PROGRAMMABLE DDRX CONTROLLERS

MAKING MODERN MEMORY CONTROLLERS PROGRAMMABLE IMPROVES THEIR VERSATILITY AND EFFICIENCY. HOWEVER, THE STRINGENT LATENCY AND THROUGHPUT REQUIREMENTS OF MODERN DDRX (DOUBLE DATA RATE MEMORY INTERFACE TECHNOLOGY) DEVICES HAVE RENDERED SUCH PROGRAMMABILITY LARGELY IMPRACTICAL, CONFINING DDRX CONTROLLERS TO FIXED-FUNCTION HARDWARE. PARDIS IS THE FIRST PROGRAMMABLE MEMORY CONTROLLER THAT CAN MEET THESE CHALLENGES AND THUS SATISFY THE PERFORMANCE REQUIREMENTS OF A HIGH-SPEED DDRX INTERFACE.

Mahdi Nazm Bojnordi
Engin Ipek
University of Rochester

The off-chip memory subsystem is a significant performance, power, and quality-of-service (QoS) bottleneck in modern computers, necessitating a high-performance memory controller that can overcome DRAM (dynamic random-access memory) timing and resource constraints by orchestrating data movement between the processor and main memory. Contemporary DDRx (double data rate memory interface technology) memory controllers implement sophisticated address mapping, command scheduling, power management, and refresh algorithms to maximize system throughput and minimize DRAM energy, while ensuring that system-level QoS targets and real-time deadlines are met. The conflicting requirements imposed by this multiobjective optimization, compounded by diversity in both workload and memory system characteristics, make high-performance memory controller design a significant challenge.

A promising way of improving the versatility and efficiency of a memory controller is to make it programmable—a proven technique that has seen wide use in other control tasks ranging from direct memory access (DMA) scheduling[1,2] to NAND flash and directory control.[3-9] In these and other architectural control problems, programmability allows the processor designers to customize the controller on the basis of system requirements and performance objectives, perform in-field firmware updates to the controller, and set up application-specific control policies. Unfortunately, the stringent latency and throughput requirements of modern DDRx devices have rendered such programmability largely impractical, confining DDRx controllers to fixed-function hardware. As a result, contemporary memory controllers are invariably confined to implementing DRAM control policies in hardwired, fixed-function hardware blocks.

Pardis (programmable architecture for the DDRx interfacing standards) is the first programmable memory controller that provides sufficiently high performance to make the firmware implementation of DDRx control policies practical.[10] Pardis divides the tasks associated with high-performance DRAM control among a request processor, a transaction processor, and dedicated command logic. The request and transaction processors

Published by the IEEE Computer Society

each have a domain-specific instruction set architecture (ISA) for accelerating common request and memory transaction processing tasks, respectively. Pardis enforces the correctness of the derived schedule in hardware through dedicated command logic, which inspects—and if necessary, stalls—each DDRx command to DRAM to ensure that all DDRx timing constraints are met. This separation between performance optimization and correctness allows the firmware to dedicate request and transaction processor resources exclusively to optimizing performance and QoS, without expending limited compute cycles on verifying the derived schedule's correctness.

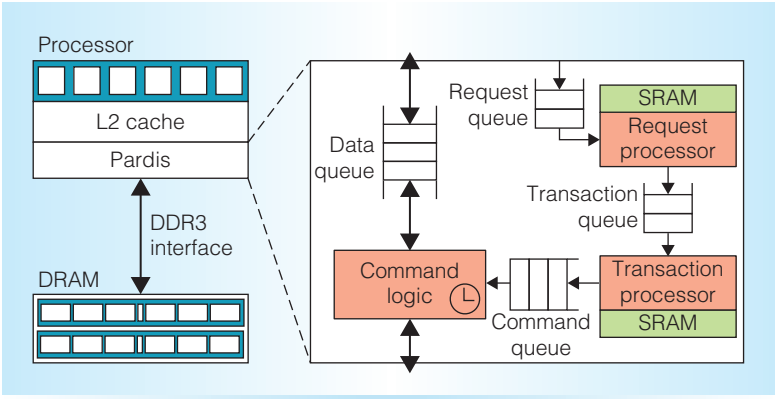


Figure 1. Example of Pardis in a computer system. Pardis receives read and write requests and generates DDRx commands to help move data between the processor and main memory. (DDR3: double data rate; SRAM: static RAM.)

