A Programmable Memory Controller for the DDRx Interfacing Standards

MAHDI NAZM BOJNORDI and ENGIN IPEK, University of Rochester

Modern memory controllers employ sophisticated address mapping, command scheduling, and power management optimizations to alleviate the adverse effects of DRAM timing and resource constraints on system performance. A promising way of improving the versatility and efficiency of these controllers is to make them programmable—a proven technique that has seen wide use in other control tasks, ranging from DMA scheduling to NAND Flash and directory control. Unfortunately, the stringent latency and throughput requirements of modern DDRx devices have rendered such programmability largely impractical, confining DDRx controllers to fixed-function hardware.

This article presents the instruction set architecture (ISA) and hardware implementation of PARDIS, a programmable memory controller that can meet the performance requirements of a high-speed DDRx interface. The proposed controller is evaluated by mapping previously proposed DRAM scheduling, address mapping, refresh scheduling, and power management algorithms onto PARDIS. Simulation results show that the average performance of PARDIS comes within 8% of fixed-function hardware for each of these techniques; moreover, by enabling application-specific optimizations, PARDIS improves system performance by 6 to 17% and reduces DRAM energy by 9 to 22% over four existing memory controllers.

Categories and Subject Descriptors: C.1.0 [**Processor Architectures**]: General; C.5.3 [**Computer System Implementation**]: Microprocessors; B.1.5 [**Control Structures and Microprogramming**]: Microcode Applications

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Programmable, memory controller

ACM Reference Format:
Bojnordi, M. N. and Ipek, E. 2013. A programmable memory controller for the DDRx interfacing standards. ACM Trans. Comput. Syst. 31, 4, Article 11 (December 2013), 31 pages.
DOI: http://dx.doi.org/10.1145/2534845

1. INTRODUCTION

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 timing and resource constraints by orchestrating data movement between the processor and main memory. Contemporary DDRx 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

This work is supported in part by the National Science Foundation under grant CCF-1217418.
Authors' addresses: M. N. Bojnordi, Electrical and Computer Engineering Department, University of Rochester; email: bojnordi@ece.rochester.edu; E. Ipek, Electrical and Computer Engineering Department, University of Rochester.

DOI: http://dx.doi.org/10.1145/2534845

optimization, compounded by the 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 the controller programmable; indeed, programmability has proven useful in the context of other complex control tasks from DMA scheduling [Martin et al. 2009; Kornaros et al. 2003] to NAND Flash [Micron Technology, Inc. 2009b] and directory control [Kuskin et al. 1994; Reinhardt et al. 1994; Carter et al. 1999; Browne et al. 1998; Agarwal et al. 1995; Firoozshahian et al. 2009]. In these and other architectural control problems, programmability allows the controller to be customized based on system requirements and performance objectives, makes it possible to perform in-field firmware updates to the controller, and enables application-specific control policies. Unfortunately, extending such programmability to a DRAM controller is complicated by the stringent latency and throughput constraints of DDRx protocols, which currently operate at data rates in excess of 10GB/s per channel. As a result, contemporary memory controllers are invariably confined to implementing DRAM control policies in ASIC-like, fixed-function hardware blocks.

This article presents PARDIS, a programmable memory controller that provides sufficiently high performance to make the firmware implementation of DDRx control policies practical. 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 each have a domain-specific ISA for accelerating common request and memory transaction processing tasks, respectively. The timing correctness of the derived schedule is enforced 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 timing 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 correctness of the derived schedule.

Synthesis results on a complete RTL implementation of the PARDIS system indicate that the proposed controller occupies less than 1.8mm^2 of area and consumes less than 152mW of peak power at 22nm. Four command scheduling policies, an address mapping technique, a refresh scheduling mechanism, and a recently proposed power management algorithm are implemented in firmware and mapped onto PARDIS for evaluation; when averaged over a set of 13 scalable parallel applications, PARDIS achieves performance and DRAM energy within 8% of fixed-function hardware for each of these techniques. Furthermore, by enabling application-specific address-mapping optimizations, PARDIS improves performance by 6 to 17% and DRAM energy by 9 to 22% over four existing memory controllers.

