Conservative Open-Page Policy for Mixed Time-Criticality Memory Controllers

Sven Goossens*, Benny Akesson†, and Kees Goossens*
*Eindhoven University of Technology, Eindhoven, The Netherlands
†CISTER-ISEP Research Centre, Polytechnic Institute of Porto, Portugal

***Abstract*—Complex Systems-on-Chips (SoC) are mixed time-criticality systems that have to support firm real-time (FRT) and soft real-time (SRT) applications running in parallel. This is challenging for critical SoC components, such as memory controllers. Existing memory controllers focus on either firm real-time or soft real-time applications. FRT controllers use a close-page policy that maximizes worst-case performance and ignore opportunities to exploit locality, since it cannot be guaranteed. Conversely, SRT controllers try to reduce latency and consequently processor stalling by speculating on locality. They often use an open-page policy that sacrifices guaranteed performance, but is beneficial in the average case.**

This paper proposes a *conservative open-page policy* that improves average-case performance of a FRT controller in terms of bandwidth and latency without sacrificing real-time guarantees. As a result, the memory controller efficiently handles both FRT and SRT applications. The policy keeps pages open as long as possible without sacrificing guarantees and captures locality in this window. Experimental results show that on average 70% of the locality is captured for applications in the CHStone benchmark, reducing the execution time by 17% compared to a close-page policy. The effectiveness of the policy is also evaluated in a multi-application use-case, and we show that the overall average-case performance improves if there is at least one FRT or SRT application that exploits locality.

I. Introduction

Embedded multi-core systems are often used to execute a variety of different applications. The functionality integrated in a single system is increasing, driven by power, area and cost advantages [1], [2]. A consequence of this trend is that hardware platforms can no longer be tailored for a specific application class. Instead, generality is needed to satisfy the diverse requirements of the integrated applications. This is a challenging problem especially in systems with applications of mixed time-criticality.

Mixed time-criticality systems combine both firm- and soft real-time applications. Firm real-time (FRT) applications require firm guarantees on response times from all SoC components, such that bounds on the application-level worst-case execution-time or throughput can be derived [3]. Deadline misses for these applications result in unacceptable loss of functionality or severe quality degradation. Conversely, soft real-time (SRT) applications can tolerate an occasional deadline miss, as long as sufficient average-case performance is provided. Focusing on either the worst- or the average-case can lead to very different design decisions that mixed time-criticality systems have to trade-off [4].

Most SoCs use a single SDRAM as their off-chip memory, which is shared across these different application classes. This papers focuses on the SDRAM controller. Its performance, i.e. the provided bandwidth and latency, has a large impact on the overall SoC performance. Both of these metrics are inherently difficult to bound because the response time of an SDRAM can vary significantly based on its state, and is also input dependent. Modern SDRAMs comprise a hierarchical structure of banks and rows that have to be opened and closed explicitly. Their response time is relatively small if consecutive accesses use the same bank and row, but is large if a different row has to be accessed, since that requires closing the open row and subsequently opening the requested row. Locality thus strongly influences the performance of the memory subsystem.

The potential benefits of exploiting locality are ignored by FRT controllers, since it cannot be guaranteed. For this reason, they typically use a *close-page policy* that maximizes the worst-case performance. SRT controllers often attempt to exploit locality by using an *open-page policy*: they leave a row open until the next request arrives, and by doing so speculate that it will access the same row. If that assumption is wrong, then a latency penalty has to be paid, but if it is true sufficiently often, then an open-page policy outperforms a close-page policy [5]. Since offering high average-case performance is essential for mixed time-criticality systems, it makes sense to try and exploit locality even in the presence of FRT applications.

This paper contains the following contributions: 1) A *conservative open-page policy* that can be used in a *mixed real-time* (MRT) context. The policy can be applied in any existing FRT controller, improving its average-case performance without sacrificing its original real-time guarantees. The main idea is to allow the command scheduler to exploit locality that presents itself within the time window that naturally exists between opening and closing a row. In this window, the controller can exploit a fraction of the locality available in the request stream, without increasing the worst-case schedule length. 2) A generic method that derives a conservative open-page command schedule based on a close-page schedule. 3) The conservative open-page policy is implemented in an existing FRT memory controller, which is used to experimentally quantify the performance benefits for a set of applications from the CHStone benchmark. The influence of the controller configuration, i.e. the number of banks each request is interleaved over and the number of bursts per bank, are also discussed, both from a FRT and a SRT perspective. The experiments show improved SRT performance without compromising FRT guarantees.