Organization of DRAM systems

Modern DRAM systems are organized into a hierarchy of channels, ranks, banks, rows, and columns to exploit locality and request-level parallelism. Contemporary high-performance microprocessors commonly integrate two to four independent memory controllers, each with a dedicated DDRx channel. Each channel consists of multiple ranks that can be accessed in parallel, and each rank comprises multiple banks organized as rows by columns, sharing common data and address buses. A set of timing constraints dictate the minimum delay between each pair of commands issued to the memory system; maintaining high throughput and low latency necessitates a sophisticated memory controller that can correctly schedule requests around these timing constraints.

A typical DDRx memory controller receives a request stream consisting of reads and writes from the cache subsystem, and generates a corresponding DRAM command stream. Every read or write request requires accessing multiple columns of a row within the DRAM system. A row must be loaded into a row buffer by an *activate* command prior to a column access. Consecutive accesses to the same row, called *row hits*, enjoy the lowest access latency; however, a *row miss* necessitates issuing a *precharge* command to precharge the bitlines within the memory array, and then loading a new row to the row buffer using an *activate* command.

Pardis overview

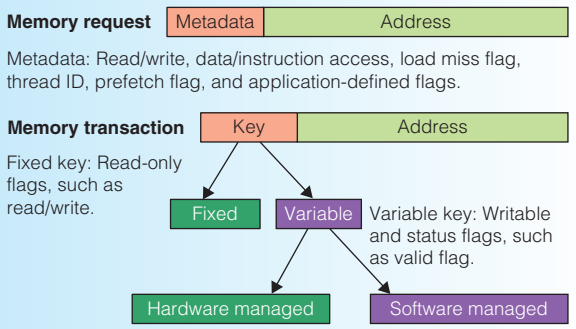
Figure 1 shows an example computer system comprising a multicore processor with Pardis, interfaced to off-chip DRAM over a third-generation double data rate (DDR3) memory channel. Pardis receives read and write requests from the last-level cache controller via a first-in, first-out (FIFO) queue, called the *request queue*, and generates DDR3 commands to orchestrate data movement between the processor and main memory using three tightly coupled processing elements.

Request processor

The request processor dequeues the next request from the head of the request queue, generates a set of DRAM coordinates—channel, rank, bank, row, and column IDs—for the requested address, and enqueues a new DDRx transaction with the generated coordinates in a transaction queue. Hence, the request processor represents the first level of translation—from requests to memory transactions—in Pardis, and is primarily responsible for DRAM address mapping.

Transaction processor

The transaction processor tracks each memory transaction's resource needs and timing constraints and uses this information to emit a sequence of DDRx commands that achieves performance, energy,



Memory command | Type | Address

Type: Valid, read, write, activate, precharge, power up, power down, refresh, sleep.

Request processor instructions

ALU	ADD, SUB, SLL, SRL, AND, OR, XOR, NOT
Control flow	JMP, BEQ, BNEQ, BTQE
Data memory	LOAD, STORE
Queue access	any instruction annotated with –R or –T

Transaction processor instructions

ALU	ADD, SUB, MIN, MAX, SLL, SRL, AND, OR, XOR, NOT
Control flow	JMP, JR, RETI, BLT, BLSG, BMSK, BEQ, BNEQ, BTQE, BCQE
Data memory	LOAD, STORE
Interrupt	MFSR, SIC
Queue access	LTQ, CTQ, UTQ, SRT, LCQ, ICQ, any instruction annotated with –C

Figure 2. Data types and instructions supported by Pardis. The request and transaction processors provide seven data types, 44 instructions, and three instruction flags. (ALU: arithmetic logic unit.)

and QoS goals. Therefore, the transaction processor is primarily in charge of DRAM command scheduling and tasks such as DRAM refresh and power management. The end result of transaction processing is a sequence of commands that are enqueued at a FIFO command queue.

Command logic

The command logic inspects the generated command stream, checks—and if necessary, stalls—the command at the head of the command queue to ensure all DDRx timing constraints are met, and synchronizes the issue of each command with the DDRx clock. The command logic is not programmable through an ISA; nevertheless, it provides configurable control registers specifying the value of each DDRx timing constraint, thereby making it possible to interface Pardis to different DDRx systems. The command logic enforces all timing constraints and guarantees the timing correctness of the scheduled command stream, making it possible to separate timing correctness from performance optimization.

Pardis architecture

Programming Pardis involves writing code for the request and transaction processors and configuring the control registers specifying DDRx timing constraints to the command logic. Together, the request and transaction processors provide the programmer with seven data types, 44 instructions, and three instruction flags (see Figure 2).

