A Mixed Critical Memory Controller Using Bank Privatization and Fixed Priority Scheduling

Leonardo Ecco, Sebastian Tobuschat, Selma Saidi and Rolf Ernst
Institute of Computer and Network Engineering
TU Braunschweig, Germany
{ecco,tobuschat,saidi,ernst}@ida.ing.tu-bs.de

***Abstract*—Mixed critical platforms are those in which applications that have different criticalities, i.e. different levels of importance for system safety, coexist and share resources. Such platforms require a memory controller capable of providing sufficient timing independence for critical applications. Existing real-time memory controllers, however, either do not support mixed criticality or still allow a certain degree of interference between applications. The former issue leads to overly constrained, and hence more expensive, systems. The latter issue forces designers to assume the worst case latency for every individual memory transaction, which can be very conservative when applied to determine the worst-case execution time (WCET) of a task that performs many memory requests. In this paper, we address both issues. The main contributions are: (1) A memory controller that allows a predetermined number of critical and non-critical applications to coexist, while providing an interference-free memory for the former. To achieve that, we treat the memory as a set of independent virtual devices (VDs). Therefore, we also provide (2) a partitioning strategy to properly map mixed critical workloads to VDs. We present experiments that show that our controller allows DRAM sharing with no interference on critical applications and minimal performance overhead on non-critical ones (they perform on average only 15% slower in the shared environment).**

I. Introduction

The trend in designing safety critical real-time embedded systems is moving from distributed setups, where each function is implemented on its own hardware unit, towards an *integrated* environment, where different functionalities are implemented in the same silicon chip and share resources [1]. If the functionalities have different levels of importance for the system safety, i.e. if they have different *criticalities*, the system is said to be *mixed critical*. For instance, the mixed critical system presented in Figure 1 has four cores, half of them running critical applications (depicted in white) and half running non-critical applications (depicted in gray).

All cores share the interconnect fabric and a DRAM, which helps to reduce implementation costs and power consumption of the final product. However, the competition for resources causes applications to interfere with each other's performance. Therefore, mechanisms to ensure that critical applications are temporally isolated must be employed [2].

This work was supported within the scope of the ARAMiS project from the German Federal Ministry for Education and Research with the funding ID 01|S11035. The responsibility for the content remains with the authors.

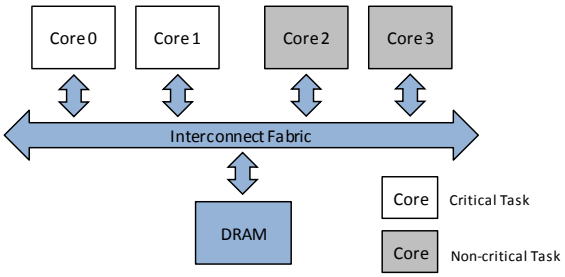


Fig. 1: A mixed critical system with shared resources.

A particular challenge is observed with dynamic RAMs (DRAMs). In these devices, the latency of an access depends heavily on the history of previous accesses. In a scenario where multiple applications share the memory, the access history is the *interleaving* of accesses performed by all requestors. (In this paper, we use the terms *requestor* and *application* interchangeably). In order to provide latency and throughput guarantees to all of them, several predictable hardware memory controllers [3]–[7] have been proposed.

Nevertheless, as we detail in Section III, they either do not support mixed criticality, i.e. do not acknowledge that applications can have different levels of importance to system safety, or they allow a certain degree of interference between them. The former issue can lead to overly constrained, and therefore more expensive systems, because latency and throughput guarantees are given to all applications, even the ones that do not require it. The latter issue forces designers to assume the worst case latency for every individual memory transaction, which can be very conservative when applied to determine the worst-case execution time (WCET) of a task that performs many memory requests [8].

In this paper, we propose the *Mixed Critical Memory Controller* (MCMC), a controller that addresses both issues. We make a clear distinction between critical applications, for whom we provide an interference-free memory (with throughput and latency guarantees), and non-critical applications, for whom we do not give service guarantees. Our approach is an extension of the bank privatization scheme presented in [3]. Such scheme abstracts the complex DRAM access protocol and display the memory as a set of independent virtual devices

(VDs), all capable of providing fixed bandwidth.

We propose to share each VD between one critical and a predetermined number of non-critical applications. We make critical applications completely unaware of the existence of interfering requestors through fixed-priority arbitration. We also propose a partitioning strategy, that can guide system designers through the task of mapping a mixed critical workload to the VDs.

To test our approach, we use a mixed critical workload based on applications from MiBench [9] and EEMBC [10]. We collect memory access traces from the applications and feed them to a cycle accurate SystemC model of our memory controller. The results show that our controller allows DRAM sharing with no interference on critical applications and minimal performance overhead on non-critical ones (they perform on average only 15% slower in the shared environment).

The rest of this paper is structured as follows. Section II covers the theoretical foundation required to understand this work. Section III discusses the limitations of the existing approaches. Section IV describes our solution. Finally, experiments are presented in Section V, followed by the conclusion in Section VI.

II. Background

We firstly discuss the concepts of performance monotonicity, timing composability and timing predictability. They are important to understand the limitations of existing approaches. Then, we present the details of the DRAM access protocol. Finally, we cover the bank privatization technique.

A. Performance monotonicity

A processor is said to be performance monotonic [11] if local reductions in its execution time cannot lead to longer overall execution times. This is not the case, for instance, for processors with out-of-order execution pipelines. These processors suffer from timing anomalies [12] and might actually have a longer overall execution time in case their memory requests are served faster than expected.

B. Timing Composability vs. Timing Predictability

Predictability is the ability to provide an upper bound on the timing properties of a system. Composability is the ability to integrate components while preserving their temporal properties [13]. Although composability implies predictability, the opposite is not true. A round-robin arbiter, for instance, is predictable, but not composable. A TDM (Time Division Multiplexing) arbiter, on the other hand, is predictable and composable.

A predictable shared resource does not completely eliminate the interference that requestors can exert on each other, only provides an upper bound on it. In other words, it still allows a small degree of interference between them. This poses two problems. Firstly, it forces designers to assume the worst case latency for every individual memory transaction, which can be very conservative when applied to determine the worst-case execution time (WCET) of a task that performs many memory requests [8]. Secondly, it prevents requestors from being verified in isolation, in case they do not exhibit performance monotonicity.

