PREPRINT: "J. Reineke, I. Liu, H. Patel, S. Kim, and E. A. Lee, "PRET DRAM Controller: On the Virtue of Privitization," in proceedings of AC M International Conference on Hardware/Software Codesign and System Synthesis (CODES/ ISSS), 2011, pp. 99-108."

PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation*

Jan Reineke
University of California, Berkeley
Berkeley, CA, USA
reineke@eecs.berkeley.edu

Isaac Liu
University of California, Berkeley
Berkeley, CA, USA
liuisaac@eecs.berkeley.edu

Hiren D. Patel
University of Waterloo
Waterloo, Ontario, Canada
hdpatel@uwaterloo.ca

Sungjun Kim
Columbia University
New York, NY, USA
skim@cs.columbia.edu

Edward A. Lee
University of California, Berkeley
Berkeley, CA, USA
eal@eecs.berkeley.edu

ABSTRACT

Hard real-time embedded systems employ high-capacity memories such as Dynamic RAMs (DRAMs) to cope with increasing data and code sizes of modern designs. However, memory controller design has so far largely focused on improving average-case performance. As a consequence, the latency of memory accesses is unpredictable, which complicates the worst-case execution time analysis necessary for hard real-time embedded systems.

Our work introduces a novel DRAM controller design that is predictable and that significantly reduces worst-case access latencies. Instead of viewing the DRAM device as one resource that can only be shared as a whole, our approach views it as multiple resources that can be shared between one or more clients individually. We partition the physical address space following the internal structure of the DRAM device, i.e., its ranks and banks, and interleave accesses to the blocks of this partition. This eliminates contention for shared resources within the device, making accesses temporally predictable and temporally isolated. This paper describes our DRAM controller design and its integration with a precision-timed (PRET) architecture called PTARM. We present analytical bounds on the latency and throughput of the proposed controller, and confirm these via simulation.

*This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0931843 (ActionWebs), and #1035672 (CSR-)CPS Ptides)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

Categories and Subject Descriptors

B.3.0 [**Memory Structures**]: General

General Terms

Design

Keywords

real-time computing, memory controller, timing predictability, temporal isolation, memory hierarchy

1. INTRODUCTION

Hard real-time embedded systems are an important class of embedded systems in which a violation of timing constraints in any part of the system could result in catastrophic outcomes. Applications of such systems include aircraft flight control, automotive engine control, and nuclear plant control. For the design and certification of such systems, it is necessary to prove that the behaviors these systems implement are guaranteed to meet their timing constraints. Whether or not this is possible and with what effort depends on the *timing predictability* of the software and the underlying execution platform.

In addition to predictable timing, it is increasingly important to achieve *temporal isolation* between distinct real-time functions. One function should not disrupt the timing of another. In particular, in order to reduce implementation cost, there is a shift from *federated architectures*, where each major function is implemented separately on a dedicated execution platform, towards *integrated architectures*, where multiple critical functions are integrated on a single, shared execution platform. For instance, it would be inefficient to provide separate high-capacity off-chip memories for each function if the memory requirements of all functions can be met with a single large memory. When moving to an *integrated architecture*, it is essential to retain the ability to develop and verify functions separately. This is particularly the case when different functions are developed by different entities.

Summarizing, the designer of an execution platform for embedded hard-real time systems faces two challenges:

1. The platform should be *timing predictable* to make the verification of timing constraints possible.
2. Different functions integrated on the platform should be *temporally isolated* to enable independent development and verification.

The integration of several increasingly complex functions in embedded systems yields memory requirements that mandate the use of high-capacity off-chip memories such as dynamic RAM (DRAM). This paper presents a novel DRAM controller that tackles the above two challenges. Predicting DRAM access latencies for conventional memory controllers is challenging for several reasons: the latency of a memory access depends on the history of previous accesses to the memory, which determine whether or not a different row has to be activated. When several applications share the memory, the history of previous accesses to the memory is the result of one of extremely many interleavings of access histories of the different applications. In addition, DRAM cells have to be refreshed periodically. Conventional memory controllers may issue refreshes and block current requests at—from the perspective of an application—unpredictable times.

Previous work by Åkesson et al. [1, 2] and Paolieri et al. [3] achieves timing predictability by dynamically scheduling precomputed sequences of SDRAM commands. While these sequences do not completely eliminate the dependence of timing on access history, they do allow to compute bounds on latency relatively easily.

Our approach significantly improves worst-case access latencies of short transfers compared with this previous work. The main innovation that enables these improvements is that we view a DRAM device not as a single resource to be shared entirely among a set of clients, but as several resources which may be shared among one or more clients individually. We partition the physical address space following the internal structure of the DRAM device, i.e., its ranks and banks. By interleaving accesses to blocks of this partition in a time-triggered way, we eliminate potential contention for shared resources within the device, which could otherwise incur high latencies. The time-triggered nature of the approach also eliminates temporal interference.

Our second innovation is our treatment of refreshes: we defer refreshes to the end of a transfer, and we perform them "manually" through row accesses rather than through the standard refresh mechanism. Both modifications improve access latencies at a slight loss of bandwidth.

The DRAM controller is part of a larger effort to develop PTARM [4], a precision-timed (PRET [5, 6]) architecture, which provides predictable and repeatable timing. PTARM employs a four-stage, four-thread thread-interleaved pipeline. The four resources provided by the memory controller are a perfect match for the four hardware threads of PTARM. We discuss the integration of our controller within PTARM, including the use of direct memory access units.

2. BACKGROUND: DRAM BASICS

We present the basics of DRAM, and the structure of modern DRAM modules. Inherent properties of DRAM and the structure of DRAM modules impose several timing constraints, which all memory controllers must obey. For more details on different DRAM standards, we refer the reader to Jacob et al. [7].

Dynamic RAM Cell.

A DRAM cell consists of a capacitor and a transistor as shown in Figure 1. The charge of the capacitor determines the value of the bit, and by triggering the transistor, one can access the capacitor. However, the capacitor leaks charge over time, and thus, it must be refreshed periodically. According to the JEDEC [8] capacitors must be refreshed every 64 ms or less.

DRAM Array.

A DRAM array contains a two-dimensional array structure of DRAM cells. Accesses to a DRAM array proceed in two phases: row accesses followed by one or more column accesses. A row access moves one of the rows of the DRAM array into the row buffer. As the capacitance of the capacitors in the DRAM cells is low compared with that of the wires connecting them to the row buffer, sense amplifiers are needed to read out their values. In order for the sense amplifiers to read out the value of a DRAM cell, the wires need to be precharged close to the voltage threshold between 0 and 1. Once a row is in the row buffer, columns can be read from and written to quickly. Columns are small sets of consecutive bits within a row.

DRAM Devices.

DRAM arrays form banks in a DRAM device. Figure 1 illustrates a bank's structure, and the location of banks within DRAM devices. Modern DRAM devices have multiple banks, control logic, and I/O mechanisms to read from and write to the data bus as shown in the center of Figure 1. Different banks within a device can be accessed concurrently. This is known as bank-level parallelism. However, the data, command, and address busses are shared among all banks, as is the I/O gating, connecting the banks to the data bus.