The rest of this paper is organized as follows. In Section II, related work is discussed. Section III gives background on

978-3-9815370-0-0/DATE13/©2013 EDAA

SDRAM controller architectures. Section IV describes the conservative open-page policy. The policy is experimentally evaluated in Section V, followed by conclusions in Section VI.

II. Related work

Memory controllers targeting SRT applications often apply sophisticated mechanisms to improve average-case performance. Examples include the enhanced exploitation of locality, prioritizing requests that target open rows [6], [7], grouping read and write requests to reduce bus turnaround overhead [8], [9] and even re-enforced learning optimization [10]. To reduce the likelihood of a page-miss, techniques that specifically focus on optimizing the moment at which a row is closed have also been proposed [11]. All these techniques interact with the command scheduling in complex ways that are effectively impossible to analyze, which means no useful bounds on the real-time performance can be derived. This makes it impossible to use these controllers in a FRT context.

Several SDRAM controllers focusing on FRT applications have been proposed, all trying to maximize the worst-case performance. [12] uses a static command schedule computed at design time, thus allowing exploitation of locality and complete analysis of the provided bandwidth and latency. However, full knowledge of the applications' behavior is required at design time, which is generally not possible. Furthermore, it is unable to deal with request streams that are input dependent. The controller proposed in [13] dynamically schedules precomputed sequences of SDRAM commands according to a fixed set of scheduling rules. The sequences obey the SDRAM timing constraints and have fixed known latencies allowing the derivation of bounds on the provided bandwidth and latency. The controller proposed in [14] follows a similar approach. [15] dynamically schedules commands at run-time according to a set of rules from which an upper bound on the execution time of a request can be determined. All of the above mentioned dynamic FRT controllers use a close-page policy and thus do not exploit locality across requests.

This paper introduces a conservative open-page policy that can be used in a FRT memory controller, enabling parts of the locality to be exploited without degrading worst-case performance. This means that the original worst-case analysis and guarantees are still valid, while the average-case performance improves. This policy is therefore useful for controllers in a MRT system.

III. Background

An SDRAM memory consists of multiple independent memory banks, each containing a memory array consisting of rows and columns. Each element in a column is a memory word. A *write* (WR) or *read* (RD) command results a transfer of multiple words, called a *burst*. For a DDR3 memory, the *burst length* (BL) is 8 words. Before data can be read or written, a row has to be opened using an *activate* (ACT) command, which places the row in the row-buffer of its bank. To access a different row, the row-buffer first has to be *precharged* before the next row can be activated. Precharging closes a row and restores it in the memory array. This can be done by either issuing an explicit-precharge command (PRE), or by setting the *auto-precharge* flag while performing the last RD or WR command to the active row. Using auto-precharge closes the row as fast as possible. All the banks share a single data- and command-bus, but in principle they are independent memories. This allows the use of bank-parallelism, for example by reading data from bank 0 while activating bank 1.

The minimum delay between each pair of commands is determined by a set of timing constraints [16]. For example, *tRC* specifies the minimum time between two ACT commands to the same bank, while *tRCD* denotes the minimum time between an ACT and a RD or WR command. *tRTP* is the minimum time between a RD and a PRE command and *tRP* is the minimum PRE-to-ACT delay. The challenge for a memory controller is to create a sequence of memory commands or *schedule* that satisfies all timing constraints.