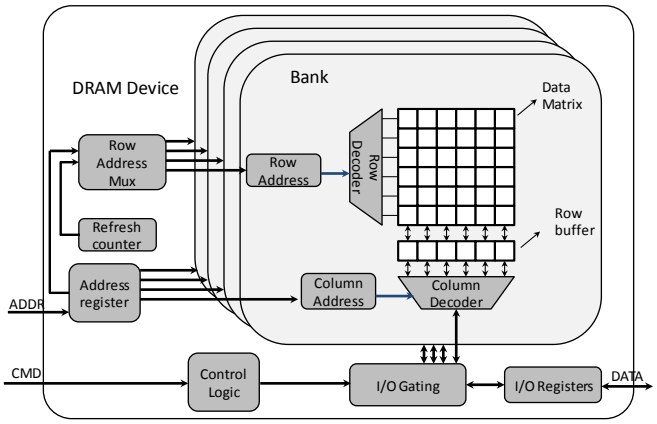


Fig. 2: Generic 4-banks DRAM device structure.

This is not the case for composable shared resources, which eliminate all interference between requestors. Requestors sharing a composable resource are completely unaware of the existence of each other or, in other words, are completely isolated. Hence, composable resources allow requestors to be verified in isolation, even when they do not exhibit performance monotonicity.

However, providing timing composability for all requestors in a system is expensive. To reach a compromise, our memory controller provides it only for critical requestors, i.e. those who are important to the safety of a system. The non-critical requestors, i.e. those who do not play a role in the system safety, do not receive guarantees.

C. Dynamic Random Access Memory (DRAM) Operation

DRAM memories have a three dimensional geometry. As depicted in Figure 2, a DRAM device is composed of banks, rows and columns [14]. The number of banks in each DRAM device varies between different DDR generations, e.g. DDR2 and DDR3 have 4 and 8 banks, respectively. Each bank has a data matrix composed of several rows. The rows contain word-sized columns, and the columns contain either 4, 8 or 16 bits. The column size matches the number of data bus pins from the device (without loss of generality, we consider the column size to be 16 bits throughout the rest of this paper).

A memory controller operates a DRAM device by issuing five different commands: *activate*, *read*, *write*, *precharge* and *refresh*. The activate (ACT) command loads a row into the bank's row buffer, a process known as opening a row. The read (RD) or write (WR) commands are used to retrieve or forward columns from or to an opened row. The acronym CAS (Column Access) refers to both RD and WR commands. The number of columns accessed by a CAS is determined by the burst length (BL) parameter. For DDR2, the burst length must be configured in the mode register inside the device and can be either 4 or 8 [15]. For DDR3, only a burst of 8 is supported [16]. A precharge (PRE) command is used to write the row

TABLE I: Timing constraints for DDR3-1333H and DDR2-400.

JEDEC Specification (cycles)			
Constraint	Description	DDR3-1333H	DDR2-400
t_{RCD}	ACT to RD/WR delay	9	3
t_{RP}	PRE to ACT delay	9	3
t_{RC}	ACT to ACT (same bank) delay	33	11
t_{RAS}	ACT to PRE delay	24	8
t_{BURST}	data bus transfer	4	2 or 4 [1]
t_{CWD}	WR to data bus transfer delay	7	2
t_{CAS}	RD to data bus transfer delay	8	3
t_{RTP}	RD to PRE delay	5	2
t_{WR}	End of a WR operation to PRE delay	10	3
t_{RRD}	ACT to ACT delay (different banks)	4	2
t_{FAW}	Four ACT window (different banks)	20	-
t_{RTW}	RD to WR delay (any bank)	7	4 or 6 [2]
t_{WTR}	End of WR to RD delay (any bank)	5	2

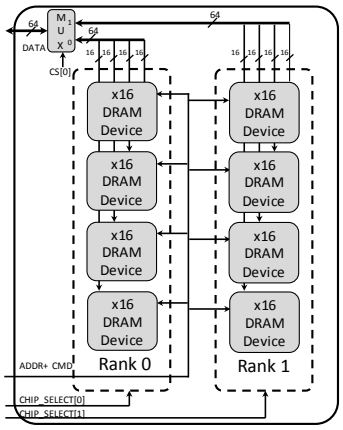


Fig. 3: DRAM module structure.

buffer back to the corresponding bank's data matrix. Lastly, the refresh (REF) command must be executed regularly to prevent the capacitors that are used to store data from discharging.

DRAM also supports delayed CAS commands (known as posted-CAS). Unlike other commands, a posted-CAS is held by the DRAM device and issued after a predefined additive latency. Both CAS and posted-CAS commands can be issued with the Auto Precharge Flag, which automatically precharges the accessed row after transferring the data, preparing it for a next access.

There are several timing constraints that dictate how many cycles apart consecutive commands must be. Table I enumerates such constraints for a DDR3-1333H and a DDR2-400 devices. For instance, t_{rtw} specifies how many cycles apart a write command should be from a read command. The constraints are measured in data bus clock cycles. Notice how the numbers are higher for the DDR3 device. The reason is that the constraints measured in nanoseconds are similar between different generations. Therefore, the higher the frequency of a device, the bigger its timing constraints measured in cycles.

Except for t_{RRD}, t_{FAW}, t_{RTW} and t_{WTR}, the constraints only apply to the bank level. This allows commands to be pipelined across different banks, which in turn increases the utilization of the data bus. For instance, the controller can issue an ACT command to bank 0, and before the corresponding row is opened, precharge the row buffer from bank 1 (preparing it for a new access). This technique is known as bank interleaving.

A controller can go even further and interleave commands to different ranks. A rank is a group of devices that share a clock, command bus and a chip-select signal. It is seen by the memory controller as a single DRAM device with a bigger number of data bus pins (wider data bus). One ore more ranks can be grouped on a printed circuit board to form a module. The module depicted in Figure 3, for instance, has 2 ranks, each with 4 DRAM devices. A multiplexer controlled by the chip select signal then selects between the data bus from rank 0 or 1.

The t_{RRD}, t_{FAW}, t_{RTW} and t_{WTR} constraints only apply to the rank level. Interleaving requests to different ranks hides the latency imposed by these constraints and allows the controller to increase further the utilization of the data bus.

D. Bank Privatization

The bank privatization scheme was introduced in [3]. It proposes a memory controller back-end that displays the memory as a set of independent virtual devices (VDs) capable of providing fixed bandwidth. A VD in this approach is a group of banks from the same rank. The virtual devices are accessed in a rank interleaved fashion, hiding the latency imposed by timing constraints. To illustrate the technique, we consider a dual rank DDR3-1333H memory module (a DDR2-400 module was used in [3]).

