 1) \quad (6)
\end{aligned}
$$

if $k_{ceil} = 0$, while if $k_{ceil} > 0$ it is upper bounded by:

$$
\begin{aligned}
& \Delta(\mathcal{T}, k_{ceil}) = t_{REQ} - 1 + M \cdot t_{REQ} + M \cdot (k_{ceil} + 1) \cdot t_{BANK} + M \cdot (k_{ceil} + 1) \cdot t_{RESP} \\
& + \overline{K_{BANK}}(\mathcal{T}, k_{ceil} + 1) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, k_{ceil} + 1) \cdot (t_{RESP} - 1). \quad (7)
\end{aligned}
$$

Proof. By definition, the static latency is equal to the maximum remaining latency of any request r_{ua} of type $\mathcal{T}$ that becomes oldest at time t_{now}. By Theorem 9, such latency is upper bounded by Equation 4; hence, we can upper bound $\Delta(\mathcal{T}, k_{ceil})$ by maximizing Equation 4.

Note that the equation is maximized by using the maximum value $c_{init} = c_{REQ} = t_{REQ}-1$, and latency contributions on all three resources such that $\mathcal{S}_{ua} = \{\texttt{REQ},\texttt{BANK},\texttt{RESP}\}$. By definition, the set $\mathcal{R}_{REQ}$ can comprise at most r_{ua} itself and one oldest request for each other core; hence, we have at most $|\mathcal{R}_{REQ}| = M$.

We next consider the number of requests C for nodes in the dependency DAG of r_{ua}, which comprise r_{ua} and requests that precede r_{ua} in its eventual chain of dependencies, as well as the number of requests $|\mathcal{R}_{BANK}|, |\mathcal{R}_{RESP}|$. Let H be the number of oldest requests that precede r_{ua} in the eventual chain of dependencies. By Rule 4, the number of non-oldest requests that precede r_{ua} in the chain is bounded by k_{ceil}. Hence, the maximum number of requests in the dependency DAG is $C = k_{ceil} + H + 1$. Since requests in $\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$ must have higher dynamic priority than r_{ua}, the only requests that can be included are: (1) the C requests in the DAG; (2) the remaining $M - H - 1$ oldest requests; (3) non-oldest requests that inherit the priority of such $M - H - 1$ oldest requests; again by Rule 4, their number is bounded by $(M - H - 1) \cdot k_{ceil}$. Summing over (1)-(3), we obtain: $|\mathcal{R}_{RESP}| = |\mathcal{R}_{BANK}| = M \cdot (k_{ceil} + 1) - H \cdot k_{ceil}$. If $k_{ceil} = 0$, then the cardinality of the sets is constant in H, while C, and thus the values of $\overline{K_{RESP}}(\mathcal{T}, C)$ and $\overline{K_{RESP}}(\mathcal{T}, C)$, are non-decreasing in H; hence, Equation 4 is maximized when $H = M - 1$, yielding Equation 6.

Finally, consider the case of $k_{ceil} > 0$. Note that based on Equation 3, 2 and independently from $\mathcal{T}$, it holds: $\overline{K_{RESP}}(\mathcal{T}, k_{ceil} + 1 + H) \leq \overline{K_{RESP}}(\mathcal{T}, k_{ceil} + 1) + H$, and similarly $\overline{K_{BANK}}(\mathcal{T}, k_{ceil} + 1 + H) \leq \overline{K_{BANK}}(\mathcal{T}, k_{ceil} + 1) + H$. Substituting the values of the parameters in Equation 4 we thus obtain:

$$
\begin{aligned}
& t_{REQ} - 1 + M \cdot t_{REQ} + M \cdot (k_{ceil} + 1) \cdot t_{BANK} - H \cdot k_{ceil} \cdot t_{BANK} \\
& + M \cdot (k_{ceil} + 1) \cdot t_{RESP} - H \cdot k_{ceil} \cdot t_{RESP} \\
& + \overline{K_{BANK}}(\mathcal{T}, k_{ceil} + 1 + H) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, k_{ceil} + 1 + H) \cdot (t_{RESP} - 1) \\
\leq\ & t_{REQ} - 1 + M \cdot t_{REQ} + M \cdot (k_{ceil} + 1) \cdot t_{BANK} - H \cdot k_{ceil} \cdot t_{BANK} \\
& + M \cdot (k_{ceil} + 1) \cdot t_{RESP} - H \cdot k_{ceil} \cdot t_{RESP} \\
& + \overline{K_{BANK}}(\mathcal{T}, k_{ceil} + 1) \cdot (t_{BANK} - 1) + H \cdot (t_{BANK} - 1) + \\
& + \overline{K_{RESP}}(\mathcal{T}, k_{ceil} + 1) \cdot (t_{RESP} - 1) + H \cdot (t_{RESP} - 1) \\
\leq\ & t_{REQ} - 1 + M \cdot t_{REQ} + M \cdot (k_{ceil} + 1) \cdot t_{BANK} + M \cdot (k_{ceil} + 1) \cdot t_{RESP} \\
& + \overline{K_{BANK}}(\mathcal{T}, k_{ceil} + 1) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, k_{ceil} + 1) \cdot (t_{RESP} - 1), \qquad (8)
\end{aligned}
$$