Memory requests often have a granularity that is larger than a single burst. L2 cache lines for example are typically 64 or 128 bytes. This property makes using more than one bank to serve a single request possible, which enables the use of bank parallelism to hide (parts of) the latency caused by timing constraints. The size of requests with which an SDRAM controller works is called the *access granularity* (AG). It is given by the number of banks a request is interleaved over (*BI*), multiplied by the number of bursts per bank (*BC*) and the memory interface width. A single row stored in a row buffer is called a *page*. The combination of banks that a single request is interleaved over is referred to as a bank-cluster. We call the combination of BI and BC a *controller configuration*. The choice of BI and BC greatly influences the worst-case performance of an SDRAM controller [17]. Although there are many small nuances, in general the worst-case bandwidth and latency grows with the access granularity. For a constant access granularity, it is beneficial to interleave over as many banks as possible to exploit bank-parallelism.

Several row-buffer-management policies or *page-policies* exist [5]. A memory controller can choose to precharge the active row as soon as possible after a request to that row is finished, which is called a *close-page policy*. This minimizes the execution time of the next request once it arrives, if it wants to access a different row. FRT controllers generally use a close-page policy with auto-precharges, because they perform better than open-page policies in the worst case. Alternatively, a controller could leave the active row open until the address for the next request is known, which is called an *open-page policy*. This policy benefits from locality by amortizing the latency of activating and precharging a row across multiple requests. However, if a different row has to be accessed then the full penalty of precharging the active row and activating a new row has to be paid. Controllers that use an open-page policy thus speculate that enough locality is present in the request stream such that they perform better in the average case.

Throughout this paper, we use a MT41J64M16 DDR3-1600 module from Micron [18] to demonstrate the proposed policy. Such a module runs at 800 MHz, has a 2 byte interface width and a data rate of 2 words per cycle, resulting in a peak bandwidth of 3.2 GB/s. Note that the method described in this paper is not specific for this module, but applies to any type of SDRAM memory.

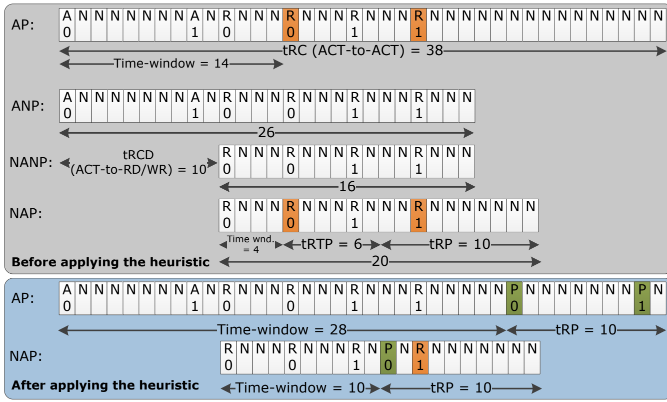

Fig. 1. Read command schedules for BI 2, BC 2. Each block represents a command, followed by an optional bank id. The orange commands have auto-precharge flags. The timing constraints that dictate the length of the schedule are shown on the arrows.

IV. Conservative open-page policy

A. Naive implementation

Exploiting locality is beneficial for the average-case performance of a memory controller, but speculative open-page policies reduce the worst-case guaranteed bandwidth and latency, which is unacceptable for FRT applications. We hence propose a *conservative open-page policy* that exploits part of the available locality without sacrificing worst-case performance. Starting off with a close-page schedule [19], the core idea is to allow a page to remain open after an access *if it is known* that the next request targets the same row in the same bank-cluster. If this is the case, then a hit is detected. The auto-precharge flags, used in the close-page policy at the last read or write command to each bank, are then omitted, as are the NOPs that are normally scheduled to satisfy the PRE-to-ACT and ACT-to-ACT constraints. The next memory access does hence not have to incorporate any activate commands, and the NOPs required to satisfy the activate-to-read/write constraint can also be omitted. This is conservative with respect to the worst-case guarantees, because the length of an access can only decrease as a result of this policy. The policy can be applied to all FRT schedulers, although the implementation effort may vary based on the implementation of their close-page scheduler.

Fig. 1 shows an example of the 4 different read schedules that are used in this policy: 1) AP: activates and precharges a page. The schedule is used if a closed page is accessed, and the next request needs another page. This is the schedule that is always used by a close-page policy. 2) ANP: contains an activate, but no precharge. A transition from the AP or NAP schedule to this schedule is made if the next access is a page-hit and it is detected in time. 3) NANP: A schedule that contains only RD or WR commands and is used if the previous and next request are both page-hits. 4) NAP: This schedule is used if the previous request was a hit, but the next request is a miss.