The chosen memory module has 2 ranks, each with 8 banks. Only a burst length of 8 is supported and, therefore, each CAS operation occupies the data bus for $BL/2 = 4$ cycles. The data bus runs at a frequency of 666.67 MHz and is operated using double data rate (data is transferred both in the rising and the falling clock edges). The data bus size (DBS) depends on the number of DRAM devices inside the ranks. Each DRAM device contributes with 16 bits. Hence, a rank with 1, 2, or 4 devices would have a data bus size of 16, 32 or 64, respectively.

We explain the scheduling mechanism. Each VD is a group of 2 banks from the same rank. There are a total of 8, named VD_0 to VD_7. A TDM arbiter periodically grants time slots to the VDs. VDs use a time slot to serve a read or write request (by issuing an ACT and a posted-CAS commands). The schedule is depicted in figure 4. Notice that each slot is 5 clock cycles long. We refer to this value as slot width (SW). Since there are 8 VDs, granting one time slot to each of them takes 40 cycles. We refer to this value as round width (RW).

We detail the contents of a slot. In the first cycle, an ACT is issued to open the required row. There is no need to precharge the bank before opening the row because the controller always uses the Auto Precharge Flag for the posted-CAS commands. In the second cycle, the controller issues a

[1]For $BL = 4$ and $BL = 8$, respectively.
[2]See footnote 1.

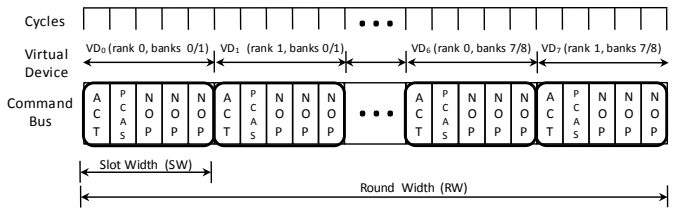


Fig. 4: A TDM scheduling round.

posted-CAS command to write or read data. The posted-CAS command is required to avoid a t_{RCD} constraint violation. The other 3 cycles of the slot are filled with NOPs to match the slot width of 5. The slot width must be 5 (instead of the 4 cycles required by a data transfer) because there is a 1 cycle offset between read and write latencies (t_{CAS} and t_{CWD}). If a VD is granted a turn in the TDM schedule but there are no outstanding requests for it, its time slot is filled with NOPS, i.e. it is said to be *empty*.

We explain the refresh strategy. The controller manually refreshes the DRAM rows when it is necessary. A row can be refreshed by simply loading it into the row buffer and sending it back to the data matrix. This is accomplished using a time slot to read data from the corresponding row (the data is simply discarded). Each VD has 32768 rows that need to be refreshed every 64 ms (2 banks, each with 16738 lines). This requires a row to be refreshed every $64ms/32678 = 1.9531\mu s$ or $1.9531\mu s/(1\mu s/666.67cycles) = 1302.08$ cycles. Which means that for each virtual device, every $\lfloor 1302/40 \rfloor = 32$th slot would be used for a refresh operation. We refer to the percentage of slots that are not refresh operations as refresh efficiency (RE). For the DDR3-1333H, the refresh efficiency is $RE = 31/32 = 0.9687$.

We calculate the bandwidth provided by this approach. If no refreshes are accounted for, each VD is able to transfer DBS bytes every RW cycles. Incorporating the refresh efficiency, the bandwidth provided by a virtual device (bw_{vd}) is given by equation 1. The total bandwidth (bw_{total}) is given by multiplying bw_{vd} with the number of VDs. For instance, for a module with a data bus width of 32 bits, the controller provides a total bandwidth of 4131.84MB/s, each device being responsible for 516.48MB/s.

$$bw_{vd} = \frac{DBS\ bytes}{RW cycles * \frac{1\mu s}{666.67 cycles}} * RE \quad (1)$$

Finally, we discuss the number of VDs in which the memory can be split. For the DDR3-1333H, the bank privatization scheme needs at least 8 VDs to function without the insertion of unnecessary NOPs in the command bus. This is because accesses to the same bank must be at least 40 cycles apart, which is the time required to open a row, issue a write command, transfer the data and precharge the row for the next access. Having 8 VDs (each being granted a 5-cycles time slot) matches this constraint perfectly. If a smaller number of VDs is required, e.g. if there are less than 8 requestors, the time slots have to be redesigned. We discuss how this can be done in Section IV.

III. Related Work

Although there are several works that address the problem of achieving predictability for DRAMs [3]–[7], they either do not support mixed criticality, i.e. do not acknowledge that applications can have different criticalities, or they do not offer timing composability, i.e. do not completely eliminate the interference on critical requestors. The first issue leads to more expensive systems because guarantees are given to all the requestors, even those who do not need it. The second issue forces designers to assume the worst case latency for every individual memory transaction, which can be very conservative when applied to determine the worst-case execution time (WCET) of a task that performs many memory requests [8]. Also, not providing composability makes the controllers suitable only for performance monotonic requestors, which is not always the case.

To our knowledge, the only real-time memory controllers that make use of bank privatization are [3] and [6]. In [3] (the work that introduced bank privatization), the authors focused on having a single requestor per VD. The approach is composable, but does not provide support for mixed criticality.

In [6], the authors also only allow one requestor per VD, but try to reduce the bandwidth waste (empty slots) by replacing the TDM arbiter with a sophisticated scheduler that exploits open-page policy, which means that rows are not precharged after a CAS operation. The idea behind it is that processor requests tend to exhibit spatial locality, which means that the probability that consecutive requests target the same row is high. The approach is, however, limited by the number of banks available, as each requestor must be assigned exclusive access to a bank. Neither timing composability nor support for mixed criticality are provided.

All the other approaches treat the memory as an indivisible unit. In [5], the authors introduce the Predator back-end. To share it between multiple requestors, the authors employ a credit-controlled static-priority (CCSP) [17] arbiter. This arbitration scheme does not support the idea of sharing a criticality level between more than one application, neither provides timing composability.

In [18], the authors used the same back-end with a TDM arbiter. The TDM arbiter can be dynamically reconfigured to better suit the current workload. Reconfiguration consists of assigning more or less time slots to an application. The TDM arbiter ensures composability. However, if an application does not use all of its TDM time slots, the bandwidth is wasted. Also, although the authors claim that the controller supports mixed criticality, guarantees are given to all requestors.