A DRAM device receives commands from its memory controller through the command and address busses. The following table lists the four most important commands and their function:

Command	Abbr.	Description
Precharge	PRE	Stores back the contents of the row buffer into the DRAM array, and prepares the sense amplifiers for the next row access.
Row access	RAS	Moves a row from the DRAM array through the sense amplifiers into the row buffer.
Column access	CAS	Overwrites a column in the row buffer or reads a column from the row buffer.
Refresh	REF	Refreshes several[1] rows of the DRAM array. This uses the internal refresh counter to determine which rows to refresh.

To read from or write to the DRAM device, the controller needs to first precharge (PRE) the bank containing the data that is to be read. It can then perform a row access (RAS = row access strobe), followed by one or more column accesses (CAS = column access strobe). Column accesses can be both reads and writes. For higher throughput, column accesses are performed in bursts. The length of these bursts is usually configurable to four or eight words. In a x16-device, columns consist of 16 bits. A burst of length four will thus result in a transfer of 64 bits. To decrease the latency between accesses to different rows, column accesses can be immediately followed by precharge operations, which is known as auto-precharge (aka closed-page policy).

There are two ways of refreshing DRAM cells within the 64 ms timing constraint:

1. Issue a refresh command. This refreshes all banks of the device simultaneously. The DRAM device maintains a refresh counter to step through all of the rows. To refresh every row and thus every DRAM cell every 64 ms, the memory controller has to issue at least 8192 refresh commands in every interval of 64 ms. Earlier devices had exactly 8192 rows per bank. However, recent higher-density devices have up to 65536 rows per bank. As a consequence, for such devices, a refresh command will refresh several rows in each bank, increasing refresh latency considerably.

[1] The number of rows depends on the capacity of the device.

2. Manually refresh rows. The memory controller performs a row access on every row in every bank every 64 ms. This forces the memory controller to issue more commands, and it requires a refresh counter outside of the memory device. Each refresh takes less time because it only accesses one row, but refreshes have to be issued more frequently.

DRAM Modules.

To achieve greater capacity and bandwidth, several DRAM devices are integrated on a memory module. The right side of Figure 1 depicts a high-level view of the dual-ranked dual in-line memory module (DIMM) that the PRET DRAM controller uses. The DIMM has eight DRAM devices that are organized in two ranks of four x16 DRAM devices each. The two ranks share the address and command inputs, and the 64-bit data bus. The chip select input determines which of the two ranks is addressed. DRAM devices within a rank operate in lockstep: they receive the same address and command inputs, and read from or write to the data bus at the same time.

Logically, the four x16 DRAM devices that comprise a rank can be viewed as one x64 DRAM device. This is how we view them for the remainder of this paper. When referring to a bank i in one of the two ranks, we are referring to bank i in each of the four x16 DRAM devices that comprise that rank. A burst of length four results in a transfer of $4 \cdot 16 \cdot 4 = 256$ bits $= 32$ bytes.

Due to the sharing of I/O mechanisms within a device, consecutive accesses to the same rank are more constrained than consecutive accesses to different ranks, which only share the command and address as well as the data bus. We later exploit this subtle difference by restricting consecutive accesses to different ranks to achieve more predictable access latencies. Our controller makes use of a feature from the DDR2 standard known as posted-CAS. Unlike DDR or other previous versions of DRAMs, DDR2 can delay the execution of CAS commands (posted-CAS). After receiving a posted-CAS, DDR2 waits for a user-defined latency, known as the additive latency AL, until sending the CAS to the column decoder. Posted-CAS can be used to resolve command bus contention by sending the posted-CAS earlier than the corresponding CAS needs to be executed. We explain this in more detail in Section 4.1.

DDR2 Timing Constraints.

The internal structure of DRAM modules described above as well as properties of DRAM cells incur a number of timing constraints, which DRAM controllers have to obey. Table 1 gives an overview of timing parameters for a DDR2-400 memory module and brief explanations. These parameters constrain the placement of commands to be send to a DDR2 module. Some of the constraints (t_{RCD}, t_{RP}, t_{RFC}) are solely due to the structure of DRAM banks, which are accessed through sense amplifiers that have to be precharged. Others result from the structure of DRAM banks and DRAM devices: $t_{CL}, t_{WR}, t_{WTR}, t_{WL}$. The four-bank activation window constraint t_{FAW} constrains rapid activation of multiple banks which would result in too high a current draw. The additive latency, t_{AL}, can be set by the user and determines how many cycles after a posted-CAS a CAS is executed.

3. RELATED WORK

The most related work is that of Åkesson et al. [1, 2] and Paolieri et al. [3]. Åkesson et al. introduce Predator, a predictable SDRAM controller. Predator is a hybrid between static and dynamic memory controllers. Instead of dynamically scheduling individual SDRAM commands, Predator's backend dynamically schedules precomputed sequences of SDRAM commands: one for writes, one for reads, one for switching from reads to writes, one for switching from writes to reads, and one for refreshes. These sequences obey all SDRAM timing constraints and have fixed latencies. While they do not completely eliminate the dependence of timing on the access history, they do allow to compute latency bounds relatively easily. To fully utilize the bandwidth DRAM devices provide, the precomputed read and write sequences access every bank of a DRAM module in an interleaved fashion.

To share access to the DRAM, Åkesson et al. propose credit-controller static-priority (CCSP) arbitration [9], which regulates the rates of requests, guarantees a maximum bandwidth, and ensures bounded latency. In contrast to, e.g. TDMA or round robin, CCSP decouples latency from rate. To provide a client with a low latency one does not have to allocate a high rate as well. Still, Predator can also be combined with more common forms of arbitration such as time-division multiple access (TDMA).

The approach of Paolieri et al. [3] called *analyzable memory controller* (short AMC) is very similar to that of Predator. The main difference between AMC and Predator is arbitration. AMC employs a round-robin arbiter, which they argue is more suitable for control-based applications than CCSP, which requires to assign priorities and bandwidth requirements.

The main difference between Predator and AMC on the one hand, and our work on the other, is that the backends of Predator and AMC share the entire memory among all clients by design. Our approach allows to share memory, but, as our analytical evaluation in Section 5 shows, assigning private partitions to clients can significantly improve latency of short transfers. In contrast to Predator and AMC we do not use the standard refresh mechanism, but perform refreshes manually, accessing all of the rows periodically. This also improves latency at a slight loss of bandwidth. When the memory is not shared, we further improve latency by pushing refreshes to the end of a transfer.

Bhat and Mueller [10] eliminate interferences between refreshes and memory accesses of tasks, so that WCET analysis can be performed without having to consider refreshes. Instead of spreading the refreshes over time, they bundle them, and refresh all lines of a DRAM device in a single or few bursts of refreshes. These refresh bursts can then be scheduled in periodic tasks and taken into account during schedulability analysis. This approach can be implemented in software. However, it neither deals with interferences between different tasks in a multi-core system nor does it provide latency guarantees.

Refreshes have earlier been considered in WCET analysis by Atanassov and Puschner [11]. They determine the maximal impact of DRAM refreshes on execution time in the long run. Execution time may increase by approximately 2% for the DRAM device considered in [11]. The approach is limited to *timing compositional* architectures [12].

Bourgade et al. [13] propose a static analysis of the DRAM state based on abstract interpretation. Their analysis tracks, for all program points, which rows may be open at those points. This information is then used to bound memory access times. The analysis of Bourgade et al. is useful and valid as long as the DRAM is not shared by several clients. Another restriction of this approach is that it does not consider refreshes.