Fig. 1 shows that for BI 2, BC 2, a read request needs 38 cycles in the close-page policy, of which only 16 cycles are used to transfer data. The average overhead caused by opening and closing a page in all configurations up to a granularity of 64 bytes is 70% and 76% for reads and writes, respectively. This shows that, if locality is exploited, it has a large impact on the efficiency with which the memory is used.

To detect hits, the memory controller inspects the address of the next request that is scheduled by the resource arbiter. This address has to be known *before the first precharge* would be executed in the AP or NAP schedules. If the target address for the next access is not known by that time, the controller has to assume a miss to prevent sacrificing worst-case guarantees. This implies that there is a limited time window in which locality can be exploited. The size of this window depends on the time required to activate a row, plus the time required to access all bursts from the first bank in the bank-cluster. The larger the burst count, the more time exists between the start of an access and the decision moment.

Similarly, there also exists an address window in which locality can be exploited, the size of which depends on the number of banks in a bank-cluster. When more banks are clustered, there are more row buffers to hold active data which means the effective page size is larger. The address range in which the next access is considered a hit is therefore also larger.

B. Increasing the time-window size

The size of the time-window has to be as large as possible to maximize the exploited locality. To maximize the window size, the precharge decision must thus be made as late as possible. For this purpose, we propose to *replace the auto-precharge flags with explicit precharges* that happen later in the schedule. To maintain the same worst-case guarantees, we do not allow the schedule to increase in length as a result of the replacement. A small heuristic is used that generates the modified schedule, based on an existing schedule that uses auto-precharges. After applying the heuristic, the size of the time-window is larger than or equal to the original window size, with no influence on the schedule length. Therefore, it is always recommended to apply this heuristic when using the policy.

The heuristic attempts to increase the size of the window by prioritizing the replacement of the *first* auto-precharge in the schedule over those that happen later. The PRE commands are placed as close to the maximum window length as possible. By doing so, the heuristic reduces the chance of placing the PRE command for an early bank needlessly close to the end of the schedule, which could be the only spot where a PRE command for a later bank is allowed to be placed.

The heuristic works as follows:

1) Find the first auto-precharge for which one of the following conditions holds: a) the RD/WR-to-PRE constraint only allows the corresponding explicit PRE command beyond the end of the original schedule, or b) all the cycles in which the explicit PRE command is allowed are already used by other commands in the original schedule.
2) The first auto-precharge that meets this requirement sets the maximum size for the time-window that can be used conservatively. All other precharges should ideally be placed after this command. If all auto-precharges can be replaced, then the heuristic tries to put the first PRE command BI-cycles from the end of the schedule to leave enough space for the PRE commands for the other banks in the bank-cluster.

TABLE I
TIME-WINDOW SIZES USING THE CONSERVATIVE OPEN-PAGE POLICY AND THE AMOUNT OF CYCLES CONTRIBUTED BY THE HEURISTIC FOR THE SCHEDULES CONTAINING PRECHARGES.

BI	1	1	2	1	2	4
BC	1	2	1	4	2	1
AG [bytes]	16	32	32	64	64	64
AP-RD [cc]	28 (+18)	28 (+14)	15 (+5)	28 (+4)	28 (+14)	15 (+5)
AP-WR [cc]	34 (+24)	28 (+14)	15 (+5)	46 (+24)	38 (+24)	15 (+5)
NAP-RD [cc]	14 (+14)	10 (+ 6)	4 (+4)	18 (+6)	10 (+6)	6 (+6)
NAP-WR [cc]	24 (+24)	28 (+24)	24 (+24)	36 (+24)	28 (+24)	12 (+12)

3) Try replacing the first auto-precharge in the schedule by an explicit precharge, by finding a cycle that does not violate timing constraints if the PRE command would be put there. First the cycles after the unmovable precharge are checked, working towards the end of the schedule. If no suitable location can be found due to timing constraint violations or because there is no space, the cycles before the unmovable precharge are checked, working towards the start of the schedule. This reduces the window-size the heuristic attempts to create. The RD/WR-to-PRE constraint limits the number of possible locations for the precharge command in this search direction.
4) Once a suitable location for the precharge is found, the auto-precharge flag can be removed and the PRE command can be inserted into the schedule. If no suitable location is found or if the first precharge in the schedule is now explicit, then stop, otherwise repeat step 3 with the (new) first auto-precharge.