In [4], the authors presented the Analyzable Memory Controller (AMC). They group the requestors into a critical and a non-critical groups. The critical group is always given priority. The non-critical group is only served in the absence of the critical. Round-robin is used to arbitrate between requestors

inside the same group. Although the scheme is able to provide upper bounds on request latency, the round-robin arbitration and the lack of independent virtual devices do not ensure timing composability.

Finally, there were also attempts that rely in a combination of software plus COTS memory controllers. In [19], the authors only consider a scenario with a single critical and more than one non-critical requestors sharing the memory. In [20] the authors rely on access pattern prediction and only consider *soft real-time* requestors. Both software approaches do not offer timing composability.

IV. Proposed solution

As described in the Introduction, we allow critical and non-critical requestors to coexist, while providing an interference-free (timing composable) memory for the former. This is achieved by splitting the memory request scheduler into a back-end, that uses bank privatization to provide the illusion of independent virtual devices (VDs), and a front-end, that arbitrates requests at VD level.

In our approach, each VD is shared between at most one critical and a predetermined number of non-critical requestors. The front-end, comprised of one fixed-priority arbiter for each VD, ensures that the critical requestors are unaware of the existence of interfering requestors. The non-critical requestors receive only the portion of bandwidth not used by the critical ones. We present later a partitioning strategy, to help system designers ensure that this portion is big enough to avoid starvation.

Although our approach is generic and can be applied to different DDR3 memory modules, throughout this section we assume a DDR3-1333H dual rank module, with 8 banks per rank. Furthermore, we refer to the number of VDs displayed by the back-end as n. An in-depth discussion about n is also presented later.

The rest of the Section is organized as follows. The architecture of our controller is covered in Section IV-A. The request scheduler is presented in Section IV-B. The discussion about n is covered in IV-C. Timing aspects of our approach are given in Section IV-D. Our partitioning strategy is described in Section IV-E. Finally, we compare the functionalities provided by our controller against those provided by existing real-time memory controllers in Section IV-F.

A. Controller Architecture

The memory controller architecture is presented in Figure 5. It comprises 6 different blocks: ports, address translation, data buffers, scheduler, access controller and data manager. For each VD, there are two sets of request/response ports, one for critical and one for non-critical requestors. The non-critical set of ports must be shared (e.g. through round robin arbitration), shall more than one non-critical requestor be assigned to a VD.

The address translation phase converts a logical address into bank, row and column numbers. The data buffer holds the data that should be written to the memory or the response that

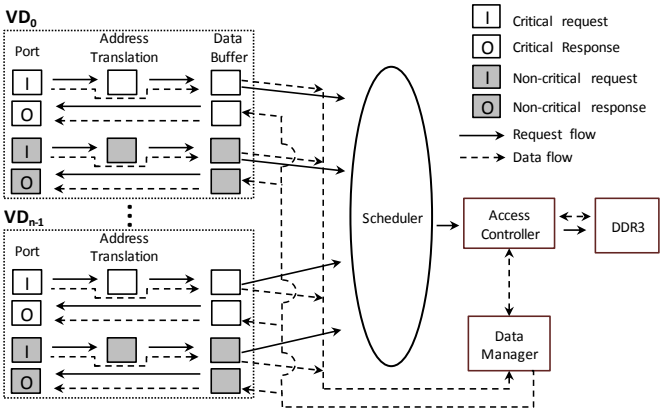


Fig. 5: Controller architecture.

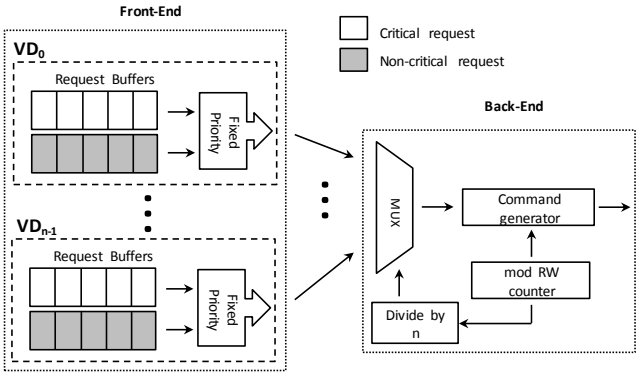


Fig. 6: Scheduler architecture.

should be sent to a requestor in case of a write or read request, respectively. The scheduler is covered in detail in Section IV-B and is left here as a black box. Notice that it only handles requests (addresses and commands). The actual data is stored in the data buffers and retrieved/forwarded by the data manager block when the request is being served. This technique was used by Heithecker et al. in [21], [22] and prevents the data from being routed through the two big multiplexers inside the scheduler. Finally, the access controller handles the physical communication with the DRAM. It is connected to the data manager, from/to which it reads/writes the request data.

B. Scheduler Architecture

The implementation of the scheduler is depicted in Figure 6. The front end is composed of a set of one critical and one non-critical buffers for each VD, plus the corresponding fixed priority arbiters.

The back-end uses a modulo-RW counter to create the illusion of VDs, where RW is the round width (40 cycles). The counter is divided by n, yielding the VD that has the current turn in the TDM schedule. The result of the division is used to control the multiplexer that interfaces with the front-end. The command generator issues commands to the memory (ACT, posted-CAS and NOPs) according to the current request being served and the modulo-RW counter.

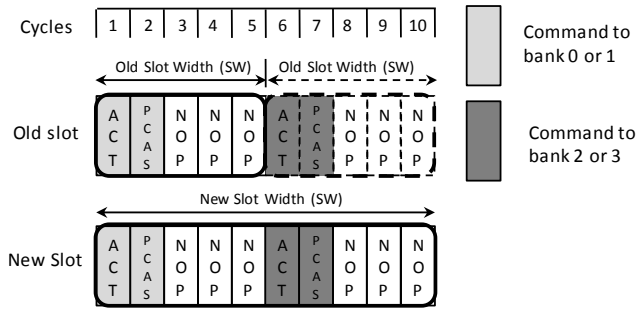


Fig. 7: New slot format (for $n = 4$).

C. Number of virtual devices

Ideally, the back-end should be configured to display one VD per each critical requestor. However, for the back-end to use the memory efficiently, n needs to be a power of 2. To explain the motive, we recall that, for the DDR3-1333H, accesses to the same bank need to be at least 40 cycles apart from each other. Since each time slot in the bank privatization scheme is 5-cycles long, a perfect bank interleaving can only be achieved if n is bigger than $40/5 = 8$.