4. PRET DRAM CONTROLLER

The memory controller is split into a backend and a frontend. The backend issues commands to the DRAM module, and the frontend connects to the processor. For this section, we specifically refer to a DDR2 667MHz/PC2-5300 memory module operating at 200Mhz, which has a total size of 512MB over two ranks with four banks on each rank. While our discussion of the design of this

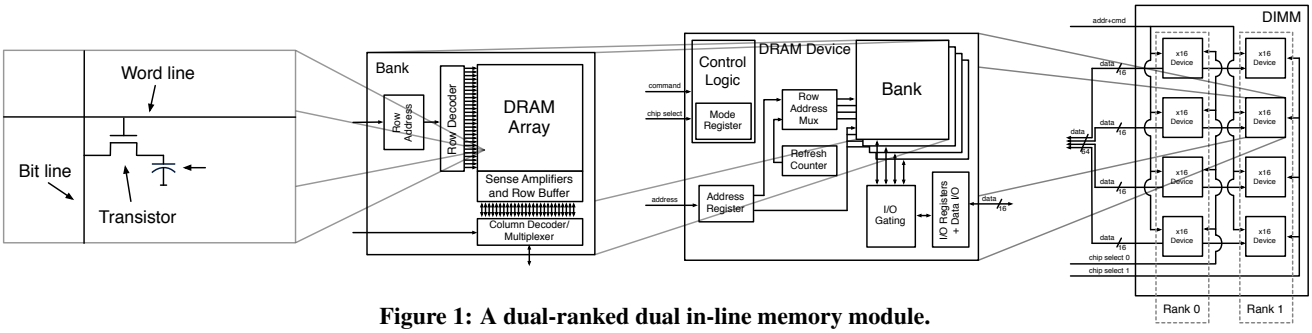

Figure 1: A dual-ranked dual in-line memory module.

Table 1: Overview of DDR2-400 timing parameters at the example of the Qimonda HYS64T64020EM-2.5-B2.

Parameter	Value (in cycles at 200 MHz)	Description
t_{RCD}	3	Row-to-Column delay: time from row activation to first read or write to a column within that row.
t_{CL}	3	Column latency: time between a column access command and the start of data being returned.
t_{WL}	$t_{CL} - 1 = 2$	Write latency: time after write command until first data is available on the bus.
t_{WR}	3	Write recovery time: time between the end of a write data burst and the start of a precharge command.
t_{WTR}	2	Write to read time: time between the end of a write data burst and the start of a column-read command.
t_{RP}	3	Time to precharge the DRAM array before next row activation.
t_{RFC}	21	Refresh cycle time: time interval between a refresh command and a row activation.
t_{FAW}	10	Four-bank activation window: interval in which maximally four banks may be activated.
t_{AL}	set by user	Additive latency: determines how long posted column accesses are delayed.

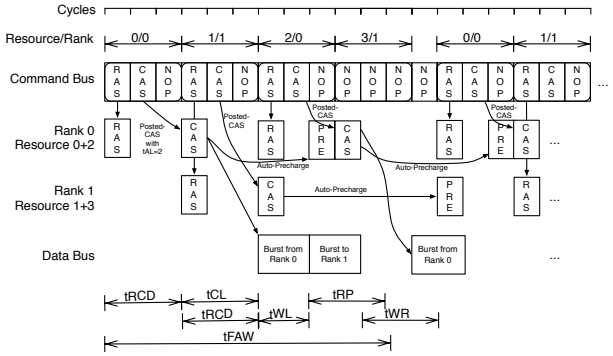

Figure 2: The periodic and pipelined access scheme employed by the backend. In the example, we perform a read from resource 0 (in rank 0), a write to resource 1 (in rank 1), and a read from resource 2 (in rank 0).

DRAM controller is specific to our DDR2 memory module, the key design features are applicable to other modern memory modules.

4.1 DRAM Controller Backend

The backend views the memory device as four independent resources: each resource consisting of two banks within the same rank. By issuing commands to the independent resources in a periodic and pipelined fashion, we exploit bank parallelism and remove interference amongst the resources. This is unlike conventional DRAM controllers that view the entire memory device as one resource. Other partitions of the eight banks would be possible, as long as all of the banks that are part of a resource belong to the same rank of the memory module, and each of the two ranks contains two resources.

Figure 2 shows an example of the following access requests from the frontend: read from resource 0 in rank 0, write to resource 1 in rank 1, and read from resource 2 in rank 0. The controller periodically provides access to the four resources every 13 cycles. In doing so, we exploit bank parallelism for high bandwidth, yet, we avert access patterns that otherwise incur high latency due to the sharing of resources within banks and ranks.

The backend translates each access request into a row access command (RAS), a posted column access command (posted-CAS) or a NOP. We refer to a triple of RAS, CAS and NOP as an access slot. In order to meet row to column latency shown in Table 1, the RAS command and the first CAS command need to be 3 cycles apart. However, we can see from Figure 2 that if we waited for 3 cycles before issuing the CAS to access the first resource, it would conflict with the RAS command for accessing the second resource on the command bus. Instead, we set the additive latency t_{AL} to 2. This way, the posted-CAS results in a CAS two cycles later within the DRAM chip. This is shown in Figure 2 as the posted-CAS appears within its rank 2 cycles after the CAS was issued on the command bus, preserving the pipelined access scheme.

The row access command moves a row into the row buffer. The column access command can be either a read or a write, causing a burst transfer of $8 \cdot 4 = 32$ bytes, which will occupy the data bus for two cycles (as two transfers occur in every cycle). We use a closed-page policy (also known as auto-precharge policy), which causes the accessed row to be immediately precharged after performing the column access (CAS), preparing it for the next row access. If there are no requests for a resource, the backend does not send any commands to the memory module, as is the case for resource 3 in Figure 2.

There is a one cycle offset between the read and write latencies. Given that requests may alternate between reads and writes, the controller inserts a NOP between any two consecutive requests. This avoids a collision on the data bus between reads and writes. By alternating between ranks, no two adjacent accesses go to the same rank. This satisfies the write-to-read timing constraint t_{WTR} incurred by the sharing of I/O gating within ranks. In addition, we satisfy the four-bank activation window constraint because within any window of size t_{FAW} we activate at most four banks due to the periodic access scheme.

With the closed-page policy, in case of a write, we need 13 cycles to access the row, perform a burst access, and precharge the bank to prepare for the next row access. This is the reason for adding a NOP after four access slots: to increase the distance between two access slots belonging to the same resource from 12 to 13 cycles. The backend does not issue any refresh commands to the memory

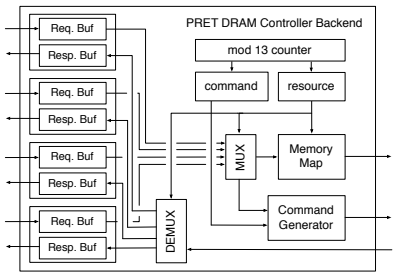

Figure 3: Sketch of implementation of the backend.

module. Instead, it relies on the frontend to refresh the DRAM cells using regular row accesses.

Implementation and Interface to Frontend.

For each resource, the backend contains single-place request and response buffers, written to and read from by the frontend. A request consist of an access type (read or write), a logical address, and in the case of a write, the data to be written. Requests are serviced at the granularity of bursts, i.e. 32 bytes in case of burst length 4 and 64 bytes in case of burst length 8.