Applying the heuristic to the example in Fig. 1 increases the time-window size from 14 cycles to 28 cycles in the AP schedule, and from 4 to 10 cycles in the NAP schedule. The results for the other configurations up to an access granularity of 64 bytes are shown in Table I. Postponing the precharge-decision by increasing the number of bursts per bank increases the size of the time-window, so to maximize the window the largest possible burst size at a specific access granularity has to be used. The effects this has on performance are discussed in Section V-B3.

V. EXPERIMENTS

The CHStone benchmark [20] is used to evaluate the proposed policy. For each application in the benchmark, a memory-trace file was generated by running it on a SimpleScalar 3.0 processor simulator [21]. The simulator was slightly modified to record the time and address of each L2 cache miss which results in a trace file containing all requests that go to the SDRAM. The traces are generated using the out-of-order execution engine (sim-outorder) with default settings except for the cache configuration. We use a unified 128 KB L2 cache with 64 byte cache lines, 512 sets and an associativity of 4. Each request in the trace thus corresponds to a cache miss of 64 bytes. Eight out of the twelve applications in the benchmark are used, the four applications that are left out use 64-bit floating-point operations that are not supported by the SimpleScalar compiler. The average requested bandwidth and the number of requests in each trace can be found in Table II.

Three factors determine the number of page-hits for an application. 1) Spatial locality within an application. The address of a request is decoded to a row, column and bank address. If a request targets the same row and bank as its predecessor, then it is a potential hit, otherwise it is a guaranteed miss. 2) Temporal locality: requests have to arrives at the memory controller before the time-window closes. 3) Interference from requests by other applications. In the following sections, the consequences of each of these factors are evaluated for the proposed policy.

TABLE II
TRACE CHARACTERISTICS

Trace	adpcm	aes	bf	gsm	jpeg	mips	motion	sha
Avg. bandwidth [MB/s]	846	878	253	1910	100	1577	2426	236
Number of requests	645	742	873	644	1685	541	617	791

A. Spatial locality

To set a baseline for the experiments, we first pre-process the traces to determine the spatial locality, i.e. the fraction of consecutive requests that target the same page. We assume that requests are not reordered, such that requests from a single application are always processed by the memory controller in the order of arrival. Reordering of requests is used by SRT controllers to improve efficiency and page-hits, but it is not used by firm real-time controller since it increases the analysis complexity and/or causes an unacceptable loss of worst-case performance.

Which bits from the address are used to determine the bank and row addresses depends on the controller configuration; the more banks a request is interleaved over, the larger the combined row-buffer size of each bank-clusters is. This means that the likelihood of two consecutive requests targeting the same bank-cluster increases with a growing BI, resulting in more hits.

Fig. 2 shows the spatial locality per trace for the three different memory configurations that offer a 64-byte access granularity. The graph shows that at least 57% of the requests in each trace can potentially benefit from an open-page policy, with a variation of at most 8% caused by the different configurations.

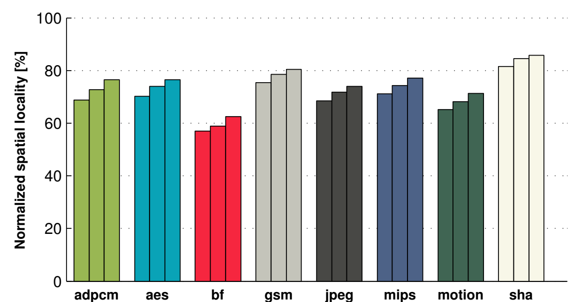

Fig. 2. Spatial locality per trace for three controller configurations, from left to right: (BI 1, BC 4), (BI 2, BC 2) and (BI 4, BC 1).