Therefore, to create a system with less than 8 VDs and still keep a perfect bank interleaving, we use the original VDs (provided using $n = 8$) as a building block to create larger VDs. For instance, it is possible to group the original 8 VDs in pairs, thereby creating 4 larger VDs, each being a group of 4 (instead of 2) banks from the same rank. Every TDM round, each larger VD is granted a 10-cycles time slot, composed of 2 original 5-cycles time slots, as depicted in Figure 7. The same can be employed to configure the back-end to display the memory as a set of 2 or 1 VDs.

This allows us to not only respect the 40 cycles constraint, but also to reuse the refresh strategy (every 32nd slot is used for a refresh). Nevertheless, it must be noticed that reducing the number of VDs increases the request size and bw_{vd} by the same factor, e.g. if the number of VDs is reduced by a factor of 2, the request size becomes two times bigger.

We summarize the relationship between request size, bandwidth and data bus size (DBS) for $n = 8$ and $n = 4$ in table II. For instance, when the DBS is 32, the request size and bw_{vd} are 32 bytes and 516.48MB/s for $n = 8$, and 64 bytes and 1032.96MB/s for $n = 4$. Notice that the total bandwidth (4131.84MB/S) only depends on the DBS.

D. Timing aspects

Our controller provides throughput guarantees to critical requestors: they have access to the entire bandwidth provided by their corresponding VDs. To derive latency guarantees, we employ the busy window approach presented in [23]. We compute a maximum q-*event* busy-time $\beta^+(q)$, which gives an upper bound on the amount of time a VD requires to serve q critical requests, assuming they arrive *sufficiently early*. By sufficiently early, we mean that the q-th request arrives before the previous q-1 were served (see [8]). This would be the case, for instance, for a DMA engine that makes requests bigger than the granularity offered by the controller.[3].

TABLE II: Request size and bandwidth for different values of the data bus size.

	Data bus size (DBS)		
	16	32	64
Number of Virtual Devices (n) = 8			
Request Size	16 bytes	32 bytes	64 bytes
Bandwidth per device (bw_{vd})	258.24MB/s	516.48MB/s	1032.96MB/s
Total Bandwidth (bw_{total})	2065.92MB/s	4131.84MB/s	8263.68MB/s
Number of Virtual Devices (n) = 4			
Request Size	32 bytes	64 bytes	128 bytes
Bandwidth per device (bw_{vd})	516.48MB/s	1032.96MB/s	2065.92MB/s
Total Bandwidth (bw_{total})	2065.92MB/s	4131.84MB/s	8263.68MB/s

Since we are using fixed priority to select between critical and non-critical requestors, we only need to consider the back-end in order to calculate $\beta^+(q)$. The back-end grants one time slot to each VD every scheduling round (RW cycles). Also, it uses every 32nd time slot to perform a refresh operation. Therefore, in the worst case scenario, the first of the q request arrives precisely one cycle after the beginning of a 31st time slot.

We present a graphical representation of the worst case scenario for $q = 2$ in Figure 8. In the Figure, the q requests are directed to VD_0, whose time slots are drawn in black. The time slots granted to the other VDs are drawn in gray. The time required to serve 2 requests is divided in 5 parts. The first one is the phase shift penalty (PSP), and refers to the $RW - 1 = 39$ cycles required before the corresponding VD is granted the next time slot. The second part refers to the overhead imposed by the refresh. The refresh basically burns a time slot, which forces the VD to wait RW cycles for the next usable time slot. The third part accounts for the q-1 scheduling rounds required to serve the first q-1 requests. The fourth part is the time required for the qth transfer to start, counting from the beginning of the corresponding time slot. We refer to this part as CAS. Finally, the fifth part accounts for the transfer time (TT). The transfer time is 4 cycles for $n = 8$ (with 5-cycles time slots), but increases if we use smaller n (bigger time slots).

In case $q > 32$, more than one refresh will be necessary. Hence, instead of RW cycles, the refresh overhead becomes $(\lfloor q * (1 - RE) \rfloor + 1) * RW$, where RE is the refresh efficiency. Consequently, the expression used to calculate the maximum q-*event* busy-time $\beta^+(q)$ is given by equation 2:

$$\beta^+(q) = PSP + (\lfloor q * (1 - RE) \rfloor + 1) * RW + (q - 1) * RW + CAS + TT \quad (2)$$

[3]In such cases, the request is broken into several smaller ones, whose sizes match the controller's granularity.

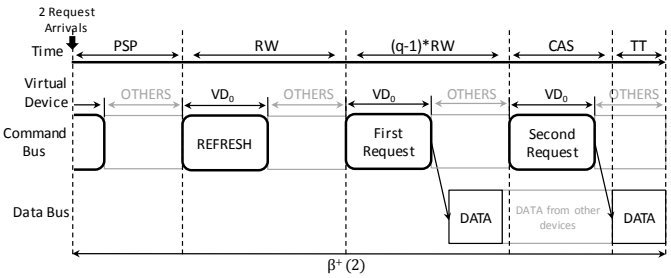
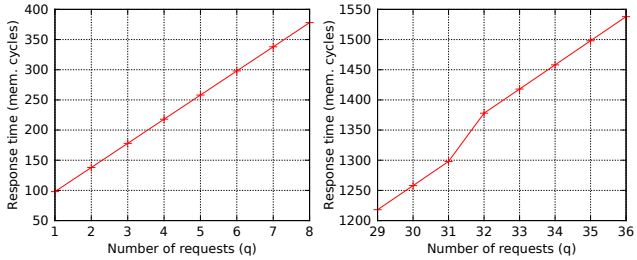


Fig. 8: Computation of $\beta^+(q)$ for $q = 2$.



(a) $1 \leq q \leq 8$ (b) $29 \leq q \leq 36$

Fig. 9: Worst case response time for different values of q.

We plot $\beta^+(q)$ for different values of q in Figures 9a and 9b. The time is measured in cycles from the memory data bus, which runs at a frequency of 666.67 MHz. We assume $n = 8$, which implies $TT = 4$ cycles. The first Figure shows $\beta^+(q)$ for a small number of requests (q between 1 and 8). For instance, when $q = 1$, the worst case latency is 98 memory cycles. In the second Figure, we use bigger values of q (between 29 and 36) to show the influence of refreshes. When q is greater or equal to 32, a second refresh is required, which causes a change in the slope of the graph.

