Duetto: Latency Guarantees at Minimal Performance Cost

Reza Mirosanlou[†], Mohamed Hassan[*], and Rodolfo Pellizzoni[†]
[†]University of Waterloo, Canada, {rmirosan, rpellizz}@uwaterloo.ca
[*]McMaster University, Canada, mohamed.hassan@mcmaster.ca

Abstract—**The management of shared hardware resources in multi-core platforms has been characterized by a fundamental trade-off: high-performance arbiters typically employed in COTS systems offer no worst-case guarantees, while dedicated real-time controllers provide timing guarantees at the cost of significantly degrading system performance. In this paper, we overcome this trade-off by introducing Duetto, a novel hardware resource management paradigm. Duetto pairs a real-time arbiter with a high-performance arbiter and a latency estimator module. Based on the observation that the resource is rarely overloaded, Duetto executes the high-performance arbiter most of the time, switching to the real-time arbiter only in the rare cases when the latency estimator deems that timing guarantees risk being violated. We demonstrate our approach on the case study of a multi-bank memory. Our evaluation based on cycle-accurate simulations shows that Duetto can provide the same latency guarantees as the real-time arbiter with limited loss of performance compared to the high-performance arbiter.**

I. Introduction

The exigent performance, power, and area required from modern real-time embedded systems motivate the demand to deploy them using multi-core platforms. This led to a challenging task: predictably managing shared hardware resources among cores such that the timing requirements of the system are still honored. Shared buses, I/Os, caches, and main memories are examples of these resources. Existing Commercial-Off-The-Shelf (COTS) arbiters for these resources (e.g. FCFS, priority-based [1], and reordering arbiters [2]) are designed to achieve high average performance, which comes at the cost of losing predictability guarantees.

To address this challenge, researchers have proposed solutions to redesign the arbiters [3], [4] and controllers [5], [6] for these resources such that they achieve predictability by construction. In addition to the commonly used Time Division Multiplexing (TDM) [3] and Round Robin (RR) schemes, researchers also proposed Harmonic RR (HRR) [4], and weighted [7] arbitration schemes. To the opposite extreme of high-performance arbiters, these solutions provide strict timing guarantees on maximum resource access latency at the expense of significantly degrading systems performance. The reason is that advanced optimizations employed by high-performance arbiters typically induce pathological scenarios that lead to extremely high latency in the worst case [8], and must be disabled to provide tight latency bounds. Nonetheless, such pathological scenarios rarely happen in practice.

In this work, we address the fundamental trade-off between average performance and predictability by introducing the Duetto reference model. We show that our reference model can assist a hardware designer in creating an architecture that provides latency guarantees with minimal loss of average performance. Specifically, we propose to pair a simple, predictable Real-Time Arbiter (RTA) with a High-Performance Arbiter (HPA), and to associate latency requirements to each core/requestor in the system. At run-time, a dedicated estimator component monitors the state of the system. Even under heavy load, most of the time, the resource is not overloaded, and no pathological case can be triggered; thus, the HPA is allowed to arbitrate accesses at no risk and maintains high average performance. Only in the rare cases when the resource is overloaded, and the maximum latency of a request could be violated, the architecture switches to the RTA to guarantee the satisfaction of all latency requirements. More in details, **we make the following contributions**. 1) We provide a conceptual description and formalization of the Duetto reference model in Section III. 2) We exemplify the usage of Duetto to design a controller architecture in Section IV for the case study of a system where cores share an interconnect to a shared multi-bank memory. The interconnect deploys a separate read and write bus connecting all cores to all memory banks and memory bus, which resembles the commodity buses existing in modern Systems-on-Chip (such as the ARM's AXI [9]). 3) Section V provides a detailed evaluation of the architecture in the case study by implementing the aforementioned resource and controller architecture in MacSim [10], a multi-core full-system, cycle-accurate simulator. Our results show that the derived architecture can achieve very close performance to a conventional high-performance arbiter while providing tight latency guarantees.

Duetto has been inspired by the Simplex reference model [11]. Simplex prevents safety violations by pairing a high-performance but unreliable component with a low-performance but trustworthy component and using a run-time checker to ensure that the unreliable component does not drive the system to an unsafe state. However, the semantics of the two models are not equivalent: our goal is to control the output of high-speed hardware arbiters at the granularity of a clock cycle, requiring a high level of parallelism in the execution model that cannot be easily supported in the Simplex logic framework.