which is the expression in Equation 7. ◀

6 Applying Duetto to Cache Coherence Design

In this section, we first briefly review the Duetto reference model introduced in [25] to address the average performance and predictability trade-off in shared resource management in multi-core systems. We then discuss how the reference model must be extended to account for the peculiarities of our cache system and form the DUEPCO architecture.

6.1 Background: Duetto Reference Model

The key idea behind Duetto is that it augments a COTS High-Performance Arbiter (HPA) with a Real-time Arbiter (RTA) such that both arbiters operate in parallel. The RTA is analyzable in the sense that it provides strict latency bounds on requests; specifically, we

use the arbiter described in Section 4. The HPA is designed to maximize average-case performance. In our implementation, the HPA uses commodity FCFS at all resources but respects the dependencies among requests to the same cache line. Arbiters control resources by issuing commands—for our cache resource, the command dictates which requests, if any, should start executing on each resource. Every clock cycle, Duetto selects either the command from the HPA or the command from the RTA and issues it to the resources. Ideally, the system utilizes the HPA most of the time, hence benefiting from its performance gains, and only switches to the RTA if there is a risk that a latency guarantee will be missed. Note that the global request queue is shared by the HPA and RTA. Since both arbiters make decisions based on the requests stored in the queue, a request is removed from the queue only after it finishes based on the actual commands issued by Duetto. Given that both arbiters can process requests out-of-order, this does not add extra complexity to the queue implementation.

In more details, for each requestor P_i and each request type $\mathcal{T}$, Duetto associates a relative deadline $D_i(\mathcal{T})$ which represents the maximum tolerable processing latency for any oldest request of that type and requestor. Such deadline can be configured by the system designer (for example, by writing to memory-mapped registers exposed by the hardware), and must be used in place of the static WCL when performing task analysis. Higher deadline values increase the worst-case resource access latency for a task, but can enable Duetto to remain in HPA mode longer. A *DTracker* module is responsible for tracking the absolute deadline of each oldest request based on its associated relative deadline and the time it becomes oldest. As long as $D_i(\mathcal{T}) \geq \Delta(\mathcal{T}, k_{ceil})$ for all requestors and types, Duetto formally guarantees that all absolute deadlines will be met. This is achieved by using a Worst-case Latency Estimator (WCLator) module to estimate the remaining processing latency of each outstanding request at run-time, assuming that Duetto selects the HPA in the current clock cycle and then switches to the RTA in all future cycles. If for every requestor, the estimated finish time of its oldest request from WCLator is lower than or equal to its absolute deadline, then Duetto selects the HPA. Otherwise, it selects the RTA. The key observation is that using online information on the state of the system (resources, state of the RTA arbiter, and queued requests) allows us to greatly reduce the pessimism inherent in the static latency computation; hence, unless the system becomes overloaded, Duetto can keep selecting the HPA. Note that in [25], the model is demonstrated on a simple resource. However, in [26], Duetto has been applied to the design of a more complex DRAM controller where, similarly to our cache system, each request requires multiple commands to be serviced.

6.2 Model Extensions

Compared to the approach in [25, 26], we must extend the reference model in two fundamental ways to apply it to our cache design. First of all, it is important to notice that for the Duetto deadline guarantee to hold, the static worst-case latency must be computed assuming any valid state of the resource at the time t when the request under analysis becomes oldest; this is because the HPA might be selected at any time before t. However, when we computed the static latency in Theorem 10, we bounded the cardinality of sets $\mathcal{R}_{BANK}$ and $\mathcal{R}_{RESP}$ assuming that arbitration Rule 4 always applies, as this ensures that no more than k_{ceil} non-oldest requests can inherit the priority of an oldest request. Unfortunately, the HPA does not need to satisfy such rule, and can instead execute any number of requests to the same cache line on the `REQ` bus before an oldest request to that line arrives and its latency is considered by the WCLator; at which point it is too late to switch to the RTA.