E. Partitioning Strategy

Accommodating a workload on our memory controller requires 3 steps: choosing n, selecting the data bus size (DBS) and mapping applications to VDs. In the first step, we select the smallest n that respects the following two constraints: it must be a power of 2 (1, 2, 4, 8 or even 16 are possible values) and it must be greater or equal to the number of critical requestors in the system. The former constraint is required to keep the bank privatization scheme profitable. The latter allows us to isolate critical requestors from each other. Notice that, if the number of critical requestors is not a power of 2, some VDs will only contain non-critical requestors.

The next step is to choose a data bus size as to match the cache line size of the requestors. For instance, if $n = 4$, the data bus size needs to be 32 so that the request size is 64 bytes (a common cache line size for modern processors).

The final step is to map requestors to VDs. We use a heuristic to do that. We firstly allow every (critical and non-critical) requestor to run in a non-shared VD, i.e. with exclusive access to one of the VDs, and measure its average request load (rate of non-empty slots). Then, we assign the requestors to the VDs as to distribute the average request load evenly, remembering to allocate at most one critical requestor per VD.

To check if the memory is properly sized, we need to observe the sum of average request loads per virtual device. If the sum of the average request load of all requestors allocated to a VD is over 100%, the memory is undersized. In this case, although the critical requestor still remains completely isolated, the non-critical ones are going to have a significative performance penalty. The solution is to add a second, or even more memory controllers to the system, as is often seen in commercially available System-on-Chips (SoCs), e.g. Tile 64 from Tilera [24], which has 4 memory controllers.

TABLE III: Real-time memory controllers features comparison

Controller	Features		
	Predictability	Composability	Mixed Criticality
PRET DRAM [3]	Yes	Yes	No
Open-Page Contr. [6]	Yes	No	No
Predator (CCSP) [5]	Yes	No	Yes
Predator (TDM) [18]	Yes	Yes	No
AMC [4]	Yes	No	Yes
Our Approach (MCMC)	Yes	Yes	Yes

F. Comparison with existing controllers

We do not make an experimental comparison because existing real-time memory controllers do not fundamentally provide the same functionalities as we do. Instead, we make a features comparison in table III. We take three features into account: predictability, composability and mixed criticality support. By predictability and composability, we mean the definitions given in Section II-B. By mixed criticality support, we mean the ability to provide latency and throughput guarantees only to critical requestors.

The left-most column lists the real-time memory controllers. Software approaches were excluded. Each row of the table describes the features of the corresponding controller. For instance, the PRET DRAM approach, which introduced the concept of bank privatization, provides predictability and composability. It does not, however, provide mixed criticality support, as it gives guarantees to all requestors, including the non-critical ones. Our approach is the only one that exhibits the three listed features.

V. Experimental Evaluation

We demonstrate that our scheme allows DRAM sharing with no interference on critical applications and a small overhead for non-critical ones. Although our approach is generic, we consider a dual rank DDR3-1333H module. We devise a mixed critical workload and feed it to a SystemC cycle-accurate model of our controller. We collect statistics about the simulation and discuss the results in detail.

The rest of the Section is organized as follows: Section V-A covers the mixed critical workload. Section V-B details

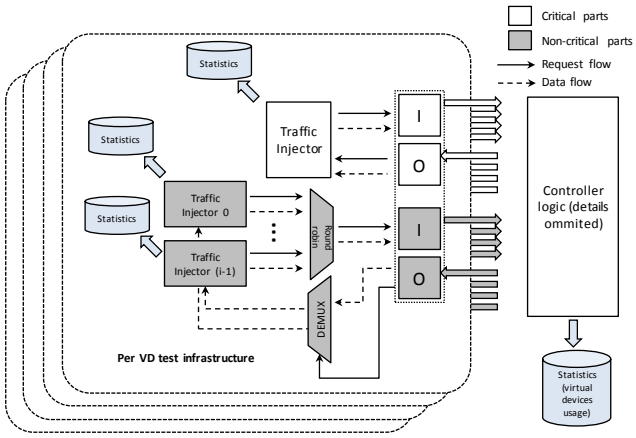


Fig. 10: Block diagram of the simulation infrastructure.

the simulation infrastructure. Section V-C shows the partitioning of the workload. Finally, Sections V-D and V-E present simulation results.

A. Mixed Critical Workload

The mixed critical workload is created using applications from EEMBC [10] and MiBench [9].

The critical applications are represented by four EEMBC AutoBench applications: *matrix01*, *cacheb01*, *aifft01* and *aiifft01*. The first application is a sparse matrix multiplication program. The second emulates the behavior of an application with a high cache miss ratio. The two last are a fast Fourier and an inverse fast Fourier transforms, respectively. The other applications from EEMBC AutoBench perform very little DRAM accesses, and could easily be served with a small ROM. Therefore, we do not use them in our experiments.

The non-critical part of the workload is represented by the Mibench [9] suite. There is a total of 21 applications from five different categories: consumer, office, network, telecommunications and security.

Memory traces were obtained by running the critical and non-critical applications in a non-shared memory environment on the Gem5 [25] platform. An ARM processor clocked at 1GHz with a single level of caching was used. The cache was split into a 32kB data and a 32kB instruction parts. The chosen block size was 64 bytes. The computation time between consecutive requests to the DRAM was measured and recorded into text files. To speed up simulation time, we limited each trace to a $100\mu s$ interval.

B. Simulation Infrastructure

The entire simulation infrastructure is developed in cycle-accurate SystemC. The memory controller model can be configured for a different number of VDs and data bus sizes. We present a block diagram of it in Figure 10. The picture omits the controller implementation details and focus on the traffic injectors and statistics that are collected.

The traffic injectors mimic the behavior of a processor with a cache. They receive as input the memory access traces collected as described in Section V-A. After a request is injected, the injector adds the corresponding computation time and only then issues the next request. We allow a maximum of 4 outstanding requests before stalling the traffic injector. Injecting the entire trace produces the total execution time of an application including both computation and memory access time. We use the expression *execute an application* to refer to the injection of its memory trace.

A round robin arbiter is used to select between the traffic injectors that represent non-critical requestors. This is necessary because, as explained in Section IV-B, there is only one set of non-critical request/response ports per virtual device. The response is routed to the right non-critical requestor using a simple demultiplexer. In the picture, the non-critical traffic injectors and extra logic to share the request/response ports are depicted in gray, while the critical traffic injector is white.