II. Case Study

We begin by describing the multi-bank memory used in our case study, so that we can use it as a running example throughout the paper. We selected such resource since it allows us to highlight the key steps in the proposed design methodology, especially concerning parallelism in the hardware; at the same time, its behavior is not so complex as to prevent us from fully detailing the latency analysis in the available space. However, we have also validated the reference model on

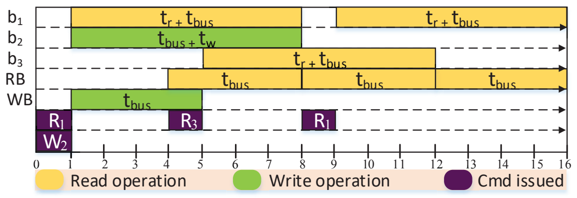

Fig. 1. An example schedule for 4 requests accessing different banks with $t_{bus} = 4$ and $t_r = t_w = 3$.

more complex memory models (specifically, Dynamic Random Access Memories (DRAMs)).

We consider a memory comprising N independent banks $b_1, ..., b_N$. All banks share one Read Bus (RB) and one Write Bus (WB). Each bank can only process either one read operation or one write operation at a time. A read operation requires t_r clock cycles to access the data in the bank, followed by t_{bus} cycles to transfer the data on the read bus. A write operation requires t_{bus} cycles on the write bus, followed by t_w clock cycles to store the data in the bank. A memory controller receives memory requests from cores, arbitrates among such requests, and sends read/write commands to the banks to trigger memory operations. The controller cannot issue a command to a bank b_j if the bank is busy processing a previous operation, or if doing so would create a bus conflict with the operation of a different bank. This implies that the controller can send at most one read and one write command to two different banks at the same time. Sending such commands takes one clock cycle.

An example schedule is shown in Figure 1. We assume that the system is initially idle, then the following requests arrive at the controller: two read requests for bank b_1; one write request for b_2; and one read request for b_3. At time 0, the controller can send both a read command to b_1 and a write command to b_2. The controller must then wait until $t = 4$ to send the read command to b_3, since sending it earlier would cause a read bus conflict. Finally, to issue the second read command to b_1, the controller must wait until the latest of two events: the bank to become idle and for the absence of read bus conflict, which means the command is issued at $t = 8$.

III. Reference Model

We introduce the Duetto reference model. Specifically, we first decompose the system into a set of communicating conceptual components as shown in Figure 2. Then, we detail the execution model and discuss the provided latency guarantees.

A. Requestors and Requests

We assume that the system comprises M distinct *requestors* $\{P_1, ..., P_M\}$, which issue *requests* for service to a shared *resource*. Depending on the resource, requestors could be cores, bus masters/DMA devices, or in general, any active hardware component. Upon being issued, a request is first stored in a *request buffer*; we call this the arrival time of the request. An *arbiter* then mediates access to the resource based on the stored requests. Finally, a request is removed from the request buffer once it completes service at the resource; we call this the finish time of the request. We assume that the requests of each requestor P_i can be totally ordered based on their arrival time, so that we can index them as $r_{i,1}, ..., r_{i,j}, ...$. Each request has

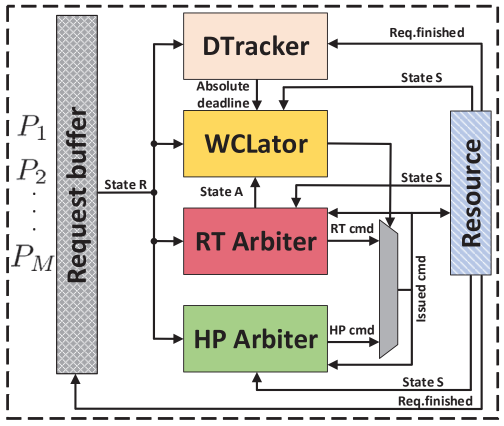

Fig. 2. Duetto reference model.

a type; function $\mathcal{T}(r_{i,j})$ returns the type of $r_{i,j}$. We use R to denote the set of outstanding requests (requests that arrived and have not yet finished) at any one point in time; conceptually, this represents the state of the buffer.

Example: in our case study, requestors are out-of-order cores. The type of a request is either read or write. Each core can issue multiple concurrent requests to any subset of banks. The considered arbiter resemble those in COTS by increasing parallelism through allowing to service requests out of order [2], [12]. For simplicity, we assume that a request finishes after the arbiter sends its read/write command; if the designer wishes to instead consider the time at which the bus transfer is completed, we can add either $t_r + t_{bus}$ (for read) or t_{bus} (for write) to the computed latency.