Request processor

The request processor is a 16-bit reduced-instruction-set computing (RISC) architecture with separate instruction and data memories; it provides specialized data types, storage structures, and instructions for address manipulation. The request processor's ISA supports two data types—an *unsigned integer* and a *request*. Programmer-visible storage structures within the request processor include the architectural registers, the data memory, and the request queue. The request processor supports fourteen 32-bit instructions of four different types: arithmetic logic unit (ALU), control flow, memory access, and queue access. Queue access instructions provide a mechanism for dequeuing requests from the request queue and enqueuing transactions at the transaction queue. After a request is dequeued from the request queue, its fields are available for processing in the register file.

Transaction processor

The transaction processor implements a 16-bit RISC ISA with split instruction and

data memories; due to the computational intensity of the tasks it supports (for example, command scheduling), the transaction processor's ISA is more powerful than that of the request processor. The transaction processor defines two new data types, called a *transaction* and a *command*. A transaction uses two key fields—fixed and variable keys—for performing associative lookups on the outstanding transactions in the transaction queue. For example, it is possible to search the fixed-key fields of all outstanding transactions to identify those transactions that occurred due to cache-missing loads. The fixed key is written by the request processor, and is read-only and searchable within the transaction processor. The variable key reflects the state of a transaction based on timing constraints, resource availability, and the state of the DRAM system. The variable key makes it possible, for example, to search for all transactions whose next command is a precharge to a specific bank.

The transaction processor provides 30 instructions comprising ALU, control flow, memory access, interrupt processing, and queue access operations. It provides 64 programmable counters for capturing processor and queue states (for example, the number of commands issued to the command queue). Each counter counts up and fires an interrupt when it reaches a preprogrammed threshold. The programmer can use the transaction processor to search for a given transaction by matching against fixed and variable keys among all valid transactions in the transaction queue; in the case of multiple matches, the transaction processor gives priority to the oldest matching transaction. A search operation requires two register operands specifying the fixed and variable keys. After a search, the transaction processor typically either

- loads a matching transaction into the architectural registers,
- updates a transaction in the queue with the contents of architectural registers, or
- counts the number of matches for a pair of fixed and variable keys.

Eventually, Pardis creates a DDRx command sequence for each transaction in the transaction processor and enqueues them in the command queue. The transaction processor allows the programmer to issue a legal command to the command queue using a dedicated instruction or an instruction flag. In addition to *precharge*, *activate*, *read*, and *write* commands, the firmware can also issue predefined control commands to control the command queue. For example, it can use a *sleep* command to throttle the DRAM system for active power management. Other DRAM maintenance commands allow changing DRAM power states and issuing a refresh to DRAM. By relying on dedicated command logic to stall each command until it is free of all timing constraints, Pardis lets the programmer write firmware code for the DDRx DRAM system without expending limited compute cycles on ensuring that all timing constraints are met.

Implementation

This article builds upon our ISCA 2012 paper and examines a scalar, pipelined implementation of Pardis as depicted in Figure 3.[10] The proposed implementation follows a six-step procedure for processing an incoming DRAM request, ultimately generating the corresponding DRAM command stream. First, Pardis assigns a unique request ID (URID) to a new DRAM request before it is enqueued at the FIFO request queue (1); the URID accompanies the request throughout the pipeline, and is used to associate the request with commands and DRAM data blocks. After a request is processed and its DRAM coordinates are assigned, a new transaction for the request is enqueued at the transaction queue (2). At the time the transaction is enqueued, the fixed key of the transaction is initialized to the request type, while the variable key is initialized based on the current state of the DRAM subsystem. A queued transaction is prioritized based on fixed and variable keys (3), after which the processor issues the next command of the transaction to the command queue (4). The command logic processes commands that are available in the command queue in FIFO order (5). A DRAM command is dequeued when it is ready to appear on the DDRx command bus (6),