The source code is instrumented to record the following statistics: the execution time of applications and the request load in the VDs. The first one is obtained from the traffic injectors, while the second is retrieved from the back-end. In the Figure, the light blue blocks represent the collected statistics.

TABLE IV: Application mapping to the virtual devices

Mapping		
Virtual Device	Critical Application	Non-Critical Applications
VD_0	*matrix01*	*tiffdither*, *tiff2rgba*, *madplay*, and *qsort*
VD_1	*aifftr01*	*susan*, *tiffmedian*, *cjpeg*, *stringsearch* and *djpeg*
VD_2	*aiifft01*	*crc32*, *blowfish*, *adpcm*, *gsm*, *lout*, *dijkstra* and *tiff2bw*
VD_3	*cacheb01*	*bitcount*, *sha*, *patricia*, *fft* and *basicmath*

C. Partitioning

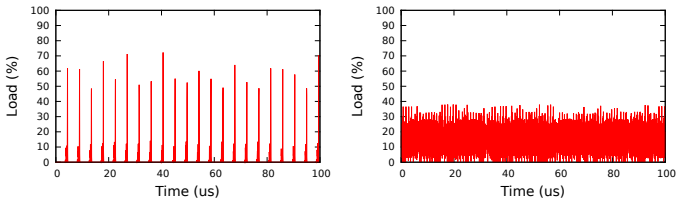
We use our partitioning strategy to map the applications to the VDs. We firstly select the appropriate n. There are 4 critical applications, hence, the back-end is configured to display the memory as a set of 4 independent VDs ($n = 4$). Secondly, we choose the data bus size. The processor that we used to generate our memory access traces has a cache line of 64 bytes. Therefore, we employ a data bus size of 32, which allows a cache line to be transferred with a single request to our controller.

Finally, we map the applications to the VDs. This requires measuring, for all applications, the average request load (rate of non-empty slots) in a non-shared memory environment. Non-shared memory environment refers to giving an application exclusive access to a VD.

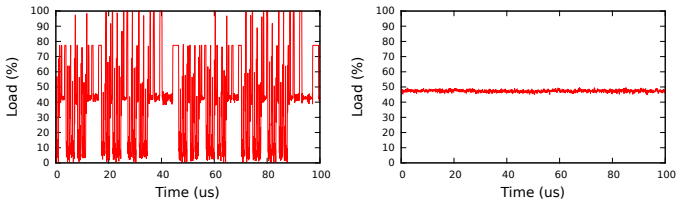
After the average request load is known, we spread the applications to the VDs as to distribute the load evenly. The resulting mapping is presented in table IV. For instance, the critical application allocated to VD_0 is *matrix01*, the non-critical ones are *tiffdither*, *tiff2rgba*, *madplay* and *qsort*.

D. Results - Non-Shared Memory

We investigate the memory access pattern from the applications that belong to our mixed critical workload. To do that,

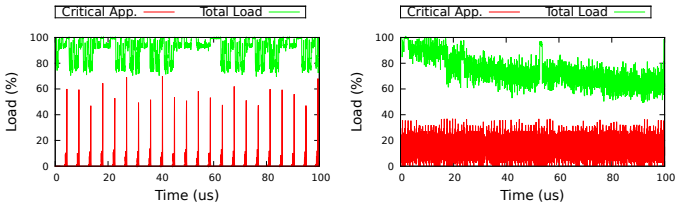


(a) matrix01 (b) aifftr01

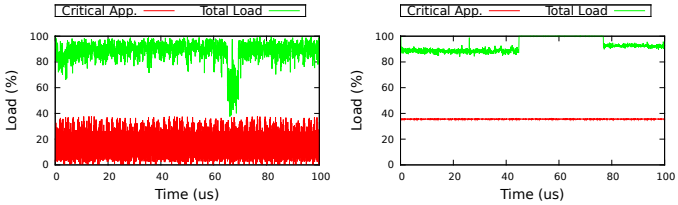


(c) qsort (d) patricia

Fig. 11: Request load of 4 different applications running with exclusive access to one of the VDs.



(a) VD_0 (b) VD_1



(c) VD_2 (d) VD_3

Fig. 12: Request load in each of the 4 virtual devices (the VDs are shared by more than one requestor, as described in table IV).

we execute each of them without the presence of interfering requestors, i.e. with exclusive access to one of the VDs, and measure how its request load (rate of non-empty slots) oscillates through time.

The results are very diverse. Certain applications have a really high cache hit ratio, and therefore access the DRAM very rarely. This is case for *crc32*, *blowfish*, *adpcm*, *stringsearch*, *bitcount*, *sha* and *susan*.

Other applications have a bursty access pattern, alternating periods of intense and low memory activity. This is the case for *matrix01*, *aifftr01*, *aiifftr01*, *madplay*, *qsort*, *cjpeg*, *djpeg*, *dijkstra*, *lout* and *gsm*. The request load graphs for applications *matrix01*, *aifftr01* and *qsort* are presented in Figures 11a, 11b and 11c, respectively. We omit the graphs for the other applications to save space. The graphs were produced sampling the load every 800-slots interval. Notice that they show several spikes, which indicate that the request load oscillates a lot.

Finally, some applications exhibit an *stable* access behavior through time. This is the case for *cacheb01*, all the *TIFF* manipulation applications and for *patricia*. To save space, we only show the request load graph for *patricia* (Figure 11d).

E. Results - Shared Memory

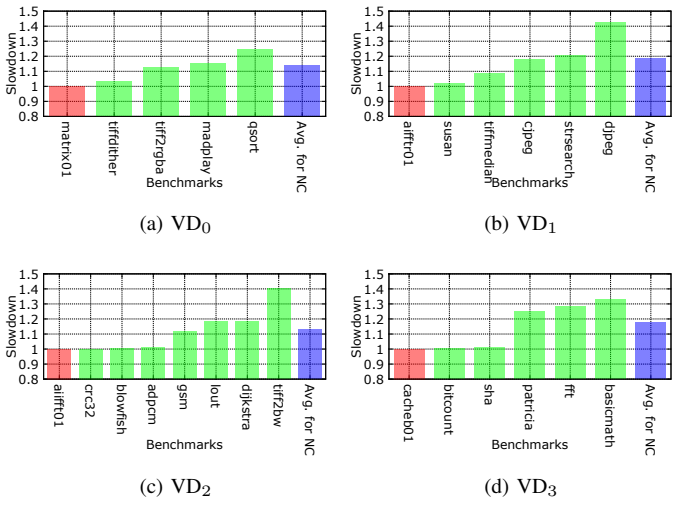
In this Section, we present simulation results obtained executing the entire workload simultaneously, as opposed to the previous Section, which does not consider VD sharing.