To address this issue, we make a conceptual change to the model of the resource. Specifically, we declare that all states where there are more than k_{ceil} pending non-oldest requests to the same cache line are invalid. This ensures that the derived static bound is correct, but does not solve the underline problem as now the HPA might be issuing invalid commands. In [25], we suggest that when the HPA cannot be guaranteed to work correctly, a checker module can be added to check the validity of the commands issued by the HPA. Therefore, we add an additional checker component that works as follows: every clock cycle, the checker receives from the RTA information on the number of pending requests per cache line, which the RTA maintains to enforce Rule 4. If $c_{REQ} = 0$ and the global request queue contains at least one non-oldest request that must execute on the `REQ` bus and targets a cache line for which there are k_{ceil} pending non-oldest requests, the HPA might issue such request on `REQ` and reach an invalid resource state. Hence, in this case the checker overrides the WCLator to forcibly select the RTA. While this approach solves the unbounded priority inversion problem, it has a downside: for low values of k_{ceil} and/or heavy data sharing among requestors, the checker might be forced to continuously select the RTA, resulting in performance loss. We explore this behavior in more details in the evaluation Section 7.

The second extension is related to the request type. Both [25] and [26] assume that the type of a request is known when the request becomes oldest. However, in our system, the sequence of resources accessed by a request, and thus its type, is only known after the request is executed on the `REQ` bus and the owner of the corresponding cache line is determined. For this reason, before an oldest request finishes executing on `REQ`, the WCLator must use the smallest among all deadlines for the possible types for the request. Once the request type is known, the WCLator switches to using the deadline for that type.

6.3 WCLator Design

We designed the WCLator following the methodology outlined in [25]. For each oldest request and given the state of the resource and global request queue, we first enumerate all commands that the HPA could issue in this clock cycle. For each command, we then use the dynamic analysis of Theorem 9 (possibly with modified value of the parameters) to compute the remaining latency for the request. The WCLator then compares the largest computed latency against the deadline for the request to determine whether the HPA can be selected. As noted in Section 6.2, in our implementation we do not know the type of a request until it finishes executing on `REQ`. Hence, to safely account for the values of $\mathcal{S}_{ua}$, $\mathcal{R}_{BANK}$ and $\mathcal{R}_{RESP}$ in Equation 4, we assume that all outstanding requests that have not yet finished executing on `REQ` must access both `BANK` and `RESP`.

Consider an oldest request r_{ua}. To illustrate the behavior of the WCLator, we enumerate the cases assuming that $c_{init} = c_{REQ}$ (the case for $c_{init} = 0$ is similar but easier, since the chain of dependencies for r_{ua} cannot be affected):

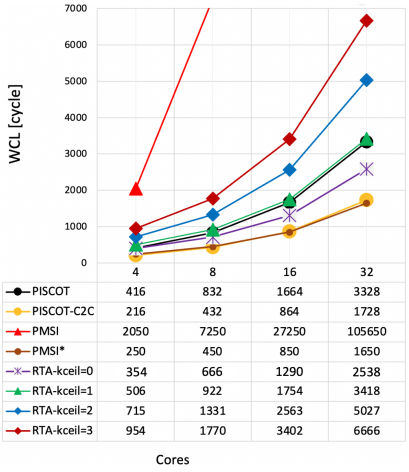
1. If $c_{REQ} > 0$, then no command can be issued on `REQ` in this clock cycle, and no estimation is required. Note that if $c_{BANK} = 0$ or $c_{RESP} = 0$, the HPA could start executing a request on `BANK` or `RESP` in this clock cycle; however, because Equation 4 always assumes the worst case where the maximum blocking time is suffered on successive resources, it follows that the bound is still safe no matter the command issued by HPA on `BANK` and/or `RESP`. Therefore, for the remaining cases, we assume $c_{REQ} = 0$ and consider the command issued on `REQ`.
2. No command: the HPA might be non-work conserving and decide to issue no command in the current cycle. In this case, the bound is equal to Equation 4 plus one, to account for the wasted clock cycle.

3. r_{ua}: since r_{ua} will start executing, we would need to apply Equation 4 after removing it from $\mathcal{R}_{REQ}$, but setting $c_{REQ} = t_{REQ}$ to account for the r_{ua} execution. In addition, if there are higher-priority requests to the same cache line as r_{ua} which have not yet executed on `REQ`, we would need to remove such requests from $\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$ and adjust $\overline{K_{BANK}}(\mathcal{T}, C), \overline{K_{RESP}}(\mathcal{T}, C)$. Note that the obtained bound will always be lower than case 2); therefore, in practice the WCLator does not need to consider this case.
4. A lower-priority request $r_{i,j}$, which does not inherit a higher priority than r_{ua} after becoming pending: we use Equation 4 with $c_{REQ} = t_{REQ}$.
5. A lower-priority request $r_{i,j}$ that inherits a higher priority than r_{ua}: in addition to the previous case, we need to include the request in $\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$. Furthermore, if $r_{i,j}$ targets the same cache line as r_{ua}, $\overline{K_{BANK}}(\mathcal{T}, C), \overline{K_{RESP}}(\mathcal{T}, C)$ must be adjusted.
6. A higher-priority request $r_{p,q}$ to either the same or a different cache line than r_{ua}: we use Equation 4 with no change to parameters. Since this bound is always lower than 2), again we do not need to consider it.