B. Request Latency and DTracker

The goal of Duetto is to provide worst-case guarantees on the latency of each request. Since a requestor can have multiple outstanding requests, and requests do not need to be serviced in the same order they arrive, it is necessary to precisely define the concept of latency. Following related work [12], [13], we use the following definition:

Definition 1 (Queuing and Processing Latency)**.** *Let* $t^a_{i,j}$ *be the arrival time of request* $r_{i,j}$*, let* $t^f_{i,j}$ *be its finish time, and let* $t^r_{i,j} = t^f_{i,k}$ *be the latest time at which a previously arrived request* $r_{i,k}, k < j$ *of the same requestor finishes (or time 0 if no such request exists). Then the* queuing latency *of* $r_{i,j}$ *is* $\max\left(0, \min(t^f_{i,j}, t^r_{i,j}) - t^a_{i,j}\right)$*, while the* processing latency *is* $\max\left(0, t^f_{i,j} - \max(t^r_{i,j}, t^a_{i,j})\right)$.

Figure 3 shows an example with three requests. Note that since $t^r_{i,j} < t^a_{i,j}$, the queuing latency of $r_{i,j}$ is zero. As in all related work, our methodology bounds the processing latency only. The key reason is that whenever a requestor issues multiple requests and stalls until they complete, the stall time is upper bounded by the sum of the processing delay of the requests. In particular, as discussed in [5], this means that the delay suffered by a real-time task executed on a core accessing a shared resource can be bounded by the sum of the processing delay of the requests issued by the task. For this reason, the

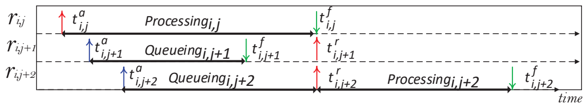

Fig. 3. Queuing and processing latency example. Assume that all previous requests $r_{i,k}$ with $k < j$ finish before $t^a_{i,j}$. We use ↑ for the arrival time $t^a_{i,j}$ of each request, ↓ for its finish time $t^f_{i,j}$, and ↑ for the start of processing: $\max(t^r_{i,j}, t^a_{i,j})$.

processing latency of $r_{i,j+1}$, which is covered by the processing time of $r_{i,j}$, is set to zero.

Based on the above discussion, latency requirements in our reference model are expressed by associating each requestor P_i and each type of request with a *relative deadline* $D_i(\mathcal{T}(r_{i,j}))$, which represents the maximum allowable processing latency for each request of that requestor and type. Consequently, the finishing time of every request $r_{i,j}$ must be no later than its *absolute deadline* $d_{i,j} = \max(t^r_{i,j}, t^a_{i,j}) + D_i(\mathcal{T}(r_{i,j}))$. The *Deadline Tracker* (DTracker in Figure 2) component is responsible for maintaining such information. Note that it suffices to maintain a single absolute deadline for each requestor P_i, associated with its oldest outstanding request $r_{i,j}$: this is because subsequent requests of P_i cannot have an absolute deadline earlier than the finish time of $r_{i,j}$. To simplify exposition, in the rest of the paper we will thus use the term "oldest request" to denote any request that is the oldest for its requestor (rather than the oldest among all requestors).

C. Commands and Resource Interface

An arbiter controls the resource to service requests by issuing one *command* every clock cycle. Like requests, every command is characterized by a type. A "no-operation" NOP command is used whenever the resource is idle.

Example: based on the discussion in Section II, the resource accepts four types of commands: NOP, RD, WR, and RD/WR. Note that while in this case a request is serviced by a single command, depending on the resource, additional commands might be required. For example, a DRAM request might require a PRE, ACT and CAS commands [14].

We assume that the command semantic is defined by an automata, which we call the *resource interface*. Essentially, the interface defines the "contract" between the resource and the arbiter; in this sense, it does not need to model the low-level internal state of the resource, but rather only those details that are relevant in terms of the behavior of the commands. For many resource types, the interface is defined by a standard, e.g. JEDEC for DRAM [14]. Let S to denote the current state of the resource interface; we say that a command is *valid* in S if the resource can accept that command. We further say that a command is *legal* if it is valid and satisfies an outstanding request in R, and use $\mathcal{L}(S, R)$ to denote the set of legal commands. Note that a command might be valid but not legal: for example, a requestor might issue a read request to a memory resource, and an erroneous arbiter might then generate a valid but incorrect write command for that request.

Example: the behavior of the resource interface can be defined using $N + 2$ timers: c^r, c^w for the read and write bus, and c^b_j for each bank b_j. c^r (c^w) is set to t_{bus} every time a RD (WR) command or RD/WR command is issued. Whenever a command is sent to bank b_j, c^b_j is set to either $t_r + t_{bus}$ (for a read operation) or $t_{bus} + t_w$ (for a write). A command is valid only if all relevant bank and bus timers are zero. We say that a request is ready if it can be serviced by issuing a legal command in the current clock cycle.