The access scheme can be implemented relatively easily by a modulo-13 counter (as the backend has period 13) and two simple combinational circuits, as schematically illustrated in Figure 3. The "resource" circuit determines which request buffer to use when generating the command and address to be sent to the memory module. The logical addresses are mapped to physical addresses based on the output of the "resource" circuit, which determines the appropriate bank. Similarly, the output of the "command" and "resource" circuits are used to determine when and from which buffer to send data on the data bus in case of writes, and, in case of reads, which response buffer to fill.

Longer Bursts for Improved Bandwidth.

Depending on the application, bandwidth might be more important than latency. Bandwidth can be improved by increasing the burst length from 4 to 8. Extending the proposed access scheme to a burst length of 8 is straightforward with the insertion of two additional NOP commands after each request to account for the extra two cycles of data being transfered on the data bus. In this case, the access slot latency for each request is increased from three to five to include the extra two NOP commands, and data will be transferred in four out of five cycles rather than in two out of three. Then, of course, latency of transfers of size less than or equal to 32 bytes increases, but the latency of large transfers decreases and higher bandwidth is achieved.

4.2 DRAM Controller Frontend

In this section, we discuss our integration of the backend within the PTARM PRET architecture [4]. We also discuss how the PRET DRAM controller could be integrated into other predictable architectures, such as those proposed by the MERASA [14], PREDATOR [12], JOP [15], or CoMPSoC [16] projects, which require predictable and composable memory performance.

4.2.1 Integration with the PTARM Architecture

PTARM [4], a PRET machine [5], is a thread-interleaved implementation of the ARM instruction set. Thread-interleaved processors preserve the benefit of a multi-threaded architecture – increased throughput, but use a predictable fine-grained thread-scheduling policy – round robin. If there is the same number of hardware threads as there are pipeline stages, at any point in time, each stage of the pipeline is occupied by a different hardware thread; there are

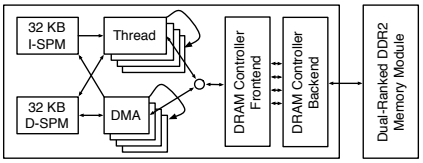

Figure 4: Integration of PTARM core with DMA units, PRET memory controller and dual-ranked DIMM.

no dependencies between pipeline stages, and the execution time of each hardware thread is independent of all others. PTARM has four pipeline stages and four hardware threads. Each hardware thread has access to an instruction scratchpad and a data scratchpad. The scratchpads provide single-cycle access latencies to the threads. The two scratchpads are shared among the four threads, allowing for shared memory communication among the threads. However, due to the thread-interleaving, only one thread can access the scratchpad at any time. Each hardware thread is also equipped with a direct memory access (DMA) unit, which can perform bulk transfers between the two scratchpads and the DRAM. Both scratchpads are dual-ported, allowing a DMA unit to access the scratchpads in the same cycles as its corresponding hardware thread. In our implementation of thread-interleaving, if one thread is stalled waiting for a memory access, the other threads are unaffected and continue to execute normally.

The four resources provided by the backend are a perfect match for the four hardware threads in the PTARM thread-interleaved pipeline. We assign exclusive access to one of the four resources to each thread. In contrast to conventional memory architectures, in which the processor interacts with DRAM only by filling and writing back cache lines, there are two ways the threads can interact with the DRAM in our design. First, threads can initiate DMA transfers to transfer bulk data to and from the scratchpad. Second, since the scratchpad and DRAM are assigned distinct memory regions, threads can also directly access the DRAM through load and store instructions.

Whenever a thread initiates a DMA transfer, it passes access to the DRAM to its DMA unit, which returns access once it has finished the transfer. During the time of the transfer, the thread can continue processing and accessing the two scratchpads. If at any point the thread tries to access the DRAM, it will be blocked until the DMA transfer has been completed. Similarly, accesses to the region of the scratchpad which are being transferred from or to will stall the hardware thread[2]. Figure 4 shows a block diagram of PTARM including the PRET DRAM controller backend and the memory module. The purpose of the frontend is to route requests to the right request buffer in the backend and to insert a sufficient amount of refresh commands, which we will discuss in more detail.

When threads directly access the DRAM through load (read) and store (write) instructions, the memory requests are issued directly from the pipeline. Figure 6, which we will later use to derive the read latency, illustrates the stages of the execution of a read instruction in the pipeline. At the end of the memory stage, a request is put into the request buffer of the backend. Depending on the alignment of the pipeline and the backend, it takes a varying number of cycles until the backend generates corresponding commands to be sent to the DRAM module. After the read has been performed by the DRAM and has been put into the response buffer, again, depending on the alignment of the pipeline and the backend, it takes a varying number of cycles for the pipeline to reach

[2]This does not affect the execution of any of the other hardware threads.

the write-back stage of the corresponding hardware thread. Unlike the thread-interleaved pipeline, the DMA units are not pipelined, which implies that there are no "alignment losses": the DMA units can fully utilize the bandwidth provided by the backend.

Store Buffer.

Stores are fundamentally different from loads in that a hardware thread does not have to wait until the store has been performed in memory. By adding a single-place store buffer to the frontend, we can usually hide the store latency from the pipeline. Using the store buffer, stores which are not preceded by other stores can be performed in a single thread cycle. By *thread cycle*, we denote the time it takes for an instruction to pass through the thread-interleaved pipeline. Other stores may take two thread cycles to execute. A bigger store buffer would be able to hide latencies of successive stores at the expense of increased complexity in timing analysis.

Scheduling of Refreshes.

DRAM cells need to be refreshed at least every 64 ms. A refresh can either be performed by a hardware refresh command, which may refresh several rows of a device at once[3], or by performing individual row accesses "manually". We opt to do the latter. This has the advantage that a single row access takes less time than the execution of a hardware refresh command, thereby improving worst-case latency, particularly for small transfers. The disadvantage, on the other hand, is that more manual row accesses have to be performed, incurring a slight hit in bandwidth.

In our device, each bank consists of 8192 rows. Thus, a row has to be refreshed every $64ms/8192 = 7.8125\mu s$. At a clock rate of 200 MHz of the memory controller, this corresponds to $7.8125\mu s \cdot (200cycles/\mu s) = 1562.5$ cycles. Since each resource contains two banks, we need to perform two refreshes every 1562.5 cycles, or one every 781.25 cycles. One round of access is 13 cycles at burst length 4, and includes the access slots to each resource plus a nop command. So we schedule a refresh every $\lfloor 781.25/13 \rfloor^{th} = 60^{th}$ round of the backend. This way, we are scheduling refreshes slightly more often than necessary. Scheduling a refresh every $60 \cdot 13$ cycles means that every row, and thus every DRAM cell, is refreshed every $60 \cdot 13\ cycles \cdot 8192 \cdot 2/(200000\ cycles/ms) \leq 63.90ms$. We are thus flexible to push back any of these refreshes individually by up to $0.1ms = 20000$ cycles without violating the refreshing requirement.

We make use of this flexibility for loads from the pipeline and when performing DMA transfers: if a load would coincide with a scheduled refresh, we push back the refresh to the next slot. Similarly, we skip the first refresh during a DMA transfer and schedule an additional one at the end of the transfer. This pushes back the refresh of a particular row by at most $60 \cdot 13$ cycles. More sophisticated schemes would be possible, however, we believe their benefit would be slim. Following this approach, two latencies can be associated with a DMA transfer:

1. The time from initiating the DMA transfer until the data has been transferred, and is, e.g., available in the data scratchpad.
2. The time from initiating the DMA transfer until the thread-interleaved pipeline regains access to the DRAM.