In this work, our goal is to evaluate the performance of the proposed system architecture and arbitration scheme based on cycle-accurate simulation; a full system implementation is deferred to future work. Nonetheless, given its potential complexity, we briefly comment on the WCLator implementation. Since the WCLator is a hardware component, all the cases above can be estimated in parallel for each request under analysis, and each result compared against the request deadline. Hence, the hardware latency is dominated by the computation of Equation 4. The value of c_{REQ} can be maintained by a simple counter. Similarly, the value of $|\mathcal{R}_{res}|$ can be maintained in separate counters for each r_{ua} and each resource based on the state of the GRR arbitration. Multiplications can be avoided by storing the corresponding values in look-up tables indexed by the corresponding parameter. Another counter can be used to keep track of the number of requests C in the eventual chain of dependencies for each r_{ua}, and index a look-up table that returns the value of $\overline{K_{BANK}}(\mathcal{T}, C) \cdot (t_{BANK} - 1) + \overline{K_{RESP}}(\mathcal{T}, C) \cdot (t_{RESP} - 1)$. The final computation is then obtained by adding at most 5 terms together, which can be performed with cascaded adders of width at most 11 bits for $M = 8$ requestors. In summary, we do not expect the WCLator implementation to constrain the clock speed.

7 Evaluation Results

We employed the open-source, cycle-accurate architectural simulation framework provided by [15] to evaluate the performance of the proposed mechanisms and compare them with other solutions. We emulate quad- and octa-core systems clocked at 2.5 GHz. Each system has a 32 KB 4-way set-associative per-core private L1 data cache (similar to ARM Cortex A53 [1]), and a 4 MB 8-ways set-associative L2 shared cache consisting of multiple separated banks. The cores are OOO and can issue up to 10 memory requests in parallel. Both L1 and LLC have a cache line size of 64 bytes. Each core/LLC bank is equipped with a dedicated cache controller that implements the MSI coherence state machine. We compare results for the proposed arbiter described in Section 4, which we denote as RTA, against state-of-the-art approaches including PMSI [10], PMSI* [17] and PISCOT [15] which also provide analytical WCL bounds and present the best average-case performance. PMSI employs unified bus architecture and provides relative high-performance gains compared to other approaches such as shared data-aware scheduling and private cache bypassing through deploying cache coherence modifications and accessing the shared data. However, its WCL is quadratic in the number of cores in the system. PMSI* follows a systematic approach that achieves the same



	4	8	16	32
PISCOT	416	832	1664	3328
PISCOT-C2C	216	432	864	1728
PMSI	2050	7250	27250	105650
PMSI*	250	450	850	1650
RTA-kceil=0	354	666	1290	2538
RTA-kceil=1	506	922	1754	3418
RTA-kceil=2	715	1331	2563	5027
RTA-kceil=3	954	1770	3402	6666

Figure 7 Per-request worst-case latency.

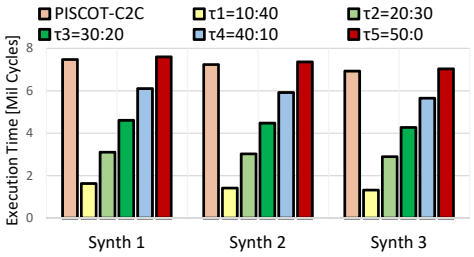


Figure 8 Sensitivity test for RTA against PISCOT-C2C.

static WCL as bypassing the shared cache and provides a tighter WCL bound compared to PMSI. However, both of these techniques rely on many coherency modifications and expose performance loss compared to other approaches. On the other hand, PISCOT decouples the request and response bus and leverages the split-transaction interconnect to achieve a tighter WCL compared to PMSI and considerable performance gains. We also compare against the COTS HPA as described in Section 6.1, which aims to achieve high average-case performance.