2. BACKGROUND AND MOTIVATION

Modern DRAM systems are organized into a hierarchy of channels, ranks, banks, rows, and columns to exploit locality and 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 $\times$ columns, sharing data and address buses. A set of timing constraints dictate the minimum delay between each pair of commands issued to memory; maintaining high throughput and low latency necessitates a sophisticated memory controller that can correctly schedule requests around these timing constraints.

A DDRx memory controller receives a request stream consisting of reads and writes from the cache subsystem, and generates a corresponding DRAM command stream.

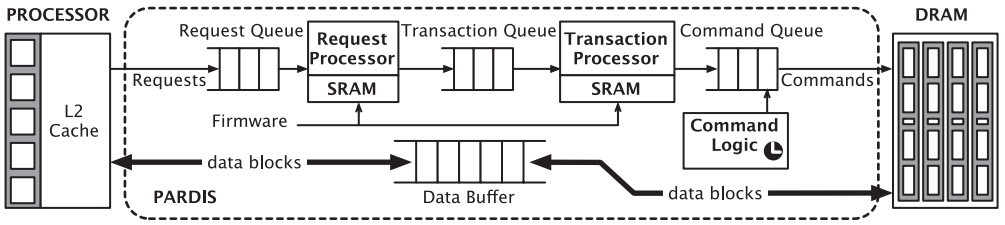


Fig. 1. Illustrative example of PARDIS in a computer system.

Every request requires accessing multiple columns of a row within DRAM. A row needs to 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, whereas 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.

3. OVERVIEW

Figure 1 shows an example computer system consisting of a multicore processor with PARDIS, interfaced to off-chip DRAM over a DDR3 channel. PARDIS receives read and write requests from the last-level cache controller, and generates DDR3 commands to orchestrate data movement between the processor and main memory. Internally, PARDIS is comprised of a *request processor*, a *transaction processor*, and *command logic*. These three tightly-coupled processing elements work in tandem to translate each memory request to a valid sequence of DDR3 commands.

3.1. Request Processor

Upon arrival at the memory controller, each request is enqueued at a FIFO *request queue* that interfaces to the request processor. The job of the request processor is to dequeue the next request at the head of the request queue, to generate a set of DRAM coordinates—channel, rank, bank, row, and column IDs—for the requested address, and to enqueue 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.

3.2. Transaction Processor

The transaction processor operates on the DDRx transactions that the request processor enqueues in the transaction queue. The primary job of the transaction processor is to track the resource needs and timing constraints for each memory transaction, and to use this information to emit a sequence of DDRx commands that achieves performance, energy, and QoS goals. The transaction processor's ISA is different from the request processor's, and offers several important capabilities. A subset of the instructions, called *transaction management instructions*, allows the firmware to categorize memory requests based on the state of the memory subsystem (e.g., requests that need a precharge), the request type (e.g., a write request), and application-specific criteria (e.g., thread IDs) to derive a high-performance, efficient command schedule. A second subset of the instructions, called *command management instructions*, allows the firmware to emit either the next required command for a given transaction (e.g., an activate command to a particular row), or a new command for various DRAM management purposes (e.g., power-management or refresh scheduling).

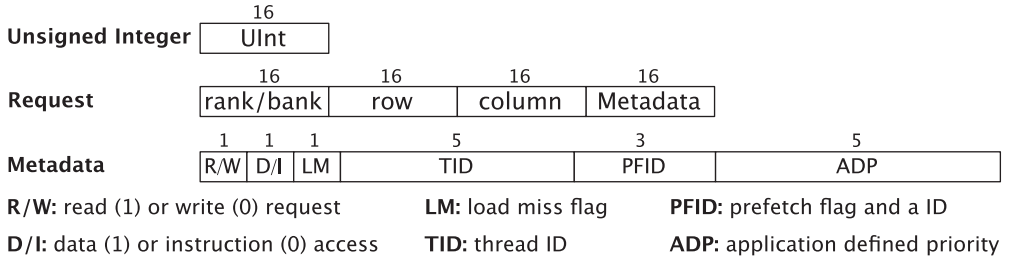


Fig. 2. Data types supported by the request processor.