Our conjecture is that latency 1 is usually more important than latency 2. Furthermore, our approach does not deteriorate latency 2. For loads sent from the pipeline, the pushed back refreshes become invisible: as the pipeline is waiting for the data to be returned and takes some time to reach the memory stage of the next instruction,

[3]Internally, this still results in several consecutive row accesses.

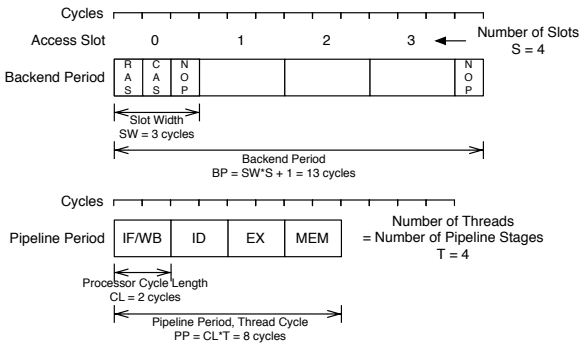

Figure 5: Terms related to the backend and the PTARM thread-interleaved pipeline.

it is not able to use successive access slots of the backend, and thus it is unable to observe the refresh at all. With this refresh scheme, refreshes do not affect the latencies of load/store instructions, and the refreshes scheduled within DMA transfers are predictable so the latency effects of the refresh can be easily analyzed.

4.2.2 Integration with Other Multi-Core Processors

Several recent projects strive do develop predictable multi-core architectures [14, 12, 15, 16]. These could potentially profit from using the proposed DRAM controller. The frontend described in the previous section makes use of specific characteristics of the PTARM architecture. However, when integrating the backend with other architectures, we cannot rely on these characteristics. A particular challenge to address is that most multi-core processors use DRAM to share data, while local scratchpads or caches are private. This can be achieved by sharing the four resources provided by the backend within the frontend. A particularly simple approach would first combine the four resources into one: an access to the single resource would simply result in four smaller accesses to the resources of the backend. This single resource could then be shared among the different cores of a multi-core architecture using predictable arbitration mechanisms such as Round-Robin or CCSP [9] or predictable and composable ones like time-division multiple access (TDMA). However, sharing the DRAM resources comes at the cost of increased latency. We investigate this cost in Section 5.3.

5. ANALYTICAL EVALUATION

The purpose of this section is twofold: to derive latency and bandwidth guarantees for our memory controller for different possible configurations, and to compare our controller in terms of latency and bandwidth with Predator and AMC.

Within PTARM, we either access the DRAM directly through loads and stores, or through DMA transfers. We proceed to derive best- and worst-case latencies for these two types of accesses.

5.1 Terms and Parameters

Figure 5 introduces parameters we will use in our latency derivations. The first part of the figure concerns the backend. A *backend period* (BP) consists of a number of access slots, denoted by S. In our current design $S = 4$ corresponding to the four resources exposed by the backend. Each access slot requires *slot width* SW cycles. In this section, "cycle" always refers to cycles of the memory controller. At burst length 4, each access slot consists of a RAS, a CAS, and a NOP, so $SW = 3 = BL/2 + 1$. In case of burst length 8, $SW = 5 = BL/2 + 1$. Thus, the length of a backend period, denoted BP is the product of the number of access slots S and the slot width SW plus the number of NOPs inserted after every S access slots. As we have seen earlier, we have to insert one

NOP to satisfy timing constraints at burst length 4. At burst length 8, this is not necessary.

The second part of Figure 5 concerns the thread-interleaved pipeline. Since the frequency of the thread-interleaved pipeline differs from the DRAM controller, a cycle of the pipeline can take several cycles of the DRAM controller. We denote the *processor cycle length* by CL, which is 2 in our current design, as the thread-interleaved pipeline is running at 100 MHz. Note that CL does not have to be an integer. The number of cycles it takes to execute one instruction of a thread is CL multiplied with the number of pipeline stages, which in our case is the same as the number of threads T. We call this, a *thread cycle* or *pipeline period* $PP = CL \cdot T$. In the current design of PTARM, $T = 4$, and so $PP = 8$.

5.2 Latency of Loads and Stores in PTARM

To determine the load latency, consider Figure 6, which decomposes a load into three parts. At the end of its memory stage, the pipeline stores its load command in the command buffer of the memory controller.

Depending on the alignment of the backend and the pipeline, the memory controller will send out corresponding RAS and CAS when its access slot is active. Let BEL denote the *backend latency*, i.e., the time a command spends in the command buffer before the corresponding RAS command has been sent out. BEL can be anywhere between 1 cycle and BP cycles[4].

After the RAS command has been sent to the DDR2 module, it takes *DRAM read latency*, DRL, cycles until the requested data is provided on the data bus. For a given memory module, and our access scheme, DRL is constant, in our case $10 + \frac{BL}{2}$ cycles. However, the requesting thread can only receive this data in its write-back stage. We denote the time that the data is buffered in the memory controller before it is received by the processor by *thread alignment latency* TAL. The *read latency* RL is then the sum of the backend latency, the DRAM read latency, and the thread alignment latency, i.e. $RL = BEL + DRL + TAL$.

We can eliminate the parameter TAL from this equation, as it follows from the backend latency: the write-back stage in which the data is received occurs $CL + k \cdot PP$ cycles after the memory stage issued the request for the smallest k such that $CL + k \cdot PP \geq BEL + DRL$. Here k is the number of thread cycles between sending the read request and receiving the data. This smallest k is determined by $\left\lceil \frac{BEL+DRL-CL}{PP} \right\rceil$. So, the read latency RL is determined as follows:

$$RL = \left\lceil \frac{BEL + DRL - CL}{PP} \right\rceil \cdot PP + CL. \qquad (1)$$

Based on RL we determine the number of thread cycles RTC that a read takes, where PP is the length of a *thread cycle* (aka *pipeline period*):

$$RTC = \left\lceil \frac{RL}{PP} \right\rceil = \left\lceil \frac{BEL + DRL - CL}{PP} \right\rceil + 1. \qquad (2)$$

The only variable parameter left in these equations is the *backend latency* BEL. BEL depends on the alignment of the backend and the thread-interleaved pipeline. BEL may vary from one memory access of a thread to the next. Whether it varies and how depends on the backend period BP and the pipeline period PP. If BP and PP are rational numbers, they have a *least common multiple* lcm, the system's hyperperiod. Alignments between the backend and the thread-interleaved pipeline recur periodically every $\mathrm{lcm}(PP, BP)$ cycles, which corresponds to $\frac{\mathrm{lcm}(PP,BP)}{PP}$ thread cycles. If the two

[4]Note that BEL is always at least 1 as it includes the RAS command.