Request bus latency is configured to 4 cycles ($t_{REQ} = 4$). The response bus latency in PISCOT is comparable to the TDM slot size in PMSI as well as PMSI* and we set them to 50 cycles in our evaluation similar to [15]. However, for RTA, the latency of all resources is configurable. Throughout this section, unless otherwise specified, we configure RTA with $t_{RESP} = 10$ cycles, 8 banks that consume $t_{BANK} = 40$ cycles to process requests and parameter $k_{ceil} = 1$. Similar to existing works [18, 10, 15], we assume that accesses that hit in the L1 cache take a single clock cycle and, as discussed in Section 3.1, LLC is a perfect cache (all LLC accesses are hits) to avoid extra delay from accessing the off-chip memory subsystem.

We employ SPLASH-3 [31] benchmark suite as it is a representative of multi-threaded applications with data sharing. In addition, we craft a wide set of synthetic benchmarks that stresses the implemented solutions. All contain mixed read and write requests to the LLC and we engineered the requests' addresses such that all requests miss in the L1 cache; hence, stress on the bus and the shared cache banks will be maximized. Different benchmarks in this set exhibit different sharing percentage as well. Due to space limitation, we show the results of three of these benchmarks (`Synth 1`, `Synth 2`, `Synth 3`) that show unique insights. There is no data-sharing among the cores in `Synth 1` while `Synth 2` and `Synth 3` exhibit 10% and 20% data-sharing respectively. In all benchmarks, the foreground core represents a high load core that bursts requests to bus/LLC, and the background cores are accessing the shared bus/LLC less frequently. Interleaving across the banks is handled using address bits themselves such that a core could access all banks as much as possible. In detail, we use bit 6^{th} (bits zero to 5^{th} are for the cache line offset) towards the MSB in the address bits of the request to determine which LLC bank it needs to be processed in.

7.1 Per-Request Worst-Case Latency

Figure 7 shows the static WCL bounds for requests generated by the cores and misses in L1 caches (see Section 5) from `REQ:RESP:BANK` type which represents the largest static WCL among the three types. We compare PMSI, PMSI*, PISCOT and PISCOT-C2C (with core to core transfers), and the proposed RTA mechanism with different values of the configurable parameter k_{ceil}. From this experiment, we can make the following observations: 1) PMSI shows a significantly higher latency bound compared to the other approaches, and the latency bound increases quadratically with scaling the number of cores. The significant added latency is due to the coherence interference on the shared data. PMSI* on the other hand presents tight static WCL bound but at the cost of performance degradation [15, 17]; 2) PISCOT shows looser bound compared to both PISCOT-C2C and PMSI* but similar to RTA since core to core transfers enable the arbiters to bypass the LLC when an owner core must respond to other cores; 3) RTA with $k_{ceil} = 1$ shows up to 1.18× looser bound compared to PISCOT but significantly tighter bound compared to PMSI. Notice that this extra amount in latency bound is due to the scheduling decisions that are made in RTA which allow one non-oldest request to process in LLC banks. This gives the system a significant advantage in terms of average performance as we will show in the next sections. It is worthwhile to stress the existing trade-off between RTA with different values of k_{ceil} and PISCOT-C2C. RTA with $k_{ceil} = 0$ represents a configuration that forces the ordering of processing such that requests are only processed if they are oldest (no non-oldest request is allowed to process in the shared banks). This improves the WCL bound such that it becomes tighter and very similar to PISCOT-C2C. However, Duetto does not work with this configuration of RTA ($k_{ceil} = 0$) since this prevents the system from reasonably leveraging the performance of the HPA; the checker module is forced to select the RTA if there is any non-oldest request needing to be serviced on the request bus.

7.2 RTA Sensitivity Test

The underlying architecture proposed in Section 3 is fully configurable to resemble the conventional high-performance bus/LLC designs. In this section, we conduct a sensitivity test on the RTA using synthetic benchmarks to justify the most efficient (and the worst) design that is aligned with commercial architectures and compare it against PISCOT-C2C. We configured a quad-core system with $t_{REQ} = 4$ and then gradually varied t_{BANK} and t_{RESP} latencies. In order to run a fair comparison, the parameters are determined such that $t_{RESP}+t_{BANK} = 50$, the response bus latency for PISCOT-C2C. Assuming $\tau = t_{RESP} : t_{BANK}$ represents a configuration of RTA in which the latency of shared banks in LLC is t_{BANK} and the latency of response bus equals t_{RESP}, Figure 8 shows the execution time of the foreground core running each of the three synthetic benchmarks. As discussed, RTA increases the parallelism through bankized LLC. Therefore, as we increase t_{BANK} in LLC and coincidentally decrease t_{RESP}, we observe that the system performance improves by finishing the task under analysis faster. In other words, by reducing the response bus latency, a significant amount of arbitration stress will be transferred to the banks rather than the response bus; hence, the system's overall performance increases by allowing more transactions to be serviced simultaneously. In detail, the core under analysis in RTA, τ_1 running `Synth 1` outperforms PISCOT-C2C by 4.58× in terms of overall throughput of the system. Note that in τ_5 where there is no parallelism in RTA, we observe a negligible performance loss compared to PISCOT-C2C (maximum 1% in overall throughput) since response bus arbiter in PISCOT-C2C is FCFS while RTA employs a fair round-robin mechanism through GRR. Going forward, we chose τ_1