B. Conservative open-page policy evaluation

The conservative open-page policy is implemented in a transaction-level SystemC model of the memory controller described in [13]. The controller uses a set of command schedules called *patterns* that are generated at design time. At run-time these patterns are issued based on the incoming request type. There are different pattern types, two of which are used for read and write accesses. Two other patterns are

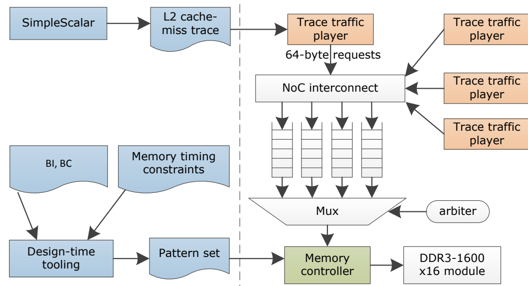

Fig. 3. Experimental setup

inserted if needed to account for bus-turnaround time when switching the data direction.

To enable the conservative open-page policy, the controller is made aware of the open row in each bank-cluster. Based on this information, one of the four schedule modes (AP, NAP, ANP or NANP) is selected for each incoming request and the appropriate pattern for that mode is issued. All experiments are performed using the three 64-byte controller configurations. First the results for BI 2, BC 2 are shown, and then the effect of the configuration is discussed.

The controller is connected to a multiplexer such that it can be shared amongst multiple master ports. We use a setup with four master ports, each of which is connected to a trace-based traffic player by means of a NoC, as shown in Fig. 3. The traffic players emulate a processor running at 1400 MHz, which means that each cycle in the trace corresponds to 0.71 ns. A player allows a maximum of four outstanding read-requests before it stalls, and further assume that all requests are independent. Note that a real processor could potentially stall due to dependencies, but that it is not uncommon for multiple cache misses to arrive at a memory controller in a relatively short interval [22], and that techniques exist to improve the available memory parallelism [23].

1) Single-application performance: In the first experiment, only one of the four trace players is active, running each of the application traces independently. We run the application two times, first using a close-page policy and then using the conservative open-page policy. The results of these experiments are shown in Fig. 4. We first determine the fraction of requests containing spatial locality that is captured within the time-window. In a single application use-case, 70% of those requests results in a hit on average, which is shown by the (striped) bars in the figure.

Next, we look at the execution-time difference as a measure for the SRT performance gains. Here, results vary wildly based on which application is used: the execution time of JPEG is reduced by only 1%, while that of MOTION is reduced by 33%. This difference can be explained by looking at the memory load of the applications in Table II. The high load of MOTION implies that it is memory bounded, and thus sensitive to changes of memory response times. Conversely, JPEG has a low load, which means it spends most of its time processing and memory response time was only a small fraction of the total execution time to begin with. The average execution-time reduction across all applications is 17%, so we conclude that our proposed technique works well, and that the benefit for an application scales with how memory intensive it is.

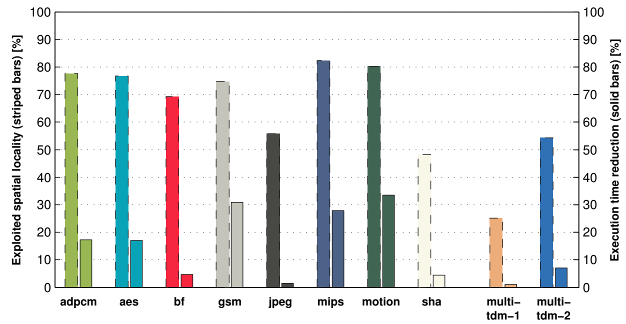

Fig. 4. The striped bars show the fraction of the spatial locality captured by the conservative open-page policy (higher is better). The solid bars show the execution time reduction achieved by the conservative open-page policy, normalized to the close-page execution time. The solid bar for the two multi-application experiments show the total execution time reduction of the running applications, normalized to the largest close-page execution time across the two experiments. All experiments in the graph use BI 2, BC 2.