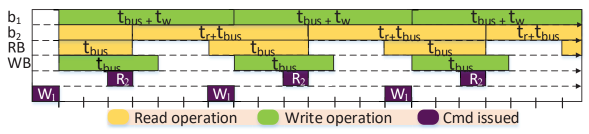

Fig. 4. Unbounded latency. A read request to b_1 is never ready, since the read bus is occupied whenever b_1 becomes idle.

D. High-Performance and Real-Time Arbiter

The reference model comprises two arbiters: a *High-Performance Arbiter* (HPA), which we assume to be optimized for maximum average performance, and a *Real-Time Arbiter* (RTA), optimized for tight latency bounds. Since this paper focuses on timing requirements, and not functional verification, for simplicity, we will assume that the HPA is *correct*, in the sense that it always issues legal commands [1]. However, we do not make any further assumptions on the way that requests and their corresponding commands are scheduled by the HPA. This ensures that the latency guarantees provided by our methodology are completely independent of the HPA. On the other hand, the behavior and internal state of the RTA, which we denote as A, must be explicitly modeled.

Example: for our evaluation in Section V, we employ a First-Ready, First-Come-First-Serve (FR-FCFS) arbiter as the HPA. At each clock cycle, this arbiter selects among ready requests, and gives priority to requests based on their arrival time. We choose this arbiter because, as shown in literature [2], it tends to maximize performance in terms of the overall Instructions Per Cycle (IPC) of the system by favoring applications with high IPC that can issue multiple concurrent memory requests. However, note that this arbiter does not provide any latency guarantee. In particular, as shown in Figure 4, we can construct a pattern of memory accesses where a read request to bank b_1 can be stalled for any amount of time by a sequence of write requests to b_1 and read requests to other banks. We discuss the design of the RTA in Section IV.

E. Execution Model and Latency Guarantees

Finally, we discuss the execution model and how latency requirements are guaranteed by the *Worst-Case Latency estimator* (WCLator) component, which is the core of our reference model. As Figure 2 illustrates, the two arbiters operate independently and in parallel. Every clock cycle, each arbiter selects one command (possibly NOP) based on its internal state, as well as the states S, R of the resource interface and request buffer. In parallel, the WCLator selects between the two arbiters; the command of the selected arbiter is then issued to the resource through the multiplexer in the figure. Since the command issued in a clock cycle might be different from the one selected by an arbiter, we require that each arbiter updates

[1]If the correctness of the HPA cannot be verified, the methodology can be extended with an additional checker module that checks the legality of the commands based on the states S, R of the resource interface and request buffer.

its internal state based on the actual issued command, rather than the one it selects. It is essential to note that because we are targeting high-speed hardware implementation, we assume that **the WCLator must make its decision without knowing which commands are selected by the two arbiters**: otherwise, the WCLator logic would have to be placed in series with the arbiters, which could greatly slow down the clock speed.

To decide between the arbiters, at each clock cycle and for each requestor P_i with one or more outstanding requests, the WCLator computes an upper bound to the finish time $t^f_{i,j}$ of its oldest request $r_{i,j}$, under the assumption that **any legal command can be sent in the current cycle, while the RTA is selected in all future cycles.** If for each such requestor, the computed finish time is less than or equal to the deadline $d_{i,j}$, then the WCLator selects the HPA. Otherwise, it selects the RTA. The intuition for this decision is that if the computed finish time is no larger than the deadline, then it is safe to continue with the (legal) command that is selected by the HPA. Note that the WCLator's estimation can be based on the current states S, R and A of the resource interface, request buffer and RTA. We next formally prove that the system meets all deadlines, as long as the latency requirements are not set to a smaller value than the latency guarantees provided by the RTA.