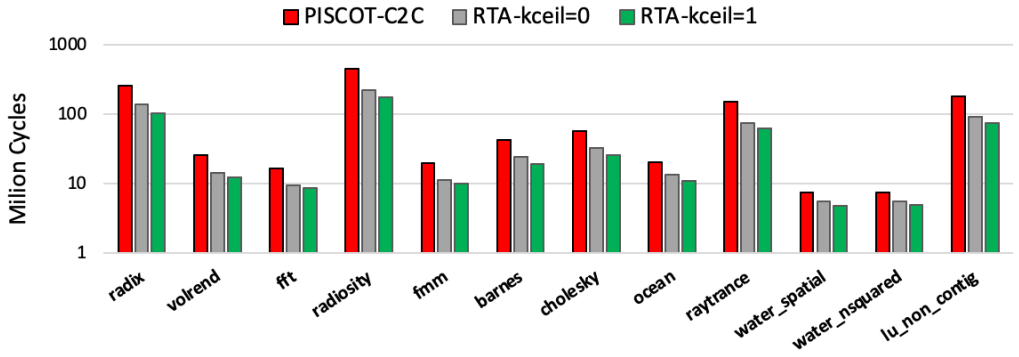
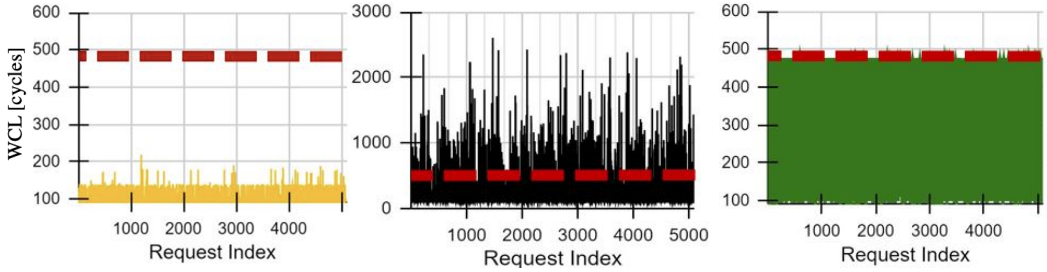


Figure 9 Total observed memory latency of Splash-3. Values in y-axis are in log scale.



(a) RTA. **(b)** HPA. **(c)** DUEPCO.

Figure 10 Observed latencies under different arbitration schemes.

as it resembles the configuration with a higher level of parallelism resembling a more realistic architecture. For example, Intel's architectures utilize a bankized shared cache to hide the shared bank processing time, which is higher than the data transfer on the bus [2].

7.3 Average Performance: SPLASH-3

We next evaluate the average performance of RTA against PISCOT-C2C based on SPLASH-3 benchmarks. Figure 9 shows the cumulative processing latency of all memory requests generated by a quad-core system. Overall, RTA shows an average latency reduction of 1.74× compared to PISCOT-C2C for $k_{ceil} = 0$, and of 2.1× for $k_{ceil} = 1$. This shows that even for realistic benchmarks, bankizing L2 leads to a significant improvement in the performance of the memory subsystem. Processing non-oldest requests leads to further performance improvements, but as previously noted based on Figure 7, this comes at the cost of increased WCL bounds.

7.4 Observed Request Latency

In the last two sets of experiments, we focus on the behavior of our Duetto design, which we call DUEPCO, compared to the RTA and HPA. Note that we configure the relative deadline for each type of request to be equal to its static WCL bound. Figures 10a, 10b, and 10c delineate the observed latency in number of cycles suffered by oldest miss requests of type `REQ:BANK:RESP` generated by a quad-core system. We show request latencies greater than 80 cycles for better visibility and run the experiment with `Synth 3` benchmark (other benchmarks/request types show similar behavior). The RTA latency bound for this setup is

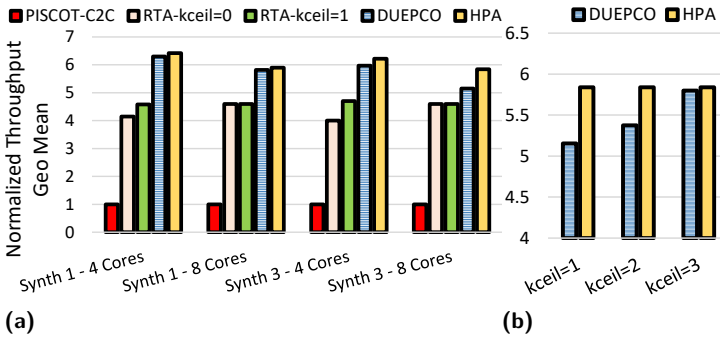


Figure 11 Total throughput of the system.