2) Multi-application performance: In the second experiment, four applications run simultaneously on four separate trace players and compete for the memory resource. This experiment shows the effect of multi-application interference on locality exploitation. Two high and two low-load applications are used (MIPS, MOTION, JPEG and BF). Work-conserving Time-division multiplexing (TDM) is used as the arbitration scheme, which means that unclaimed slots from one application can be used by another application if it has a request available. In the MULTI-TDM-1 experiment, each application gets one out of four slots in the TDM table, which means that it can at least get a quarter of the memory bandwidth. This potentially leads to fine-grained interleaving of requests from different applications, which destroys locality that was present in the original memory trace. This effect is visible in Fig. 4: only 25% of the potential locality is captured which is significantly lower than the average captured locality in the single application case. Consequently the total execution-time reduction is also relatively small. The individual execution time reduction of the applications ranges from 0 to 3.5%.

To retain more of the locality in the request stream, the arbiter in the MULTI-TDM-2 experiment is modified: each application gets two consecutive slots in a TDM table having eight slots in total. This has implications on the worst-case response time resulting in a trade-off between FRT and SRT performance; where in the first TDM-schedule there were at most three interfering slots, in the second there are at most six interfering slots. Giving an application two consecutive slots has a large impact on the fraction of exploited locality: 54% of it captured, more than 2 times as much as in the MULTI-TDM-1 experiment, resulting in a total execution-time reduction of 7%. The individual execution times drop between 2.6% (JPEG) and 27% (MIPS). We can conclude that successfully applying the conservative open-page policy in a multi-application use-case is possible, under the condition that the arbiter allows at least part of the consecutive requests from an application to be scheduled consecutively.

3) Controller configuration influence: The controller configuration has a large impact on the worst-case guaranteed bandwidth and latency, as shown in [17]. The configurations that we consider all have a granularity of 64 bytes: interleaving over either 1, 2 or 4 banks, while doing 4, 2 or 1 burst per bank,

TABLE III
RESULTS FOR EXPERIMENT 3.

Banks Interleaving	1	2	4
Burst Count	4	2	1
Average exploited locality [%]	78.7	70.6	70.1
Average exec. time reduction [%]	16.1	17.2	17.0
Worst-case bandwidth [MB/s]	901	1050	1144

respectively. Table III shows that the worst-case bandwidth delivered by those configurations increases with BI, with a 21% difference between BI 1 and BI 4. Fig. 2 shows a trend in the same direction; the higher BI, the higher the spatial locality. A trend in the opposite direction is visible for the window size (see Table I). This leads to the observation that for an increasing BI, both worst-case bandwidth and spatial locality increase, but the size of the time-window decreases.

Table III shows the average fraction of exploited locality and the average execution time for the eight benchmark applications. The conservative open-page policy captures the largest fraction of potential locality in the configuration with the largest window-size (BI 1, BC 4). However, the execution-time reduction is largest for the (BI 2, BC 2) configuration. In absolute numbers, the average execution time for (BI 2, BC 2) is only 0.3% smaller than that of (BI 1, BC 4). We conclude that the performance differences across configurations are so small that they are insignificant, so the selection of a configuration can be made based on the real-time guarantees that it offers to FRT applications without significantly impacting SRT performance.

4) MRT performance: A MRT workload is tested in the final experiment. Two traffic players are active in this experiment: the first one runs each of the benchmark applications and is configured to block immediately when a request is issued, unblocking once the response arrives. This means the application does not benefit from any of its own locality. This models what happens in case an in-order processor is used to execute the application. The second trace player runs a FRT video task that we model using a synthetic traffic stream. It requires a bandwidth of 270 MB/s, which is the combined read and write rate required to transport 60 frames of $1024 \cdot 786$ pixels from and to the memory, assuming 3 bytes/pixel. A high degree of locality is available in this application stream and we assume all of its requests are independent, i.e. the IP running the application is fully pipelined. The objective of this experiment is twofold: it allows us to experimentally verify that the real-time constraints of the FRT task are still satisfied, and it also quantifies the impact of the locality exploitation by the pipelined FRT application on the execution time of the non-pipelined SRT application.