3.3. Command Logic

The purpose of the command logic is to inspect the generated command stream, to check, and if necessary, to stall, the command at the head of the command queue to ensure all DDRx timing constraints are met, and to synchronize 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. Since the command logic enforces all timing constraints and guarantees the timing correctness of the scheduled command stream, it becomes possible to separate timing correctness from performance optimization.

4. INSTRUCTION SET 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.

4.1. Request Processing

The request processor is a 16-bit RISC architecture with separate instruction and data memories (i.e., a Harvard architecture). The primary goals of the request processor are address mapping and translating each request to a DDRx transaction; to achieve these goals, the request processor provides specialized data types, storage structures, and instructions for address manipulation.

4.1.1. Data Types. Request processing algorithms are dominated by arithmetic and logical operations on memory addresses. Two data types, an *unsigned integer* and a *request*, suffice to represent the information used in these algorithms (Figure 2). An unsigned integer is 16 bits wide, and can be used by every instruction except jumps. A request is 64 bits wide, comprising a 48-bit address and a 16-bit metadata field recording information about the DRAM request: the type of memory operation (read or write), the destination cache type (data or instruction), whether the access is initiated by a load miss, the owner thread's ID, whether the request is a prefetch, and other application-specific priority flags.

4.1.2. Storage Model. Programmer-visible storage structures within the request processor include the architectural registers, the data memory, and the request queue. The request processor provides 32 architectural registers (R0-R31); of these, one (R0) is hardwired to zero; four (R1-R4) are dedicated to reading a 64-bit request from the request queue; and four (R5-R8) are used for temporarily storing a transaction until it is enqueued at the transaction queue. The data memory has a linear address space with 16-bit data words, indexed by a 16-bit address.

Instruction Format I: Arithmetic, Shift, and Logic

1	1	4	5	5	5	
R	T	opcode	destination	source 1	source 2	

Instruction Format II: Control Flow and Data Memory Access

1	1	4	5	5	16
R	T	opcode	operand 1	operand 2	immediate value

Fig. 3. Instruction formats supported by the request processor.

Table I. Four Different Types of Instructions Supported by the Request Processor

	Instruction/Flag		Operation	Description
Arithmetic and Logical	ADD	Rd, Rs1, Rs2	$R_d \leftarrow R_{s1} + R_{s2}$	Addition
	SUB	Rd, Rs1, Rs2	$R_d \leftarrow R_{s1} - R_{s2}$	Subtraction
	SLL	Rd, Rs1, Rs2	$R_d \leftarrow R_{s1} \ll R_{s2}$	Shift Left Logically
	SRL	Rd, Rs1, Rs2	$R_d \leftarrow R_{s1} \gg R_{s2}$	Shift Right Logically
	AND	Rd, Rs1, Rs2	$R_d \leftarrow R_{s1} \wedge R_{s2}$	bitwise AND operation
	OR	Rd, Rs1, Rs2	$R_d \leftarrow R_{s1} \vee R_{s2}$	bitwise OR operation
	XOR	Rd, Rs1, Rs2	$R_d \leftarrow R_{s1} \oplus R_{s2}$	bitwise XOR operation
	NOT	Rd, Rs1	$R_d \leftarrow \neg R_{s1}$	bitwise NOT operation
Memory Access	LD	Rd, Rs1, Vim	$R_d \leftarrow MEM[R_{s1} + V_{im}]$	Load Data
	SD	Rs1, Rs2, Vim	$MEM[R_{s2} + V_{im}] \leftarrow R_{s1}$	Store Data
Control Flow	BEQ	Rs1, Rs2, Vim	$if(R_{s1} = R_{s2})$ $PC \leftarrow V_{im}$	Branch if Equal
	BNEQ	Rs1, Rs2, Vim	$if(R_{s1} \neq R_{s2})$ $PC \leftarrow V_{im}$	Branch if Not Equal
	JMP	Vim	$PC \leftarrow V_{im}$	Jump
	BTQE	Vim	$if(TQ.empty = true)$ $PC \leftarrow V_{im}$	Branch if the Transaction Queue is empty
Queue Access	-R		$R_{1...4} \leftarrow RQ[head]$	Request read
	-T		$TQ[tail] \leftarrow R_{5...8}$	Transaction write

R_d: destination register, R_{s1}: source register 1, R_{s2}: source register 2, V_{im}: immediate value
MEM: data memory, PC: program counter, RQ: request queue, TQ: transaction queue

4.1.3. Instructions. As depicted in Figure 3, the request processor supports 32-bit instructions with one, two, or three operands represented by two instruction formats. Table I shows all of the supported request processing instructions.