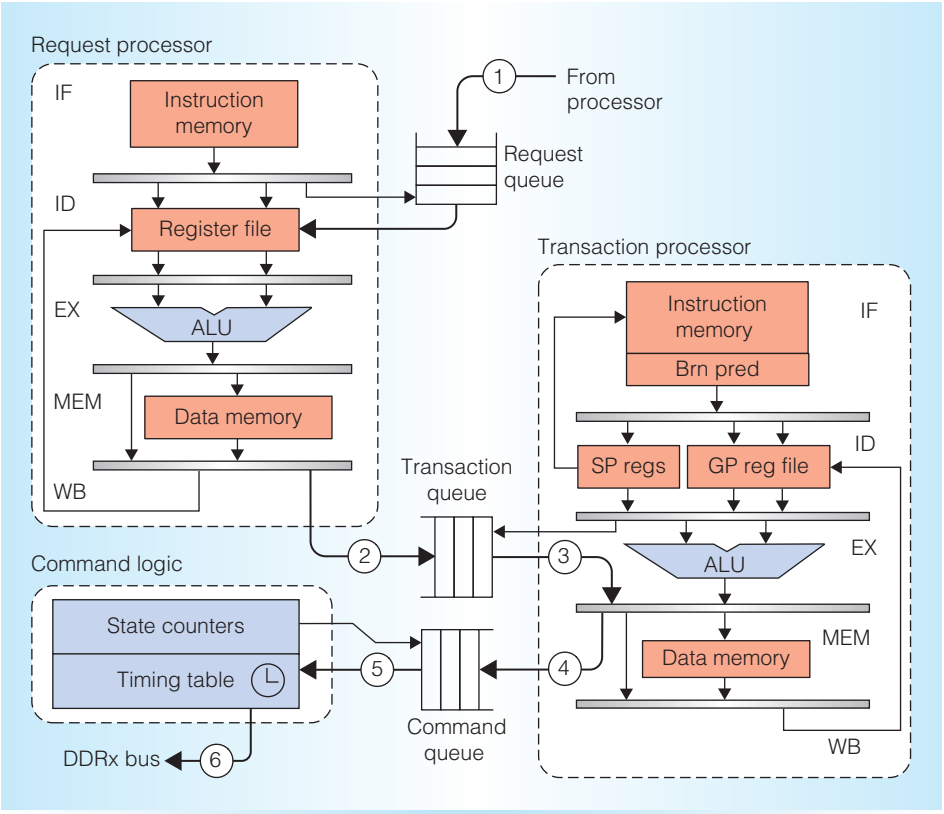


Figure 3. Example of the proposed Pardis implementation. This implementation follows a six-step procedure for processing an incoming DRAM request, ultimately generating the corresponding DRAM command stream. (IF: instruction fetch; ID: instruction decode; EX: instruction execute; MEM: memory access; WB: writeback; DDRx bus: double data rate bus; Brn pred: branch prediction; SP regs: special-purpose registers; GP reg file: general-purpose register file; ALU: arithmetic logic unit.)

and is issued to the DRAM subsystem at the next rising edge of the DRAM clock.

Request processor

The request processor implements a five-stage pipeline with a read interface to the request queue and a write interface to the transaction queue. In the first stage, the processor fetches an instruction from the instruction memory. The request processor predicts that all branches are taken, so when it mispredicts a branch, it nullifies the wrong-path instruction. In the second stage, the processor decodes the fetched instruction to extract control signals, reads operands from the register file, and dequeues the next request from the request queue if the instruction is annotated with a request flag (R-flag). If a request must be dequeued but the request queue is empty, the request processor stalls the decode and fetch stages until a new request arrives at the request queue. (Instructions in later pipeline stages continue uninterrupted.) Request registers (R1 through R4) can be written only from the request queue side (on a dequeue), and are read-only to the request processor. In the third stage, a simple 16-bit ALU executes the desired ALU operation or computes the effective address if the instruction is a load or a store. Loads and stores access the data memory in the fourth stage. In the final stage, the result of every instruction is written back to the register file, and if the transaction flag (T-flag) of the instruction is set, a new transaction is enqueued at the transaction queue.

Transaction processor

The transaction processor is a 16-bit, five-stage, pipelined processor. In the first stage, the processor fetches the next instruction from a 64-Kbyte instruction memory. The implementation divides branch and jump instructions into two categories: fast and slow. Fast branches include jump and branch on queue status instructions such as "branch if the transaction queue is empty" (BTQE) and "branch if the command queue is empty" (BCQE), for which the next instruction can be determined in the fetch stage; as such, these branches are not predicted and incur no performance losses due to branch mispredictions. Slow branches depend on register contents and are predicted by an 8-Kbyte-entry g-share branch predictor. Critical branches in the transaction processor are usually coded using the fast branch instructions (for example, infinite scheduling loops, or queue state checking).

In the second stage, the processor decodes the instruction, reads general- and special-purpose registers, and sets special-purpose interrupt registers. Special-purpose registers are implemented using a 64-entry array of programmable counters. The proposed implementation of Pardis uses 32 of these programmable counters (S0 through S31) for timer interrupts, and the remaining 32 programmable counters (S32 through S63) for collecting statistics to aid in decision making.

After decode, in the third stage, a 16-bit ALU performs arithmetic and logic operations; the transaction queue is accessed in parallel. Command queue and data memory accesses occur in the fourth stage, and the processor writes the result of the instruction back to the register file in the fifth stage.