476 cycles based on the derived WCL analysis in Section 5 which is shown as a red bar in the figures. In HPA, we observe large latency spikes throughout the execution up to 3420 cycles since HPA favors requests from the cores that generate the highest number of requests, are faster, and target the banks that are idle which can starve (theoretically) or delay for a long time (practically) requests targeting busy banks. Figure 10a shows that RTA respects the latency bound for all requests from every core and the latencies are always below the WCL bound. However, there is a gap between the latencies and the static WCL bound since static analysis conducted in Section 5 must assume that the oldest requests of all cores access the same bank at the same time in addition to the non-oldest requests, which is unlikely in practice. Finally, Figure 10c shows that DUEPCO stretches the latency of requests towards the latency bound, but the bound is never violated. This allows the system to continue selecting the HPA as long as possible.

7.5 Average Performance: Synthetic Benchmarks

To measure the average performance of DUEPCO, we use the total throughput of the system based on synthetic benchmarks to stress the system. Figure 11a shows the geometric mean of throughput across all cores for RTA, HPA and DUEPCO normalized to the overall throughput of PISCOT-C2C. The figure represents the results for four different setups: 1) a quad-core system running `Synth 1`; 2) a quad-core system running `Synth 3`; 3) an octa-core system running `Synth 1`; 4) an octa-core system running `Synth 3`. We make the following observations: 1) RTA, HPA, and DUEPCO outperform the single-bank architecture approach deployed in PISCOT-C2C significantly, by up to 6.4×; 2) DUEPCO shows very small slowdown compared to HPA in `synth 1` and `synth 3 - 4 core` (at most 2%); 3) in an octa-core system and `synth 3` benchmark, we observe a slowdown of 11%. Following the discussion in Section 6, since DUEPCO employs RTA with $k_{ceil} = 1$, it has to exclude the invalid states from the HPA by switching to RTA. Recall that `Synth 3` benchmark exposes 20% data-sharing among the cores, and this leads to the case that multiple cores compete to access the same cache line in a particular bank. Therefore, DUEPCO selects the RTA regardless of the WCLator estimation according to the checker logic. However, by increasing the number of allowed requests to the same cache line (k_{ceil}), we expect that DUEPCO selects the HPA more often. As shown in Figure 11b, DUEPCO that employs RTA with $k_{ceil} = 3$ exhibits only 1% slowdown compared to HPA. Notice that relaxing the parameter k_{ceil} forces us to use a higher value for the static WCL bound for each oldest request as shown in Figure 7. Therefore, we do not consider higher values for the parameter.

8 Conclusions

Employing shared memory in multi-core platforms improves programmer productivity and degrades the obstacle to using such platforms in real-time systems. Hardware cache coherence can accommodate such shared memory and extend the advantages of on-chip caching to all system memory. However, extending hardware cache coherence throughout traditional schemes such as coherency protocol modifications to provide predictability hurts the performance of the system. In this work, we demonstrate that by employing the COTS interconnect architecture along with proposing to bankize the on-chip cache, DUEPCO is able to pair a clever global arbitration mechanism with Duetto to significantly improve the performance of the system while providing predictability. Notice that while we propose DUEPCO with simple buses, potentially the same arbitration scheme could be added to other bus architectures such as AXI in ARM platforms. However, the fundamental constraint to consider is that the arbiter must have exclusive visibility into the queues of each requestor.