Arithmetic and logical instructions. Supported ALU operations include addition, subtraction, logical shifts, and bitwise logical operations. All ALU instructions can use any of the 32 architectural registers as an input operand; however, registers R0 to R4 are not writable by the ALU instructions. (R0 is hardwired to zero, whereas R1-R4 are dedicated to reading read memory requests from the request queue.)

Memory access instructions. Only loads and stores access the data memory, and only the displacement addressing mode (16-bit immediate + register) is supported for simplicity.

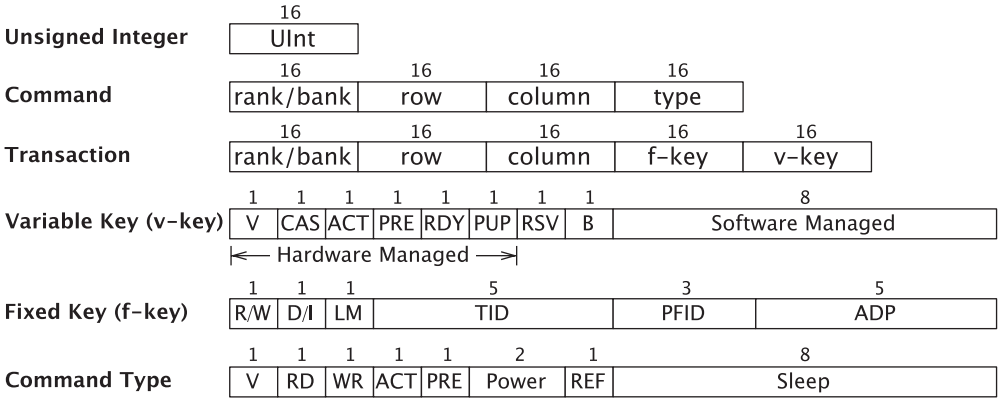
Control flow instructions. The request processor supports both jumps and branches. Possible branch conditions that can be tested are equality and inequality between two registers, and whether the transaction queue is empty. State of the transaction queue is useful for context switching between control policies at the request processor. The target address of a branch is a 16-bit immediate value, which is an absolute address to the instruction memory.

Queue access flags. All queue access operations in the request processor are performed using instruction flags. The firmware needs a mechanism for dequeuing requests from the request queue and enqueuing transactions at the transaction queue. To fulfill this need, request processing instructions are equipped with two flags called "R" and "T". An instruction annotated with the R flag dequeues the request at the head of the request queue, and loads the request fields into registers R1-R4 prior to execution; likewise, after an instruction annotated with the T flag executes, it enqueues a new transaction based on the contents of registers R5-R8 at the transaction queue. Hence, a typical sequence of instructions for processing a request involves (1) copying different fields of the 64-bit request into general-purpose registers with the R flag; (2) operating on these fields to compute channel, rank, bank, row, and column IDs; and (3) copying the resulting transaction fields from the register file to the transaction queue with the T flag. A single instruction is allowed to be annotated with both R and T flags, in which case it dequeues a request, operates on it, and enqueues a transaction based on the contents of R5-R8. After a request is dequeued from the request queue, its fields are available for processing in the register file; therefore, all request processor instructions can operate on each of the four fields of a request.

4.2. Transaction Processing

The transaction processor implements a 16-bit RISC ISA with split instruction and data memories, and is in charge of command scheduling and DRAM management. These tasks require sophisticated instructions and necessitate a more powerful ISA than that of the request processor.