Command logic

The command logic implementation uses masking and timing tables initialized at boot time based on DDRx parameters, plus a dedicated down counter for each DRAM timing constraint imposed by the DDRx standard. During each DRAM cycle, the command logic inspects the command at the head of the command queue, and retrieves a bit mask from the masking table to mask out timing constraints that are irrelevant to the command under consideration, such as column address latency (tCAS) in the case of a precharge. The remaining unmasked timers are used to generate a ready signal indicating whether the command is ready to be issued to the DRAM subsystem at the next rising edge of the DRAM clock.

Evaluation highlights

We evaluate the performance potential of Pardis by comparing fixed-function hardware and Pardis-based firmware implementations of the first-come, first served (FCFS),[11] first-ready, first-come, first-served (FR-FCFS),[11] parallelism-aware batch scheduler (Par-BS),[12] and thread cluster memory scheduling (TCMS)[13] algorithms. We also implement in firmware a recent DRAM power management algorithm proposed by Hur and Lin[14] and compare both the performance and the energy of this implementation to the fixed-function hardware implementation of the same algorithm. We evaluate DRAM refresh management on Pardis by comparing the fixed-function hardware implementation of the Elastic Refresh technique to its firmware implementation.[15] Finally, we evaluate the performance potential of application-specific optimizations enabled by Pardis by implementing custom address-mapping mechanisms. We evaluate DRAM energy and system performance by simulating 13 memory-intensive parallel applications, running on a heavily modified version of the SuperEscalar (SESC) simulator.[16] We measure the area, frequency, and power dissipation of Pardis by implementing the proposed system in Verilog HDL and synthesizing the proposed hardware.

Area, power, and delay: Where are the bottlenecks?

Figure 4 shows synthesis results on the area, power, and delay contributions of different hardware components. At 22 nm, a fully synthesizable implementation of Pardis operates at over 2 GHz, occupies 1.8 mm^2 of die area, and dissipates 152 mW of peak power; higher frequencies, lower power dissipation, or a smaller-area footprint can be attained through custom—rather than fully synthesized—circuit design. Most of the

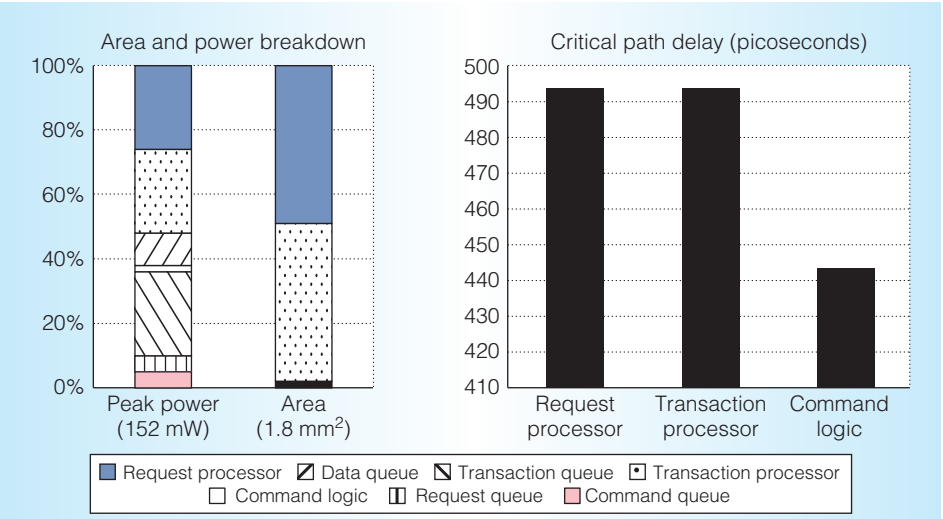


Figure 4. Delay, area, and peak-power characteristics of the synthesized Pardis implementation. Pardis operates at over 2 GHz, occupies 1.8 mm^2 of die area, and dissipates 152 mW of peak power. The black section at the bottom of the area column (about 2 percent) represents the data queue, transaction queue, command logic, request queue, and command queue combined.