Definition 2 (Static Worst-Case Latency (WCL) Bound)**.** *For every requestor P_i and request type, let $\Delta_i\big(\mathcal{T}(r_{i,j})\big)$ to be an upper bound to the processing latency of $r_{i,j}$ assuming any possible state of the resource interface S, request buffer R and RTA A at time $\max(t^r_{i,j}, t^a_{i,j})$, and that the WCLator always selects the RTA from $\max(t^r_{i,j}, t^a_{i,j})$ onward.*

Theorem 3. *No request misses its deadline if for all requestors and request types it holds: $D_i\big(\mathcal{T}(r_{i,j})\big) \geq \Delta_i\big(\mathcal{T}(r_{i,j})\big)$.*

Proof. By contradiction, assume there exists a request $r_{i,j}$ that misses its deadline at $d_{i,j}$. Since $d_{i,j} = \max(t^r_{i,j}, t^a_{i,j}) + D_i\big(\mathcal{T}(r_{i,j})\big)$ and no older request of P_i can finish after $t^r_{i,j}$, it follows that $r_{i,j}$ is the oldest request of P_i in the interval $[\max(t^r_{i,j}, t^a_{i,j}), d_{i,j}]$. We consider two cases: (1) the WCLator always sends commands of RTA in $[\max(t^r_{i,j}, t^a_{i,j}), d_{i,j} - 1]$; (2) or the WCLator sends at least one command of HPA during such interval; in which case let t be the latest time at which an HPA command is sent.

Case (1): by Definition 2 and since the RTA is always selected, $r_{i,j}$ must finish by $\max(t^r_{i,j}, t^a_{i,j}) + \Delta_i\big(\mathcal{T}(r_{i,j})\big) \leq \max\big(t^r_{i,j}, t^a_{i,j} + D_i\big(\mathcal{T}(r_{i,j})\big)\big) = d_{i,j}$; a contradiction.

Case (2): since the WCLator selects the HPA at time t, while the RTA is selected for all following cycles until $d_{i,j}$, and because by assumption the HPA is correct, it follows that $t^f_{i,j} \leq d_{i,j}$, again a contradiction. □

The key intuition behind Duetto is that using run-time information on the state of the system typically allows the WCLator estimation to be much tighter than any possible static WCL bound. Hence, unless the system becomes fully loaded, the WCLator can keep selecting the HPA and avoid loss of average performance.

IV. Architecture Design

We next show how to employ the reference model to design a concrete architecture. We consider a use-case where the resource, HPA and request buffer have already been designed/implemented, and the designer wishes to add support for latency requirements using Duetto. The detailed DTracker design depends on the request buffer, but we argue that it is generally straightforward [2]. Therefore, we focus on the design of the RTA and WCLator. We do not claim that an automated process is possible; however, we believe that the design can proceed through a sequence of four conceptual steps, which we illustrate based on our case study.

Step (A): RTA design. We design a dynamic RR arbiter, which provides the same latency guarantees to every requestor without unduly limiting bank parallelism. A requestor is removed from the RR queue when its oldest request finishes, and enqueued at the back of the RR queue if it has any outstanding request. For a requestor P_i, we use hp_i to denote the set of higher priority requestors, i.e. the requestors that are ahead of P_i in the RR queue. We say that a request of P_i to bank b_k is blocked if there is a non-ready oldest request of a requestor in hp_i that also targets b_k. The RTA arbitrates between non-blocked ready requests based on a two-level arbitration scheme: in the first level, it gives priority to oldest requests over non-oldest requests; at the second level, it uses the RR order of requestors. This means that if, for example, the oldest request of the highest priority requestor is non-ready because its bank is busy, the controller can still service a lower priority request to another bank. However, if the highest priority request is non-ready because its data bus is busy, the controller cannot service a lower priority request of the opposite type (read to write or vice versa) to the same bank, since this could result in the pattern in Figure 4.

Step (B): Dynamic RTA Latency Analysis. We compute an upper bound to the remaining latency (i.e., the time to finish) of the oldest request $r_{i,j}$ of P_i assuming that the RTA is always selected. We encode the states S, R and A into a small set of *analysis parameters* used to derive latency equations. Due to space limitations, we only consider the case of a read request, but the write case is similar. Let b_k be the bank targeted by $r_{i,j}$. We use $k^{bank,r}, k^{bank,w}$ to denote the number of read/write oldest requests of requestors in hp_i that target b_k, and $k^{bus,r}, k^{bus,w}$ to denote the number of oldest requests of requestors in hp_i that target another bank; note that such parameters can be easily derived at run-time based on the state of the request buffer R and RR queue in the RTA (A).