4.2.1. Data Types. In addition to a basic 16-bit unsigned integer, the transaction processor defines two new data types called a *transaction* and a *command*. A transaction consists of three fields: an *address*, a *fixed key* (f-key in Figure 4), and a *variable key* (v-key in Figure 4). The address field is 48 bits wide and is in DRAM-coordinate format, where the least significant bits represent the byte offset, the next few bits represent the page ID, and so on (Figure 8). The fixed and variable key fields are used 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 are due to cache-missing loads. A 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 variable key consists of two disjoint parts called the *hardware managed* and *software managed* regions. The hardware managed region comprises a valid bit (V); three flags indicating the next valid DRAM command for the transaction (i.e., a read/write, precharge, or activate); and a programmed ready bit (RDY). The hardware managed region is automatically updated by hardware each cycle, whereas



V: transaction valid flag
CAS: column access (read or write)
ACT: DRAM activate command
PRE: DRAM precharge command
RDY: ready flag indicating that the next command is ready
PUP: DRAM power up command
B: busy flag indicating that the transaction is under process
R/W: read (1) or write (0) request
D/I: data (1) or instruction (0) access
LM: load miss flag
RSV: reserved flag
TID: thread ID
PFID: prefetch flags
ADP: application defined priority
RD: DRAM read command
WR: DRAM write command
Power: power up, power down
REF: DRAM refresh command
Sleep: sleep command and duration

Fig. 4. Data types supported by the transaction processor.

the software managed region can only be modified by a dedicated instruction that overwrites its fields.

The request processor may enqueue new transactions while the transaction processor is working on one iteration of a scheduling loop. To prevent these new transactions from interfering with the ongoing policy computation, the transaction processor uses theâ busy flag (B) that marks the transactions that are currently being worked on. Associative search instructions include this flag in their search key to avoid interference from the request processor.

A command consists of two fields called *address* and *type*. The command can be a DRAM data transfer command such as a read, write, precharge, or activate, a power management command such as power up or power down, a refresh command, or a special "sleep" command that is interpreted by the command logic as a multicycle throttling request for active power management.

4.2.2. Storage Model. The transaction processor provides the programmer with *register*, *data memory*, *transaction queue*, and *command queue* storage abstractions. The processor has 64 general-purpose registers (R0-R63), with R0 hardwired to zero. In addition, the processor provides 64 special-purpose registers (S0-S63) bundled as an array of counters for implementing timer-based interrupts and statistics counters for decision making. Both the instruction and data memories are accessed by 16-bit addresses, which results in address space sizes of 64KB each. The transaction processor accesses the outstanding transactions in the transaction queue via associative search instructions, and generates a command sequence to be enqueued at the command queue.

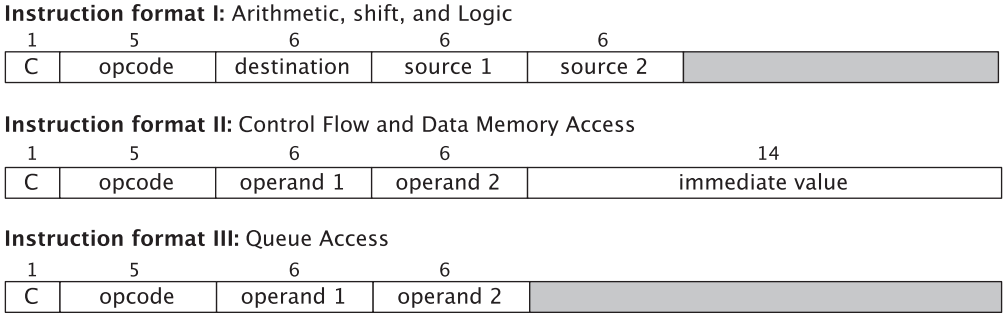


Fig. 5. Instruction formats supported by the transaction processor.

4.2.3. Instructions. The transaction processor provides 32 instructions, comprising ALU, control flow, memory access, interrupt processing, and queue access operations. The transaction processor provides three instruction formats (Figure 5), which allow an instruction to use up to three operands (register specifiers and an immediate value). Table II shows all instructions supported by the transaction processor.

Arithmetic and logical instructions. The ISA supports 12 ALU instructions, including ADD, SUB, MIN, MAX, shifts, and bitwise logical operations.

Memory access instructions. Only loads and stores are permitted to access the data memory, and the only supported addressing mode is displacement (16-bit immediate + register).