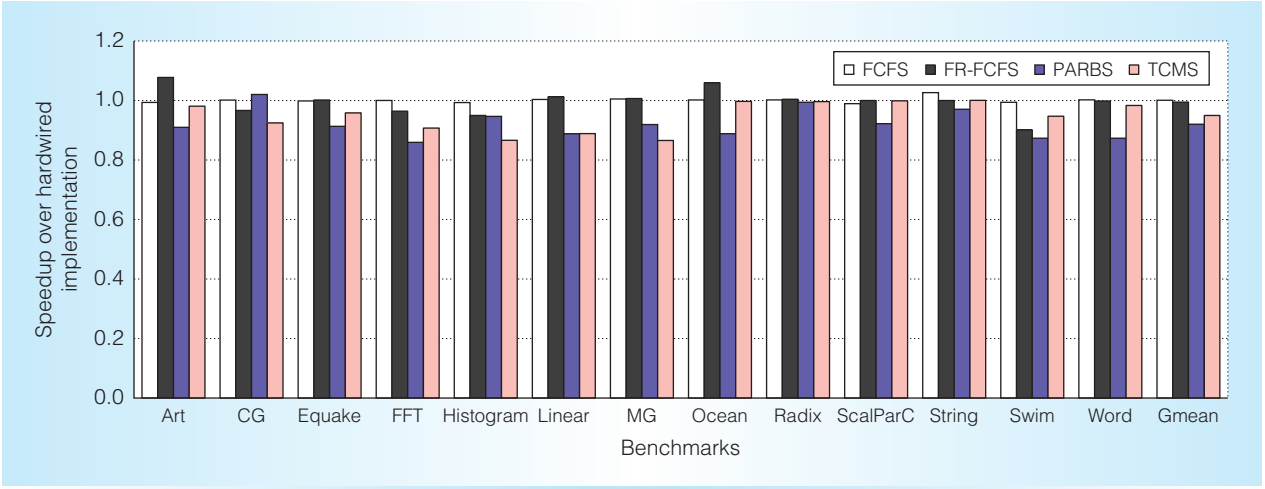


Figure 5. Performance of Pardis-based and hardwired implementations for the first-come, first-served (FCFS), first-ready, first-come, first-served (FR-FCFS), parallelism-aware batch scheduler (Par-BS), and thread cluster memory scheduling (TCMS) algorithms. Pardis-based implementations achieve performance within 8 percent of a hardwired memory controller.

area is occupied by the request and transaction processors because of four 64-Kbyte instruction and data static RAM (SRAM) arrays; however, the transaction queue—which implements associative lookups using content-addressable memory (CAM)—is a major power-hungry component (it uses 29 percent of the total power). Other major consumers of peak power are the transaction processor (29 percent) and the request processor (28 percent).

Scheduling policies

Figure 5 compares Pardis-based firmware implementations of FCFS,[11] FR-FCFS,[11] Par-BS,[12] and TCMS[13] scheduling algorithms

to their fixed-function hardware implementations. Pardis achieves virtually the same performance as fixed-function hardware on FCFS and FR-FCFS schedulers across all applications. For some benchmarks (for example, Art and Ocean with FR-FCFS), the Pardis version of a scheduling algorithm outperforms the fixed-function hardware implementation of the same algorithm by a small margin. This improvement is an artifact of the higher latency incurred in making decisions when using Pardis, which generally results in greater queue occupancies. As a result of having more requests to choose from, the scheduling algorithm can exploit bank parallelism and row buffer locality more effectively under the Pardis implementation. However, for Par-BS and TCMS—two compute-intensive scheduling algorithms—Pardis suffers from higher processing latency, thus hurting performance by eight percent and five percent, respectively.

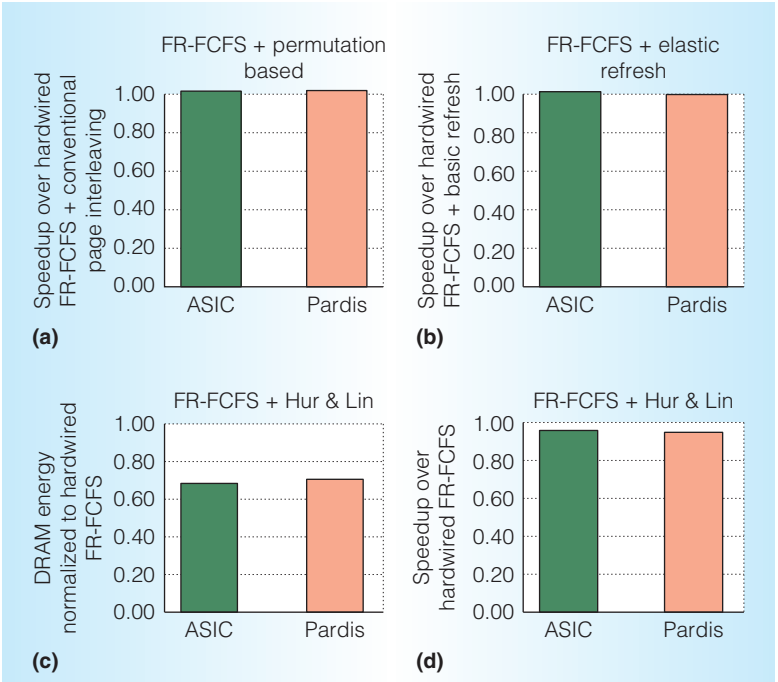


Figure 6. Comparison of Pardis-based and hardwired implementations: performance of address-mapping schemes (a), performance of refresh management (b), DRAM energy consumption under Hur and Lin's power management algorithm[14] (c), and performance under Hur and Lin's power management algorithm[14] (d).