We partition the mixed critical workload according to what was described in Section V-C. We simulate and measure the oscillation of the request load imposed on the 4 VDs (named VD_0 to VD_3). We also compute the slowdown of each application. The slowdown is the ratio between the execution time in the non-shared and in the shared memory environment. For instance, a slowdown of 1.15 means that the application runs 15% slower when sharing the memory with interfering requestors. The smaller the slowdown, the smaller the overhead of executing the application in a shared-memory environment.

The request loads imposed by the workload on VD_0, VD_1, VD_2 and VD_3 are presented in Figures 12a, 12b, 12c and 12d, respectively. The graphs were produced by sampling the load every 800-slot interval. The red line represents the percentage of slots occupied by requests issued by the critical requestor. The green line represents the total load. The total load is defined as the percentage of slots that are non-empty. The average total load during the entire period of 100 μs was 90.7%, 74.4%, 88.1% and 93.1% for VD_0, VD_1, VD_2 and VD_3, respectively.

The slowdown for the applications that were allocated to VD_0, VD_1, VD_2 and VD_3 are presented in Figures 13a, 13b, 13c, 13d, respectively. In the Figures, the (leftmost) red bar represents the critical application and the green bars (in the middle) represent non-critical ones. The (rightmost) blue bar is used for the average slowdown of all the non-critical applications.

The slowdown for the critical applications was 1.0 in all 4 virtual devices. This means that the applications' execution times remain the same in the non-shared and in the shared memory environment, which confirms that our controller provides timing composability for critical requestors. The average slowdown for the non-critical applications was 1.14, 1.18, 1.12 and 1.17 for applications allocated to VD_0, VD_1, VD_2 and VD_3, respectively. The average slowdown over all non-critical applications was 1.15. The non-critical applications least affected by running in a shared memory environment were, as expected, the ones that presented a high spatial locality. *Crc32*, *blowfish* and *adpcm*, for instance, mostly only had compulsory cache misses and, therefore, made very little



(a) VD_0 (b) VD_1

(c) VD_2 (d) VD_3

Fig. 13: Slowdown per application.

DRAM accesses. Applications that require more often DRAM accesses, such as *jpeg* decoding, suffered the biggest impacts.

VI. Conclusion

In this paper we propose a memory controller suitable for mixed critical environments. We reuse the bank privatization scheme proposed in [3], that presents the memory as a set of independent virtual devices with fixed bandwidth. In each virtual device, we allocate one critical and a predetermined number of non-critical requestors. A fixed priority arbiter then eliminates the interference on critical requestors. To test our approach, we devised a mixed critical workload based on applications from Mibench and EEMBC. We collected memory access traces from the applications and fed them to a cycle accurate SystemC model of our controller. The results show that it ensures complete isolation for critical applications while having a minimal performance overhead on non-critical ones (they perform on average only 15% slower in the shared environment). Future work in this area includes providing an environment capable of accommodating more than one critical requestor per virtual device and the development of an FPGA prototype.

References

[1] R. Pellizzoni *et al.*, "Handling mixed-criticality in soc-based real-time embedded systems," in *Proceedings of the Seventh ACM International Conference on Embedded Software*, ser. EMSOFT '09. New York, NY, USA: ACM, 2009, pp. 235–244.

[2] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, April 2010, pp. 13–22.

[3] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: bank privatization for predictability and temporal isolation," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '11. New York, NY, USA: ACM, 2011, pp. 99–108.

[4] M. Paolieri, E. Quiñones, and F. J. Cazorla, "Timing effects of ddr memory systems in hard real-time multicore architectures: Issues and solutions," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1s, pp. 64:1–64:26, Mar. 2013.

[5] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A Predictable SDRAM Memory Controller," in *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM Press New York, NY, USA, Sep. 2007, pp. 251–256.

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

[7] S. Goossens, T. Kouters, B. Akesson, and K. Goossens, "Memory-map selection for firm real-time sdram controller," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, march 2012, pp. 828 –831.

[8] S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Transactions on Embedded Computing Systems (Special Issue on Model Driven Embedded System Design)*, vol. 10-2, no. 22, dec 2010.

[9] M. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001, pp. 3–14.

[10] E. M. B. Consortium *et al.*, "Eembc benchmark suite," 2008.

[11] J. Lee and K. Asanovic, "Meterg: Measurement-based end-to-end performance estimation technique in qos-capable multiprocessors," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, April 2006, pp. 135–147.

[12] R. Wilhelm *et al.*, "The worst-case execution-time problem — overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[13] I. Liu, J. Reineke, and E. Lee, "A pret architecture supporting concurrent programs with composable timing properties," in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, 2010, pp. 2111–2115.

[14] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[15] *JESD79-2F: DDR2 SDRAM Specification*, JEDEC, Arlington, Va, USA, Nov. 2009.

[16] *JESD79-3F: DDR3 SDRAM Specification*, JEDEC, Arlington, Va, USA, Jul. 2012.

[17] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, 2008, pp. 3–14.

[18] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens, "A reconfigurable real-time sdram controller for mixed time-criticality systems," in *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.

[19] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, 2012, pp. 299–308.

[20] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Shah, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, 2013, pp. 55–64.

[21] S. Heithecker, A. do Carmo Lucas, and R. Ernst, "A mixed qos sdram controller for fpga-based high-end image processing," in *Signal Processing Systems, 2003. SIPS 2003. IEEE Workshop on*, Aug., pp. 322–327.

[22] S. Heithecker and R. Ernst, "Traffic shaping for an fpga based sdram controller with complex qos requirements," in *Proceedings of the 42nd annual Design Automation Conference*, ser. DAC '05. New York, NY, USA: ACM, 2005, pp. 575–578.

[23] K. Tindell, A. Burns, and A. Wellings, "An extendible approach for analyzing fixed priority hard real-time tasks," *Real-Time Systems*, vol. 6, no. 2, pp. 133–151, 1994.

[24] S. Bell *et al.*, "Tile64 - processor: A 64-core soc with mesh interconnect," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, Feb 2008, pp. 88–598.

[25] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.