Control flow instructions. Nine control flow instructions are supported to detect various memory system states and events. In addition to conventional jumps and branches, the ISA provides "branch if the transaction queue is empty" (BTQE); "branch if the command queue is empty" (BCQE); and "branch if less-than or skip if greater-than" (BLSG) instructions. BLSG allows the firmware to implement nested if-then-else constructs with high performance by reducing the number of instruction fetches and overlapping branch delay slots. (An example usage of this instruction is presented in Section 5.4.)

Interrupt programming instructions. The transaction processor provides 64 programmable counters which are used for capturing processor and queue states (e.g., the number of commands issued to the command queue). Every counter counts up and fires an interrupt when a preprogrammed threshold is reached. A programmable interrupt counter is written by a "set interrupt counter" (SIC) instruction, and is read by a "move from special register" (MFSR) instruction. SIC accepts two register specifiers, and an immediate value specifying the counter ID. One of the two register operands is the address of the interrupt service routine for handling the interrupt, and the other register is used for specifying the top counter value, after which the counter interrupt must fire. By default, the execution of an interrupt service routine cannot be interrupted by any counter. From the time an interrupt fires until its corresponding service routine executes a "return from an interrupt service routine" (RETI) instruction, any new interrupts are logged for future processing. The transaction processor, however, allows the programmer to remove this protection by an "unmask interrupt counters" (UIC) instruction, and to enable masking interrupts by a "mask interrupt counters" (MIC) instruction. A counter is read by the MFSR instruction, which moves the value of the specified counter to a general-purpose register.

Queue access. The transaction processor allows the programmer to search for a given transaction by matching against fixed and variable keys among all valid transactions

Table II. Instruction Set Architecture of the Transaction Processor

	Instruction/Flag		Operation	Description
	`ADD`	`Rd, Rs1, Rs2`	$R_d \leftarrow R_{s1} + R_{s2}$	Addition
	`SUB`	`Rd, Rs1, Rs2`	$R_d \leftarrow R_{s1} - R_{s2}$	Subtraction
	`MIN`	`Rd, Rs1, Rs2`	$if(R_{s1} < R_{s2})$ $R_d \leftarrow R_{s1}$ *else* $R_d \leftarrow R_{s2}$	Minimum of the two source operands
Arithmetic and Logical	`MAX`	`Rd, Rs1, Rs2`	$if(R_{s1} > R_{s2})$ $R_d \leftarrow R_{s1}$ *else* $R_d \leftarrow R_{s2}$	Maximum of the two source operands
	`SLL`	`Rd, Rs1, Rs2`	$R_d \leftarrow R_{s1} \ll R_{s2}$	Shift Left Logically
	`SRL`	`Rd, Rs1, Rs2`	$R_d \leftarrow R_{s1} \gg R_{s2}$	Shift Right Logically
	`AND`	`Rd, Rs1, Rs2`	$R_d \leftarrow R_{s1} \wedge R_{s2}$	bitwise AND operation
	`OR`	`Rd, Rs1, Rs2`	$R_d \leftarrow R_{s1} \vee R_{s2}$	bitwise OR operation
	`XOR`	`Rd, Rs1, Rs2`	$R_d \leftarrow R_{s1} \oplus R_{s2}$	bitwise XOR operation
	`NOT`	`Rd, Rs1`	$R_d \leftarrow \neg R_{s1}$	bitwise NOT operation
Memory Access	`LD`	`Rd, Rs1, Vim`	$R_d \leftarrow MEM[R_{s1} + V_{im}]$	Load Data
	`SD`	`Rs1, Rs2, Vim`	$MEM[R_{s2} + V_{im}] \leftarrow R_{s1}$	Store Data
Queue Access	`LTQ`	`Rd, Rs1, Rs2`	$R_d \leftarrow TQ.search(R_{s1}, R_{s2})$	Load from the Transaction Queue
	`CTQ`	`Rd, Rs1, Rs2`	$R_d \leftarrow TQ.count(R_{s1}, R_{s2})$	Count matches in the Transaction Queue
	`UTQ`	`Rs1, Rs2, Vim`	$TQ.update(R_{s1}, R_{s2}, V_{im})$	Update the Transaction Queue
	`SRT`	`Rs1`	$TQ.ready \leftarrow Rs1$	Set Ready Threshold
	`LCQ`	`Rd`	$R_d \leftarrow CQ.state$	Load state of the Command Queue
	`ICQ`	`Rs1`	$CQ.issue(R_{s1})$	Issue command to the Command Queue
Control Flow	`BLT`	`Rs1, Rs2, Vim`	$if(R_{s1} < R_{s2})$ $PC \leftarrow V_{im}$	Branch if Less-Than
	`BLSG`	`Rs1, Rs2, Vim`	$if(R_{s1} < R_{s2})$ $PC \leftarrow V_{im}$ $elseif(R_{s1} > R_{s2})$ $PC \leftarrow PC + 1$	Branch if Less-than; Skip if Greater
	`BMSK`	`Rs1, Rs2, Vim`	$if(R_{s1} \wedge R_{s2})$ $PC \leftarrow V_{im}$	Branch if Masking results is non-zero
	`BEQ`	`Rs1, Rs2, Vim`	$if(R_{s1} = R_{s2})$ $PC \leftarrow V_{im}$	Branch if Equal
	`BNEQ`	`Rs1, Rs2, Vim`	$if(R_{s1} \neq R_{s2})$ $PC \leftarrow V_{im}$	Branch if Not Equal
	`BTQE`	`Vim`	$if(TQ.empty = true)$ $PC \leftarrow V_{im}$	Branch if the Transaction Queue is empty
	`BCQE`	`Vim`	$if(CQ.empty = true)$ $PC \leftarrow V_{im}$	Branch if the Command Queue is empty
	`JR`	`Rs1`	$PC \leftarrow R_{s1}$	Jump to Register
	`JMP`	`Vim`	$PC \leftarrow V_{im}$	Jump
Interrupt Access	`MFSR`	`Rd, Ss1`	$R_d \leftarrow S_{s1}$ $S_{s1} \leftarrow 0$	Move From Special Register
	`SIC`	`Sd, Rs1, Vim`	$S_d.setIntCnt(R_{s1}, V_{im})$	Set Interrupt Counter
	`RETI`		$PC \leftarrow IC.return()$	Return from Interrupt service routine
	`MIC`		$IC.mask \leftarrow true$	Mask Interrupt Counter
	`UIC`		$IC.mask \leftarrow false$	Unmask Interrupt Counter