Theorem 4. *If the oldest request of P_i at time t is read request $r_{i,j}$ targeting b_k, and the RTA is always selected from t onward, its remaining latency $t^f_{i,j} - t$ is bounded by:*

$$\text{if } k^{bank,w} = 0: \quad c^{init,r} + k^{bank,r} \cdot (t_r + 2 \cdot t_{bus} - 1) + k^{bus,r} \cdot t_{bus} + 1, \tag{1}$$

$$\text{if } k^{bank,w} > 0: \quad c^{init,rw} + k^{bank,r} \cdot (t_r + 2 \cdot t_{bus} - 1) + k^{bank,w} \cdot (t_w + 2 \cdot t_{bus} - 1) + (k^{bus,r} + k^{bus,w}) \cdot t_{bus} + 1, \tag{2}$$

[2] To simplify implementation, it is preferable to store the number of cycles remaining until the absolute deadline, rather than the absolute deadline itself.

where:

$$\text{if } c^r \geq c^b_k: \qquad c^{init,r} = c^r, \tag{3}$$

$$\text{if } c^r < c^b_k: \qquad c^{init,r} = c^b_k + t_{bus} - 1, \tag{4}$$

$$\text{if } c^r \geq c^b_k \wedge c^w \geq c^b_k: \qquad c^{init,rw} = \max(c^r, c^w), \tag{5}$$

$$\text{if } c^r < c^b_k \vee c^w < c^b_k: \qquad c^{init,rw} = c^b_k + t_{bus} - 1. \tag{6}$$

Proof. Since oldest ready requests are arbitrated in RR order, and requests to b_k are blocked if there is a higher priority non-ready request, it follows that the RTA will service a sequence of exactly $k^{bank,r} + k^{bank,w}$ requests to b_k followed by $r_{i,j}$ itself. Furthermore, after a request in the sequence becomes ready and non-blocked, it can still be delayed by higher priority requests targeting the same bus but a different bank; if $k^{bank,w} > 0$, then conflicts can happen over both the read and write bus; hence we consider $k^{bus,r} + k^{bus,w}$ conflicting requests, otherwise ($k^{bank,w} = 0$), we only consider the $k^{bus,r}$ read requests. Then, the remaining latency of $r_{i,j}$ can be obtained by summing: (1) the time until the first request in the sequence to b_k first becomes ready; we call this either $c^{init,r}$ (if $k^{bank,w} = 0$) or $c^{init,rw}$ (if $k^{bank,w} > 0$). (2) The latency between issuing the command for one request in the sequence, and the time the next request in the sequence becomes ready. Each request in the sequence occupies b_k for either $t_r + t_{bus}$ (read) or $t_{bus} + t_w$ (write) cycles; furthermore, a lower priority request could be serviced the cycle before the bank becomes idle, adding an extra $t_{bus} - 1$ cycles of delay. Hence, the overall latency over the sequence is equal to: $k^{bank,r} \cdot (t_r + 2 \cdot t_{bus} - 1) + k^{bank,w} \cdot (t_w + 2 \cdot t_{bus} - 1)$. (3) The delay of higher priority requests targeting a different bank; as argued, this is $(k^{bus,r} + k^{bus,w}) \cdot t_{bus}$ if $k^{bank,w} > 0$, and $k^{bus,r} \cdot t_{bus}$ otherwise. (4) One clock cycle to issue a RD or RD/WR command to service $r_{i,j}$. Summing the four terms yields Equations 1, 2.

Finally, we consider $c^{init,r}, c^{init,rw}$. As shown in Equations 3-6, the two cases differ only in which bus timers we need to consider, based on the type of requests in the sequence to b_k. If at time t, the relevant bus timer(s) is larger or equal than the bank timer c^b_k, then the first request in the sequence will become ready when the bus timer(s) expire. Otherwise, it is again possible for a lower priority request to be serviced one cycle before b_k becomes idle, resulting in an initial delay of $c^b_k + t_{bus} - 1$. This concludes the proof. $\square$

Step (C): Static WCL Bound. The static WCL bound $\Delta_i\big(\mathcal{T}(r_{i,j})\big)$ is obtained by maximizing the remaining latency in Equations 1-6 over all possible values of the parameters. Specifically, we set c^b_k to its maximum value $\max(t_r, t_w) + t_{bus} - 1$ (a request was issued to b_k the previous cycle), and set $k^{bank,r} + k^{bank,w} = M - 1$, $k^{bus,r} = k^{bus,w} = 0$ (requests to bank b_k generate larger delay, and there can only be one oldest request for each of the $M - 1$ other requestors), yielding:

$$\Delta_i(\text{read}) = M \cdot \big(\max(t_r, t_w) + 2 \cdot t_{bus} - 1\big). \tag{7}$$

Repeating the analysis for a write request yields the same bound. Hence, in our example, all request types have the same static WCL bound; and since we treat all requestors equally, the bound does not depend on the requestor either. However, more complex arbitration schemes could differentiate between request types [8], [15] or requestors [3], [7] based on criticalities. The obtained bound is predictable, in the sense that it is linear in the number of requestors; each requestor contributes a latency term $\max(t_r, t_w) + t_{bus}$, which represents the worst-case intra-bank time, plus a blocking term $t_{bus} - 1$, which represents the price for allowing inter-bank parallelism.