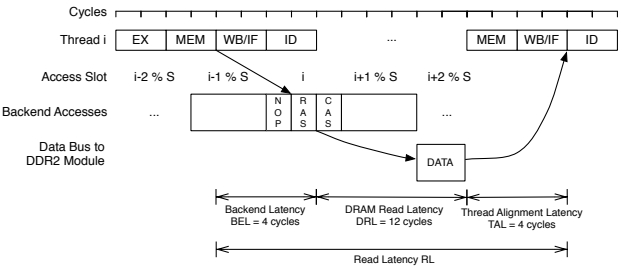

Figure 6: Example of a load operation by hardware thread i in the thread-interleaved pipeline.

periods do not have a common multiple, BEL will not be periodic in general. Let $BEL(i)$ denote the BEL in thread cycle i, then

$$BEL(i) = BEL(0) - (i \cdot PP) \bmod BP. \qquad (3)$$

Our current version of PTARM contains four threads, so $T = 4$, at a clock frequency of 100 MHz, so $CL = 2$, and thus $PP = 8$. For burst length 4, $BP = 13$. Then $BEL(i)$ periodically cycles through $\frac{\mathrm{lcm}(PP,BP)}{PP} = \frac{104}{8} = 13$ different values. If the thread-interleaved pipeline and the backend are activated simultaneously, then $BEL(0)$ will be 1 for the first thread, 2 for the second thread, and so on. For thread 0, the backend latency then cycles through 1, 6, 11, 3, 8, 13, 5, 10, 2, 7, 12, 4, 8 periodically.

We can plug these values into the read latency equations. Assuming additionally $DRL = 10 + \frac{BL}{2} = 12$, we get $RTC = \left\lceil \frac{BEL+DRL-CL}{PP} \right\rceil + 1 = \left\lceil \frac{BEL+10}{8} \right\rceil + 1$. As the backend latency BEL is at most $BP = 13$, RTC is either $\left\lceil \frac{1+10}{8} \right\rceil + 1 = 3$ or $\left\lceil \frac{13+10}{8} \right\rceil + 1 = 4$ thread cycles. For burst length 8, RTC varies between 3 and 5 thread cycles.

For comparison, Predator [2], which has been designed for high bandwidth, takes 53 cycles to service the smallest possible request at burst length 4 and 70 cycles at burst length 8, which translates to 8 and 10 thread cycles, respectively.

By delaying refreshes until after a load has been performed, they do not have an impact on load latency. As the data is returned shortly before (burst length 8) or even at the same time (burst length 4) as the next access slot comes around a thread is not able to utilize two consecutive access slots through loads.

Stores are fundamentally different from loads, as the pipeline does not have to wait for a store to finish before resuming execution. By buffering stores, we can shield the pipeline from store latencies. However, in the long run, with finite buffers, we are still limited by the sustainable bandwidth of the memory: The memory is able to process a store every BP cycles. Furthermore refreshes are scheduled in every RFP^{th} access slot, where RFP denotes the *refresh period*. In the long run, a store thus requires $\frac{BP}{PP} \cdot \frac{RFP}{RFP-1}$ thread cycles. In the case of burst length 4, this is $\frac{13}{8} \cdot \frac{60}{59} = 1.65$, in case of burst length 8, this is $\frac{20}{8} \cdot \frac{39}{38} = 2.57$. Stores, which are not surrounded by other stores can be handled within a single thread cycle. Other stores, may take 2 or 3 cycles depending on the burst length and the state of the store buffer. By introducing a larger store buffer, we would be able to to hide the latency of a bounded number of consecutive stores, at the expense of higher implementation cost and increased complexity in timing analysis.

5.3 Latency of DMA Transfers

We anticipate that the main way of transferring data to and from the DRAM will be through the DMA mechanism, which can fully utilize the bandwidth provided by the memory. Unlike loads and stores from the pipeline, DMA transfers can be of varying sizes.

5.3.1 Derivation of Worst-case DMA Latencies

To carry out a transfer of x bytes, a DMA unit needs to send $\lceil \frac{x}{BL \cdot 8} \rceil$ requests to the backend. It has to wait up to BEL cycles to send the first request, then it can send requests every $BP = 4 \cdot (1 + \frac{BL}{2})$ cycles. BEL is at most BP. After sending the last request to the backend, it takes $DRL = 10 + \frac{BL}{2}$ cycles for the resulting burst transfer to finish. Thus, the latency $DL(x)$ of a transfer of x bytes from the DRAM in cycles of the memory controller is

$$DL(x) = BEL + BP \cdot \left(\left\lceil \frac{x}{BL \cdot 8} \right\rceil - 1 \right) + DRL \quad (4)$$

$$\leq (4 + 2 \cdot BL) \cdot \left\lceil \frac{x}{BL \cdot 8} \right\rceil + 10 + \frac{BL}{2}. \quad (5)$$

This equation, however, does not consider refreshes yet. As noted before, we associate two latencies with a DMA transfer:

1. The time $DL^{\overline{r}}(x)$ from initiating the DMA transfer until the data has been transferred, and is, e.g., available in the data scratchpad. The superscript $\overline{r}$ indicates that $DL^{\overline{r}}(x)$ does *not* include the final refresh.
2. The time $DL^{r}(x)$ from initiating the DMA transfer until the thread-interleaved pipeline regains access to the DRAM. The superscript r indicates that $DL^{r}(x)$ includes the final refresh.

One could further distinguish between transfers from DRAM to scratchpad and from scratchpad to DRAM. Due to space constraints, we only consider the former, which incurs higher latencies. $DL^{\overline{r}}(x)$ can be computed from $DL(x)$ by adding latency incurred by refreshes beyond the first one, which will be accounted for in $DL^{r}(x)$:

$$DL^{\overline{r}}(x) = DL(x) + BP \left\lceil \frac{\lceil \frac{x}{BL \cdot 8} \rceil}{RFP - 1} - 1 \right\rceil \quad (6)$$

$$= DL(x) + (4 + 2 \cdot BL) \left\lceil \frac{\lceil \frac{x}{BL \cdot 8} \rceil}{RFP - 1} - 1 \right\rceil \quad (7)$$

where RFP is the refresh period. At burst length 4, $RFP = 60$, at burst length 8, $RFP = 39$. $DL^{r}(x)$ is simply $DL^{\overline{r}}(x) + BP$.

In order to assess the value of privatization, we also determine latencies for a scenario in which the four resources of the backend are shared among four clients in a round-robin fashion. These four clients could be the four threads of the PTARM or four cores in a multi-core processor. This shall also indicate whether the PRET DRAM controller is a viable option in such a scenario.

By $DL_{n,s}(x)$ we denote the latency of a transfer of size x, where the DMA unit has access to n resources, which are each shared among s clients. A transfer of size x will then be split up into n transfers of size x/n. Due to the sharing of the resources, only every s^{th} access slot is available in each resource.

$$DL_{n,s}(x) = s \cdot BP \cdot \left\lceil \frac{x}{n \cdot BL \cdot 8} \right\rceil + DRL \quad (8)$$

$$= s \cdot (4 + 2 \cdot BL) \cdot \left\lceil \frac{x}{n \cdot BL \cdot 8} \right\rceil + \frac{BL}{2} + 9. \quad (9)$$

For space reasons, we limit our analysis to the second of the two latencies associated with a DMA transfer, which is derived similarly to the non-shared case:

$$DL^{r}_{n,s}(x) = DL_{n,s}(x) + BP \left\lceil \frac{s \cdot \lceil \frac{x}{n \cdot BL \cdot 8} \rceil}{RFP - 1} \right\rceil \quad (10)$$

$$= DL_{n,s}(x) + (4 + 2 \cdot BL) \left\lceil \frac{s \cdot \lceil \frac{x}{n \cdot BL \cdot 8} \rceil}{RFP - 1} \right\rceil . \quad (11)$$

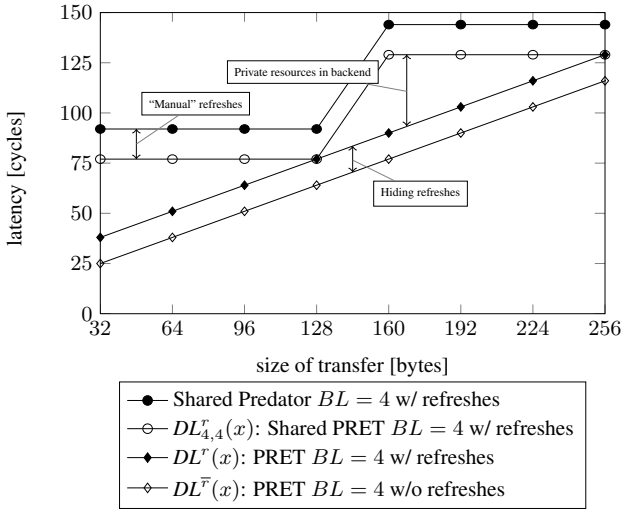

Figure 7: Latencies for small request sizes up to 256 bytes under Predator and PRET at burst length 4. In this, and all of the following figures, one cycle corresponds to 5 ns.