R_d: destination register, R_{s1}: source register 1, R_{s2}: source register 2, V_{im}: immediate value
MEM: data memory, PC: program counter, RQ: request queue, TQ: transaction queue
S_d: destination special register, S_{s1}: source special register 1, IC: interrupt controller

in the transaction queue; in the case of multiple matches, priority is given to the oldest matching transaction. Prior to a search, the search key is stored in an architectural register. If the search key is stored in an even-numbered register, the following odd-numbered register is used to store a bit mask that determines which bits from the key should contribute to the search. The most common type of search operation at the transaction processor involves finding all transaction queue entries with a particular flag on (e.g., all transactions that need a precharge command). This type of search requires the same bit pattern for both the key and the mask; as such, a single register is sufficient to store both the key and the mask. This is made possible by storing the relevant key in an odd-numbered register. A search operation requires two register operands specifying the fixed and variable keys, and is typically followed by one of three actions:

(1) *Load transaction.* Loading a transaction involves executing a "load transaction queue" (LTQ) instruction, which writes the next command of the selected transaction (Figure 4) to a specified destination register, and the address field to a set of dedicated address registers. If the search operation preceding LTQ results in a mismatch, LTQ sets the valid bit (Figure 4) of the command field to zero; future instructions check this bit to determine if the search has succeeded.
(2) *Update transaction.* The transaction processor allows the programmer to update a transaction using the "update transaction queue" (UTQ) instruction. The lower eight bits of the immediate field of UTQ are written into the software managed region of the variable key. This allows firmware to classify matches based on decision-making requirements; for example, the batch-scheduler algorithm in Par-BS [Mutlu and Moscibroda 2008] can mark a new batch of transactions using UTQ.
(3) *Count the number of matches.* Using a "count transaction queue" (CTQ) instruction, the programmer can count the number of transactions that match the preceding search, and can store the result in a specified destination register. This capability allows the firmware to make decisions according to the demand for different DRAM resources; for example, a rank with no pending requests can switch to a low power state, or a heavily contended bank can be prioritized.