Address mapping

To evaluate the performance of different DRAM address-mapping techniques on Pardis, we mapped the permutation-based interleaving technique[17] onto Pardis and compared it to its fixed-function hardware implementation (Figure 6a). The average performance of the two implementations differed by less than 1 percent.

Power management

DRAM power management with Pardis was evaluated by implementing Hur and Lin's queue-aware power management technique[14] in firmware and comparing the results to a fixed-function hardware implementation (see Figure 6c for energy and Figure 6d for performance); in both cases, the underlying command scheduling algorithm is FR-FCFS. The hardwired implementation reduces average DRAM energy by 32 percent over conventional FR-FCFS at a cost of four percent lower performance. The firmware implementation of queue-aware power management with Pardis shows similar results: 29 percent DRAM energy savings at a cost of five percent performance loss.

Refresh

To evaluate DRAM refresh management on Pardis, we considered a conventional on-demand DDR3 refresh method[18] as the baseline to which we compared fixed-function hardware and Pardis-based firmware implementations of the recently proposed Elastic Refresh algorithm[15] (Figure 6b). The Pardis-based refresh mechanism takes advantage of interrupt programming to manage the state of the ranks and to issue refresh commands at the right time. The results indicate that the average performance of firmware-based Elastic Refresh is within one percent of fixed-function hardware.

Memory system bandwidth and power are two extremely important problems that have a significant impact on overall system performance and energy efficiency. As a result, researchers have designed numerous memory controller optimizations to improve system performance, aiming at different performance objectives and system requirements.[12,14,15] These proposals mainly focus on optimizing existing control functions or adding new capabilities

to memory controllers—address mapping, command scheduling, QoS maintenance, DRAM refresh management, and memory power optimization are some examples. Such optimizations must satisfy different user objectives and system requirements; this complicates memory controller design. Not only is it challenging to satisfy multiple conflicting performance requirements in an existing hardwired memory controller, but it's impossible to incorporate application-specific optimizations into control policies implemented in fixed-function hardware. A programmable platform for memory controller design would therefore be a significant improvement over today's rigid and relatively inefficient systems.

In addition to its potential impact on existing memory systems, programmability also holds the promise of solving some of the key problems in next-generation memory interfaces. One effective solution to the off-chip memory bandwidth problem is to employ high-speed communication links between processor cores and a highly banked memory subsystem. This approach has recently been employed in Micron's hybrid memory cube (HMC)[19] to achieve significant improvements in memory bandwidth and power efficiency. The HMC, however, implements the memory controller on the DRAM package; as a result, the processor designer loses the ability to dictate exactly how the memory controller operates. A programmable memory controller would give that control back to the processor designers by letting them develop firmware for memory control functions.

Designing a programmable memory controller is a significant challenge. Compared to a hardwired memory controller, a programmable controller could result in slower request processing, thereby decreasing throughput and efficiency. In addition, as a result of instruction processing overheads, the control firmware may add extra latency to every memory access. Moreover, the controller's power dissipation and area may become serious limiting factors. Hence, it is critical to strike a careful balance between the controller's versatility and complexity.

Pardis is the first programmable DRAM controller to address these challenges. Unlike prior work on intelligent memory controllers (for instance, Impulse[20]) that allow configurable access to memory blocks via physical address remapping, Pardis provides programmability and configurability down to internal DRAM resources. To achieve a high degree of versatility with acceptable complexity, Pardis introduces a judicious division of labor between specialized hardware and firmware: request and transaction processing in firmware, and configurable timing validation in hardware. This task separation allows request and transaction processing resources to be dedicated exclusively to deriving the best schedule, without the burden of any extra cycles to verify the derived schedule's timing.

Pardis enables novel capabilities at the main memory controller. As opposed to a hardwired memory controller, a programmable controller allows application-specific control policies to manage the underlying main memory resources more efficiently; provides the required infrastructure for applying in-field updates, as well as patches to fix bugs and revise the control firmware; adds the ability to context-switch among different command schedulers in a multiprogrammed setting; and returns DRAM control to processor designers in next-generation HMC systems. MICRO

Acknowledgments

This work was supported in part by NSF grant CCF-1217418.