Step (D): WCLator design. Consider again the oldest request $r_{i,j}$ of P_i targeting bank b_k at time t. To estimate its finish time $t^f_{i,j}$, we enumerate a set of cases based on which request(s) (at most one read and one write) could be serviced by a legal command issued at time t. For each case, we compute a bound to the remaining latency of $r_{i,j}$, so that we can obtain $t^f_{i,j}$ by summing the remaining latency with t.

- **(1)** If $r_{i,j}$ is serviced, then its remaining latency is 1 cycle.
- **(2)** Otherwise, the remaining latency is computed by using Equations 1-6 but with a modified value of some parameters, as detailed in the sub-cases below, to account for the command sent at t; this is possible because by definition the WCLator computes $t^f_{i,j}$ assuming that the RTA is selected from $t + 1$ onward.
- **(2.1)** If a request to b_k is serviced, set $c^b_k = t_r + t_{bus}$ or $c^b_k = t_{bus} + t_w$, depending on the type of request; and subtract one from $k^{bank,r}$ or $k^{bank,w}$ if it is the oldest request of a requestor in hp_i.
- **(2.2)** If a request to another bank is serviced, set either $c^r = t_{bus}$ or $c^w = t_{bus}$, depending on the type of request; and subtract one from $k^{bus,r}$ or $k^{bus,w}$ if it is the oldest request of a requestor in hp_i.
- **(2.3)** If neither sub-case (2.1) or (2.2) apply, then a NOP is issued. In this case, no change is needed if the value of $c^{init,r}$ in Equation 1, or the value of $c^{init,rw}$ in Equation 2, is greater than zero; otherwise, add one to the latency computed by the equation to account for the cycle wasted by issuing the NOP.

At run-time and for each requestor P_i, the WCLator then uses the set of legal commands $\mathcal{L}(S, R)$ to determine which cases can apply; and takes $t^f_{i,j}$ as the maximum over all such cases. Here, sub-case (2.1) can be excluded if $c^b_k > 0$, or there is no request targeting b_k apart from $r_{i,j}$ in the request buffer; combining such information with the state of the RR queue in the RTA allows the WCLator to also determine whether $k^{bank,r}$ or $k^{bank,w}$ should be decreased or not (note that if possible, the latter case must be considered since it leads to higher latency). Similar considerations hold for sub-case (2.2). The presented design is well-suited for hardware implementation, because each case for every requestor can be computed in parallel [3], and the resulting $t^f_{i,j}$ compared against $d_{i,j}$ to determine if the HPA can be selected. Hence, the complexity of the implementation depends on the latency equations. As shown by Equations 1-2, we argue that most analysis for predictable arbiters [4], [7], [8], [12], [15] yield equations that involve adding terms, where each term depends on an analysis parameter. This can be computed efficiently in hardware by using one look-up table for each term, and then cascading the results through a sequence of adders.

V. Evaluation

We use MacSim [10], an x86 multi-processor architectural simulator, to model the requestors in our evaluation. We incorporate eight 8-wide superscalars (i.e. it can process multiple

[3]To reduce area, an optimized implementation can merge cases that have similar bounds and remove those with provably smaller latency.

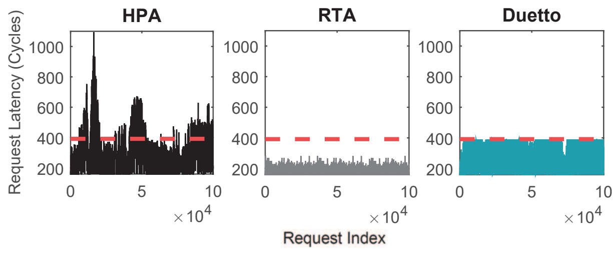

Fig. 5. Request latency of HPA, RTA, and Duetto.

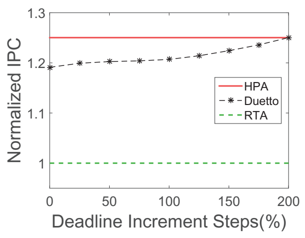

Fig. 6. IPC ($t_r = t_w = 0$).