5.3.2 Analysis of Worst-case DMA Latencies

For comparison, we have also determined access latencies for Predator based on Åkesson's dissertation [2]. Figure 7 shows access latencies of PRET and Predator for transfers up to 256 bytes, as they frequently occur in fine-grained scratchpad allocation code, or when filling cache lines. We compare four scenarios involving PRET and Predator:

1. $DL^{\overline{r}}(x)$: Latencies of transfers using one of the four resources at burst length 4, excluding the cost of a final refresh.

2. $DL^{r}(x)$: Latencies of transfers using one of the four resources at burst length 4, including the cost of all refreshes.

3. $DL^{r}_{4,4}(x)$: Latencies of transfers using all of the four resources at burst length 4 shared among four clients (using round-robin arbitration), including the cost of all refreshes.

4. Latencies of transfers using Predator at burst length 4 shared among four clients (using round-robin arbitration), including the cost of all refreshes.

Hiding refreshes (Scenario 1 vs Scenario 2) saves $BP = 13$ cycles in all cases. The benefit of private resources can be seen comparing Scenario 2 with Scenario 3. When sharing all banks, the minimum transfer size is 128 bytes (one burst of 32 bytes to each of the four resources). For transfer sizes that are not multiples of this size, private resources reduce latency significantly. The most extreme case is that of a 32-byte transfer where latency is reduced from 77 to 38 cycles. The slight advantage of shared PRET (Scenario 3) compared with shared Predator (Scenario 4) can mostly be explained by the manual refresh mechanism employed in PRET.

For larger transfers, the bandwidth provided by the memory controller becomes more important, and private DRAM resources are less beneficial. This is illustrated in Figure 8. For both burst length 4 and 8, PRET and Predator show very similar latencies. Predator's slightly flatter slope is due to fewer read/write switches and the use of the standard refresh mechanism, which adversely affects latencies of small transfers. For 2 KB transfers, burst length 8 reduces latency by approximately 22% compared with burst length 4.

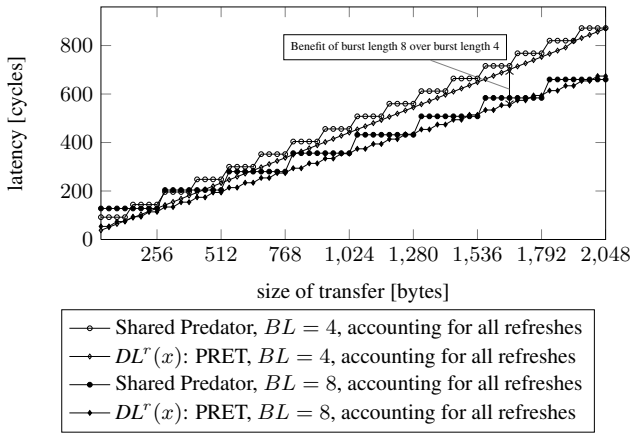

Figure 8: Latencies of Predator and PRET for request sizes up to 2KB under burst lengths 4 and 8.

5.4 Bandwidth

We describe the peak bandwidth achieved by the PRET DRAM controller. In the case of the burst length being 4, disregarding refreshes, we send out four CAS commands every 13 cycles. Each CAS results in a transfer of a burst of size $8 \cdot 4 = 32$ bytes over the period of two cycles[5]. The memory controller and the data bus are running at a frequency of 200 MHz. So, disregarding refreshes the controller would provide a bandwidth of 200 MHz$\cdot \frac{4}{13} \cdot 32$ bytes $\approx$ 1.969GB/s. We issue a refresh command in every 60^{th} slot. This reduces the available bandwidth to $\frac{59}{60} \cdot 1.969$GB/s ≈ 1.936GB/s, which are 60.5% of the data bus bandwidth.

For burst length 8, we transfer $8 \cdot 8 = 64$ bytes every five cycles and perform a refresh in every 39^{th} slot, resulting in an available bandwidth of 200MHz $\cdot \frac{38}{39} \cdot \frac{1}{5} \cdot 64$ bytes ≈ 2.494GB/s, or 77.95% of the data bus bandwidth.

6. EXPERIMENTAL EVALUATION

We present experimental results to verify that the design of the PRET DRAM controller honors the derived analytical bounds. We have implemented the PRET DRAM controller, and compare it via simulation with a conventional DRAM controller. We use the PTARM simulator[6] and extend it to interface with both memory controllers to run synthetic benchmarks that simulate memory activity. The PTARM simulator is a C++ simulator that simulates the PRET architecture with four hardware threads running through a thread-interleaved pipeline. We use a C++ wrapper around the DRAMSim2 simulator [17] to simulate memory access latencies from a conventional DRAM controller. A first-come, first-served queuing scheme is used to queue up memory requests to the DRAMSim2 simulator. The PRET DRAM controller was also written in C++ based on the description in Section 4. The benchmarks we use are all written in C, and compiled using the GNU ARM cross compiler. The DMA transfer latencies that are measured begin when the DMA unit issues its first request and end when the last request from the DMA unit is completed.

6.1 Experimental Results

We setup our experiment to show the effects of interference on memory access latency for both memory controllers. We first setup our main thread to run different programs that initiate fixed-size DMA transfers (256, 512, 1024, 2048 and 4096 bytes) at random intervals. The DMA latencies of the main thread is what is measured and shown in Figure 9 and Figure 10. To introduce interference within the system, we run a combination of two programs on the other hardware threads in PTARM simulator. The first program continuously issues DMA requests of large size (4096 bytes) in order to fully utilize the memory bandwidth. The second program utilizes half the memory bandwidth by issuing DMA requests of size 4096 bytes half as frequently as the first program. In Figure 9, we define thread occupancy on the x-axis as the memory bandwidth occupied by the combination of all threads. 0.5 means we have one thread running the second program along side the main thread. 1.0 means we have one thread running the first program along side the main thread. 1.5 means we have one thread running the first program, one thread running the second program, and both threads are running along side the main thread, and so on. 3 is the maximum we can achieve because the PTARM simulator has a total of four hardware threads (the main thread occupies one of the four). We measured the latency of each fixed size transfer for the main thread to observe the transfer latency in the presence of interference from memory requests by other threads.