References

1. J. Martin et al., ''A Microprogrammable Memory Controller for High-Performance Dataflow Applications,'' *Proc. 35th European Solid-State Circuits Conf.* (ESSCIRC 09), IEEE, 2009, pp. 348-351.
2. G. Kornaros et al., ''A Fully Programmable Memory Management System Optimizing Queue Handling at Multi Gigabit Rates,'' *Proc. 40th Design Automation Conf.* (DAC 03), ACM, 2003, pp. 54-59.
3. Micron Technology, ''TN-29-01: Increasing NAND Flash Performance,'' 2006; www.micron.com/~/media/Documents/Products/Technical%20Note/NAND%20Flash/tn2901.pdf.

4. J. Kuskin et al., ''The Stanford FLASH Multiprocessor,'' *Proc. 21st Int'l Symp. Computer Architecture* (ISCA 94), IEEE CS, 1994, pp. 302-313.
5. S.K. Reinhardt, J.R. Larus, and D.A. Wood, ''Tempest and Typhoon: User-level Shared Memory,'' *Proc. 21st Int'l Symp. Computer Architecture* (ISCA 94), IEEE CS, 1994, pp. 325-336.
6. J. Carter et al., ''Impulse: Building a Smarter Memory Controller,'' *Proc. 15th Int'l Symp. High-Performance Computer Architecture* (HPCA 99), IEEE CS, 1999, pp. 70-79.
7. M. Browne et al., ''Design Verification of the S3.mp Cache Coherent Shared-Memory System,'' *IEEE Trans. Computers,* Jan. 1998, pp. 135-140.
8. A. Agarwal et al., ''The MIT Alewife Machine: Architecture and Performance,'' *Proc. 22nd Ann. Int'l Symp. Computer Architecture* (ISCA 95), ACM, 1995, pp. 2-13.
9. A. Firoozshahian et al., ''A Memory System Design Framework: Creating Smart Memories,'' Proc. *36th Int'l Symp. Computer Architecture* (ISCA 09), ACM, 2009, pp. 406-417.
10. M.N. Bojnordi and E. Ipek, ''PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards,'' *Proc. 39th Int'l Symp. Computer Architecture* (ISCA 12), IEEE, 2012, pp. 13-24.
11. S. Rixner et al., ''Memory Access Scheduling,'' *Proc. 27th Int'l Symp. Computer Architecture* (ISCA 00), IEEE, 2000, pp. 128-138.
12. O. Mutlu and T. Moscibroda, ''Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems,'' *Proc. 35th Int'l Symp. Computer Architecture* (ISCA 08), IEEE, 2008, pp. 32-41.
13. Y. Kim et al., ''Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior,'' *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE, 2010, pp. 65-76.
14. I. Hur and C. Lin, ''A Comprehensive Approach to DRAM Power Management,'' *Proc. Int'l Symp. High Performance Computer Architecture* (HPCA 08), IEEE CS, 2008, pp. 305-316.
15. J. Stuecheli et al., ''Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory,'' *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE, 2010, pp. 375-384.
16. J. Renau et al., ''SESC Simulator,'' Jan. 2005; http://sesc.sourceforge.net.
17. Z. Zhang, Z. Zhu, and X. Zhang, ''A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality,'' *Proc. 33rd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* 2000, ACM, pp. 32-41.
18. JEDEC, *DDR3 SDRAM Specification,* 2010.
19. J. Jeddeloh and B. Keeth, ''Hybrid Memory Cube New DRAM Architecture Increases Density and Performance,'' *Proc. IEEE Symp. VLSI Technology,* IEEE, 2012, pp. 87-88.
20. J. Carter et al., ''Impulse: Building a Smarter Memory Controller,'' *Proc. 5th Int'l Symp. High Performance Computer Architecture* (HPCA 99), IEEE CS, 1999, pp. 70-79.

Mahdi Nazm Bojnordi is a doctoral candidate in electrical and computer engineering at the University of Rochester. His research focuses on designing main memory controllers capable of performing application-specific performance and energy-efficiency optimizations. Nazm Bojnordi has an MS in electrical and computer engineering from the University of Tehran.

Engin Ipek is an assistant professor in the Departments of Computer Science and Electrical and Computer Engineering at the University of Rochester. His research focuses on computer architecture, with an emphasis on multicore architectures, hardware-software interaction, and high-performance memory systems. Ipek has a PhD in electrical and computer engineering from Cornell University.

Direct questions and comments about this article to Mahdi Nazm Bojnordi, CSB Room 401, University of Rochester, Rochester, NY 14627; bojnordi@ece.rochester.edu.