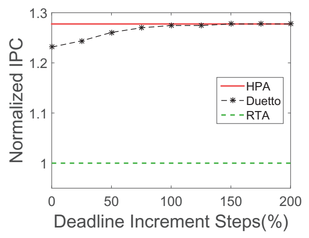

Fig. 7. IPC ($t_r = t_w = 30$).

instructions per cycle) cores clocked at 1GHz and implement the case study in the open-source MCsim memory controller simulator [16]. All cores share a bus connected to the multi-banked memory, as explained in the case-study throughout the paper. We employ two types of synthetic benchmarks: latency-sensitive and bandwidth-oriented from IsolBench [17]. We run the non memory-intensive benchmark on the foreground core and memory-intensive benchmarks on the background cores. Memory requests are interleaved among all banks at the granularity of a cache line (64 bytes).

Figure 5 delineates the processing latency of each request under HPA, RTA and Duetto when all requestors contend for access to $N = 8$ banks with a bus time $t_{bus} = 10$ cycles and processing time $t_r = t_w = 30$. For visualization reasons, the figure only incorporates requests with latency longer than 150 cycles. The red line represents the static WCL bound. For HPA, we observe large latency spikes throughout the execution. This is because HPA prioritizes requests that target a ready bank, which can starve (theoretically) or delay for a long time (practically) requests targeting busy banks. RTA guarantees the latency bound for all requests as expected. However, none of the requests come close to the static WCL bound (392 cycles): this is because the static analysis must assume that all requestors access the same bank at the same time, which is unlikely in practice. Finally, for Duetto we used the minimum possible relative deadline $D_i\big(\mathcal{T}(r_{i,j})\big) = \Delta_i\big(\mathcal{T}(r_{i,j})\big)$ for each requestor. Duetto stretches the latency of requests towards the relative deadline (red line), allowing it to keep selecting the HPA as long as possible.

We use the aggregate IPC of the workload over 8 cores as a measure of performance. Figures 6 and 7 show the IPC of RTA, HPA and Duetto normalized by the IPC of RTA, when setting either $t_r = t_w = 0$ or $t_r = t_w = 30$. Notice that, $t_r = t_w = 0$ implies the requestors only compete to access the read/write buses, i.e. there is no bank parallelism. For Duetto we first set all deadlines to the minimum possible value, and then progressively increase them up to $3 \times \Delta_i$ (200% increase). From the figures, the performance of Duetto is already close to HPA with the strictest latency requirements, and relaxing such requirements further increases its performance until it matches the HPA. Furthermore, bank parallelism improves the relative performance of Duetto, as it increases the difference between the static WCL bound and the dynamic bound. We have also experimented by changing the number of banks and the way that requests are interleaved, but due to space limitations we omit such results as they show little variations in terms of the relative performance between HPA, RTA, and Duetto.

VI. Conclusion and Future Work

We introduced the Duetto reference model, a novel paradigm for shared hardware resource management in real-time embedded systems. By pairing a high-performance COTS arbiter with a predictable real-time arbiter and dynamically switching between the two at run-time, Duetto is able to overcome the traditional trade-off between average case performance and predictability. We believe that Duetto can be demonstrated on a broad spectrum of resources, including DRAM and bus/cache.

References

[1] W. Bain Jr and S. Ahuja, "Performance analysis of high-speed digital buses for multiprocessing systems," in *ACM Annual symposium on Computer Architecture (ISCA)*, 1981.

[2] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *ACM SIGARCH Computer Architecture News*, 2000.

[3] F. Hebbache, M. Jan, F. Brandner, and L. Pautet, "Shedding the shackles of time-division multiplexing," in *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[4] M.-K. Yoon, J.-E. Kim, and L. Sha, "Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[5] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of dram latency in multi-requestor systems," in *2013 IEEE 34th Real-Time Systems Symposium (RTSS)*, 2013.

[6] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Drambulism: Balancing performance and predictability through dynamic pipelining," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[7] M. Hassan and H. Patel, "Criticality-and requirement-aware bus arbitration for multi-core mixed criticality systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[8] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[9] ARM, "Amba axi protocol specification v2. 0," *ARM Holdings*, 2010.

[10] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.

[11] T. L. Crenshaw, E. Gunter, C. L. Robinson, L. Sha, and P. Kumar, "The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures," in *IEEE International Real-Time Systems Symposium (RTSS)*, 2007.

[12] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[13] R. Mancuso, R. Pellizzoni, N. Tokcan, and M. Caccamo, "Wcet derivation under single core equivalence with explicit memory budget assignment," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.

[14] D. S. Standard, "Jedec jesd79-3," 2007.

[15] H. Yun, R. Pellizzon, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[16] R. Mirosanlou, D. Guo, M. Hassan, and R. Pellizzoni, "Mcsim: An extensible dram memory controller simulator," *IEEE Computer Architecture Letters (CAL)*, 2020.

[17] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.