Compared to a close-page policy, the average execution time of the benchmark applications is reduced by 7.9% using the conservative open-page, while still satisfying the constraints of the FRT application. Using the policy, the controller manages to serve the FRT application faster, allowing more time to be spent on the SRT application which hence benefits indirectly. The MOTION benchmark again benefits most (13.4%), while JPEG shows the smallest improvement (1.9%). We conclude that if there is at least one application that exploits locality, then all the other applications that share the memory can benefit and the overall average-case performance increases.

VI. CONCLUSION

This paper deals with the problem of mixed time-criticality workloads in the context of an SDRAM controller. Existing controllers optimize for either worst-case or average-case performance, but not for the combination of the two. We proposed a conservative open-page policy that improves the average-case performance without sacrificing real-time guarantees. It exploits some of the locality in the request stream, which reduces the average-case latency. A method to create the conservative open-page schedules based on an existing close-page schedule is shown and the influence of the controller configuration on its performance is evaluated. The execution-time reduction for single- and multi-application use-cases is quantified, and we show that it is beneficial for the overall average-case performance as long as at least one of the memory clients has multiple requests that can be scheduled consecutively.

VII. ACKNOWLEDGEMENTS

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, Catrene CA104 COBRA, ARTEMIS 100202 RECOMP, PT FCT, and NL STW 10346 NEST.

REFERENCES

[1] “International Technology Roadmap for Semiconductors (ITRS) - System Drivers,” 2011, http://www.itrs.net/reports.html.

[2] C. van Berkel, “Multi-core for Mobile Phones,” in *Proc. DATE*, 2009.

[3] L. Steffens *et al.*, “Real-Time Analysis for Memory Access in Media Processing SoCs: A Practical Approach,” *Proc. ECRTS*, 2008.

[4] T. Henzinger and J. Sifakis, “The discipline of embedded systems design,” *Computer*, vol. 40, 2007.

[5] B. Jacob *et al.*, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Pub, 2007.

[6] J. Shao and B. Davis, “A burst scheduling access reordering mechanism,” in *Proc. HPCA*, 2007.

[7] O. Mutlu and T. Moscibroda, “Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems,” *ACM SIGARCH*, vol. 36, 2008.

[8] A. Burchard *et al.*, “A real-time streaming memory controller,” in *Proc. DATE*, 2005.

[9] S. Heithecker and R. Ernst, “Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements,” in *Proc. DAC*, 2005.

[10] E. Ipek *et al.*, “Self-optimizing memory controllers: A reinforcement learning approach,” in *Proc. ISCA*, 2008.

[11] J. Dodd, “Adaptive page management,” US Patent 7,076,617.

[12] S. Bayliss and G. Constantinides, “Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search,” in *Proc. FPT*, 2009.

[13] B. Akesson *et al.*, “Architectures and modeling of predictable memory controllers for improved system integration,” in *Proc. DATE*, 2011.

[14] J. Reineke *et al.*, “PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation,” in *Proc. CODES+ISSS*, 2011.

[15] M. Paolieri *et al.*, “An Analyzable Memory Controller for Hard Real-Time CMPs,” *Embedded Systems Letters, IEEE*, vol. 1, no. 4, 2009.

[16] *DDR3 SDRAM Specification*, JESD79-3E ed., JEDEC, 2010.

[17] S. Goossens *et al.*, “Memory-Map Selection for Firm Real-Time Memory Controllers,” in *Proc. DATE*, 2012.

[18] Micron Technology Inc., “DDR3-1600-1Gb SDRAM Datasheet, 02/10 EN edition,” 2006.

[19] B. Akesson *et al.*, “Automatic Generation of Efficient Predictable Memory Patterns,” in *Proc. RTCSA*, 2011.

[20] Y. Hara *et al.*, “Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis,” *Journal of Information Processing*, vol. 17, 2009.

[21] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: An infrastructure for computer system modeling,” *Computer*, vol. 35, 2002.

[22] Z. Zhu and Z. Zhang, “A performance comparison of DRAM memory system optimizations for SMT processors,” in *Proc. HPCA*, 2005.

[23] V. Pai *et al.*, “Code transformations to improve memory parallelism,” in *Proc. ACM/IEEE international symposium on Microarchitecture*, 1999.