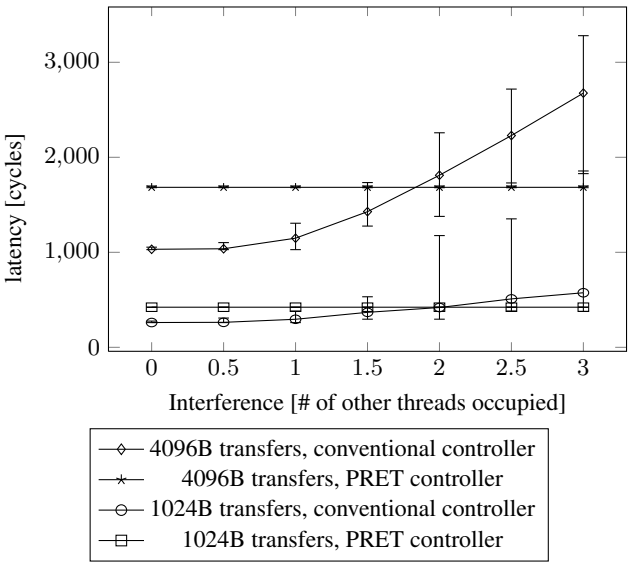

Figure 9: Latencies of conventional and PRET memory controller with varying interference from other threads.

In Figure 9, we show measurements taken from two different DMA transfer sizes, 1024 and 4096 bytes. The marks in the figure show the average latency measured over 1000 iterations. The error bars above and below the marks show the worst-case and best-case latencies of each transfer size over the same 1000 iterations. In both cases, without any interference, the conventional DRAM controller provides better access latencies. This is because without any interference, the conventional DRAM controller can often exploit row locality and service requests immediately. The PRET DRAM controller on the other hand uses the periodic pipelined access scheme, thus even though no other threads are accessing memory, the memory requests still need to wait for their slot to get access to the DRAM. However, as interference is gradually introduced, we observe increases in latency for the conventional DRAM controller. This could be caused by the first-come, first-served buffer, or by the internal queueing and handling of requests by DRAMSim2. The PRET DRAM controller however is unaffected by the interference created by the other threads. In fact, the latency values that were measured from the PRET DRAM controller remain the

[5]In double-data rate (DDR) memory two transfers are performed per clock cycle.

[6]The PTARM simulator is available for download at `http://chess.eecs.berkeley.edu/pret/release/ptarm`.

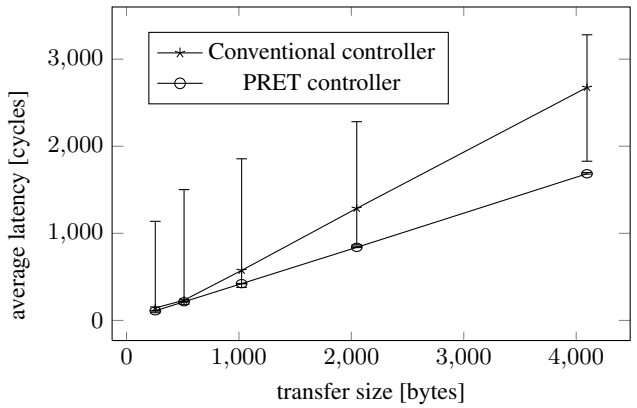

Figure 10: Latencies of conventional and PRET memory controller with maximum load by interfering threads and varying transfer size.

same under all different thread occupancies. This demonstrates the temporal isolation achieved by the PRET DRAM controller. Any timing analysis on the memory latency for one thread only needs to be done in the context of that thread. We also see the range of memory latencies for the conventional DRAM controller increase as the interference increases. But the range of access latencies for the PRET DRAM controller not only remains the same throughout, but is almost negligible for both transfer sizes[7]. This shows the predictable nature of the PRET DRAM controller.

In Figure 10 we show the memory latencies under full load (thread occupancy of 3) for different transfer sizes. This figure shows that under maximum interference from the other hardware threads, the PRET DRAM controller is less affected by interference even as transfer sizes increase. More importantly, when we compare the numbers from Figure 10 to Figure 8, we confirm that the theoretical bandwidth calculations hold even under maximum bandwidth stress from the other threads.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented a DRAM controller design that is predictable with significantly reduced worst-case access latencies. Our approach views the DRAM device as multiple independent resources that are accessed in a periodic pipelined fashion. This eliminates contention for shared resources within the device to provide temporally predictable and isolated memory access latencies. We refresh the DRAM through row accesses instead of standard refreshes. This results in improved worst-case access latency at a slight loss of bandwidth. Latency bounds for our memory controller, determined analytically and confirmed through simulation, show that our controller is both timing predictable and provides temporal isolation for memory accesses from different resources.

Thought-provoking challenges remain in the development of an efficient, yet predictable memory hierarchy. In conventional multicore architectures, local memories such as caches or scratchpads are private, while access to the DRAM is shared. However, in the thread-interleaved PTARM, the instruction and data scratchpad memories are shared, while access to the DRAM is not. We have demonstrated the advantages of privatizing parts of the DRAM for worst-case latency. It will be interesting to explore the consequences of the inverted sharing structure on the programming model.

We envision adding instructions to the PTARM that allow threads to pass ownership of DRAM resources to other threads. This would, for instance, allow for extremely efficient double-buffering implementations. We also plan to develop new scratchpad allocation techniques, which use the PTARM's DMA units to hide memory latencies, and which take into account the transfer-size dependent latency bounds derived in this paper.

[7]The range (worst-case latency - best-case latency) was approximately 90ns for 4096 bytes transfers and approximately 20ns for 1024 byte transfers.

8. REFERENCES

[1] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable SDRAM memory controller," in *CODES+ISSS*. ACM, 2007, pp. 251–256.

[2] B. Akesson, "Predictable and composable system-on-chip memory controllers," Ph.D. dissertation, Eindhoven University of Technology, Feb. 2010.

[3] M. Paolieri, E. Quiñones, F. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time CMPs," *IEEE Embedded Systems Letters*, vol. 1, no. 4, pp. 86–90, 2010.

[4] I. Liu, J. Reineke, and E. A. Lee, "A PRET architecture supporting concurrent programs with composable timing properties," in *44th Asilomar Conference on Signals, Systems, and Computers*, November 2010.

[5] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *DAC*. New York, NY, USA: ACM, 2007, pp. 264–265.

[6] D. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *DAC*. ACM, June 2011.

[7] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann Publishers, September 2007.

[8] JEDEC, *DDR2 SDRAM SPECIFICATION JESD79-2E.*, 2008.

[9] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in *RTCSA*, Aug. 2008, pp. 3 –14.

[10] B. Bhat and F. Mueller, "Making DRAM refresh predictable," in *ECRTS*, 2010, pp. 145–154.

[11] P. Atanassov and P. Puschner, "Impact of DRAM refresh on the execution time of real-time tasks," in *Proc. IEEE International Workshop on Application of Reliable Computing and Communication*, Dec. 2001, pp. 29–34.

[12] R. Wilhelm *et al.*, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE TCAD*, vol. 28, no. 7, pp. 966–978, 2009.

[13] R. Bourgade, C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Accurate analysis of memory latencies for WCET estimation," in *RTNS*, Oct. 2008.

[14] T. Ungerer *et al.*, "MERASA: Multi-core execution of hard real-time applications supporting analysability," *IEEE Micro*, vol. 99, 2010.

[15] M. Schoeberl, "A java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54, no. 1-2, pp. 265 – 286, 2008.

[16] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM TODAES*, vol. 14, no. 1, pp. 1–24, 2009.

[17] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, Jan. 2011.