References

1. Arm cortex-a53 mpcore processor technical reference manual r0p3. https://developer.arm.com/documentation/ddi0500/e/level-1-memory-system/about-the-l1-memory-system. Accessed: 2022-01-23.
2. Intel® 64 and ia-32 architectures optimization reference manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. Accessed: 2021-07-20.
3. ARM. Arm® cortex®-r8 mpcore processor. https://developer.arm.com/documentation/100400/0001/xdc1471434436160, 2019.
4. Matthias Becker, Dakshina Dasari, Borislav Nikolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 14–24. IEEE Computer Society, 2016. doi:10.1109/ECRTS.2016.14.
5. Micaiah Chisholm, Namhoon Kim, Bryan C Ward, Nathan Otterness, James H Anderson, and F Donelson Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 57–68. IEEE, 2016. doi:10.1109/RTSS.2016.015.
6. Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, pages 27:1–27:25, Dagstuhl, Germany, 2019.
7. Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable dram controllers. *ACM Trans. Embed. Comput. Syst.*, 17(2), February 2018. doi:10.1145/3158208.
8. Mohamed Hassan. Heterogeneous mpsocs for mixed-criticality systems: Challenges and opportunities. *IEEE Design & Test*, 35(4):47–55, 2017.
9. 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)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
10. Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246. IEEE, 2017.
11. Mohamed Hassan and Hiren Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, 2016. doi:10.1109/RTAS.2016.7461327.

12 Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A framework for scheduling dram memory accesses for multi-core mixed-time critical systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 307–316. IEEE, 2015.

13 Mohamed Hassan and Rodolfo Pellizzoni. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336, 2018. doi:10.1109/TCAD.2018.2857379.

14 Mohamed Hassan and Rodolfo Pellizzoni. Analysis of memory-contention in heterogeneous cots mpsocs. In *Euromicro Conference on Real-Time Systems*, 2020.

15 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, 2020. doi:10.1109/RTSS49844.2020.00029.

16 Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. Designing predictable cache coherence protocols for multi-core real-time systems. *IEEE Transactions on Computers*, 70(12):2098–2111, 2021. doi:10.1109/TC.2020.3037747.

17 Anirudh Mohan Kaushik and Hiren 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, 2021. doi:10.1109/RTAS52030.2021.00017.

18 Anirudh Mohan Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. Carp: A data communication mechanism for multi-core mixed-criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 419–432, 2019. doi:10.1109/RTSS46320.2019.00044.

19 Manpreet S Khaira. Fast first-come first served arbitration method, November 12 1996. US Patent 5,574,867.

20 Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014. doi:10.1109/RTAS.2014.6925998.

21 Namhoon Kim, Micaiah Chisholm, Nathan Otterness, James H. Anderson, and F. Donelson Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 223–234, 2017. doi:10.1109/RTAS.2017.14.

22 Benjamin Lesage, Isabelle Puaut, and André Seznec. Preti: Partitioned real-time shared cache for mixed-criticality real-time systems. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, pages 171–180, 2012.

23 Reza Mirosanlou, Danlu Guo, Mohamed Hassan, and Rodolfo Pellizzoni. Mcsim: An extensible dram memory controller simulator. *IEEE Computer Architecture Letters*, 19(2):105–109, 2020. doi:10.1109/LCA.2020.3008288.

24 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Drambulism: Balancing performance and predictability through dynamic pipelining. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 82–94, 2020. doi:10.1109/RTAS48715.2020.00-15.

25 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duetto: Latency guarantees at minimal performance cost. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1136–1141, 2021. doi:10.23919/DATE51398.2021.9474062.

26 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. DuoMC: Tight DRAM Latency Bounds with Shared Banks and Near-COTS Performance. In *ACM International Symposium on Memory Systems (MEMSYS)*, pages 1–14, 2021.

27 NXP. Qorlq® t4240, t4160 and t4080 multicore processors, 2018.

28 Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems. *ACM SIGARCH Computer Architecture News*, 37(3), 2009.

29 Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *2008 Real-Time Systems Symposium*, pages 221–231, 2008. doi:10.1109/RTSS.2008.42.

30 Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995.

31 Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 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.

32 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, et al. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

33 Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling cache coherence to expose interference (artifact). In *Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik*, 2019.

34 Nathanal Sensfelder, Julien Brunel, and Claire Pagetti. On how to identify cache coherence: Case of the nxp qoriq t4240. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2020.

35 Ashok Singhal, Bjorn Liencres, Jeff Price, Frederick M Cerauskis, David Broniarczyk, Gerald Cheung, Erik Hagersten, and Nalini Agarwal. Implementing snooping on a split-transaction computer system bus, November 2 1999. US Patent 5,978,874.

36 Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis lectures on computer architecture*, 6(3):1–212, 2011.

37 Nivedita Sritharan, Anirudh Kaushik, Mohamed Hassan, and Hiren Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 433–445, 2019. doi:10.1109/RTSS46320.2019.00045.

38 Calvin K Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 749–753, 1976.

39 Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383, 2013. doi:10.1109/RTSS.2013.44.

40 Zhuanhao Wu, Anirudh Mohan Kaushik, Paulos Tegegn, and Hiren Patel. A hardware platform for exploring predictable cache coherence protocols for real-time multicores. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 92–104, 2021. doi:10.1109/RTAS52030.2021.00016.