Eventually, a DDRx command sequence is created for each transaction in the transaction processor and enqueued in the command queue. The transaction processor allows the programmer to issue a legal command to the command queue by placing the command type and the address in a set of command registers, and then executing an "issue command queue" (ICQ) instruction. An alternative to using ICQ is to use a command flag that can be added to any instruction (-C). In addition to precharge, activate, read and write commands, the firmware can also issue a "sleep" command to throttle the DRAM system for active power management. The sleep command specifies the number of cycles for which the command logic should stall once the sleep command reaches the head of the command queue. 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 allows the programmer to write firmware code for the DDRx DRAM system without worrying about timing constraints or synchronization with the DRAM clock; however, knowing the time at which different commands will become ready to issue is still critical to deriving a high-performance, efficient command schedule. To allow the firmware to deliver better performance by inspecting when a command will become ready, a ready bit is added to each transaction; by default, the ready bit indicates that the command will be ready in the next clock cycle; however, the programmer can change this to a larger number of cycles using a "set ready threshold" (SRT) instruction.

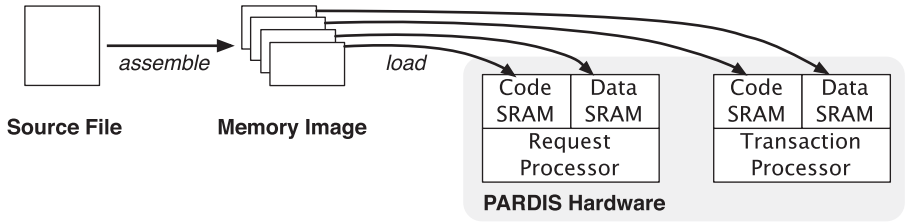


Fig. 6. Illustrative example of PARDIS firmware implementation.

```
# Page Interleaving Address Mapping
start: ADD-R R5,R1,R0
       ADD   R6,R2,R0
       ADD   R7,R3,R0
       ADD-T R8,R4,R0
       JMP   start
```

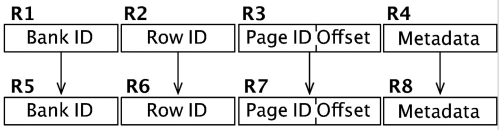


Fig. 7. Illustrative example of page interleaving at the request processor. The destination register in each line of code is the leftmost register.

5. FIRMWARE IMPLEMENTATION

Writing firmware for PARDIS includes programming the request and transaction processors. To achieve this, the programmer initializes the SRAM arrays holding code and data on the request and transaction processors (Figure 6). The two code SRAMs hold address mapping and command scheduling algorithms, while all the required constant values for initialization are stored in the data SRAMs. The firmware model enables implementing address mapping, command scheduling, refresh management, and DRAM power management efficiently.

5.1. Example Firmware Code for Page Interleaving and Permutation-Based Address Mapping

As explained in Section 4.1.2, registers R1-R4 are used for holding the address and metadata fields of the request once the request is dequeued from the request queue, and registers R5-R8 are used for enqueuing the next transaction at the transaction queue. The firmware can either directly copy R1-R4 to R5-R8 to implement page interleaving [Jacob et al. 2008], or can operate on R1-R4 to implement more sophisticated address mapping heuristics. Figure 7 shows an example code snippet that implements page interleaving. In the figure, an infinite loop iteratively dequeues the next request, copies the contents of the request registers to transaction registers, and enqueues a new transaction at the transaction queue. The first instruction of the loop is annotated with the R flag, which forces it to block until the next request arrives. Since one source operand of each ADD instruction in the example is the hardwired zero register (R0), each ADD instruction effectively copies one source request register to a destination transaction register. The last ADD instruction is annotated with the T flag to check for available space in the transaction queue, and to enqueue a new transaction.

As a second example of address mapping at the request processor, an implementation of permutation-based page interleaving [Zhang et al. 2000] is shown in Figure 8. In every iteration of the address mapping loop, an AND instruction first filters out unwanted bits of the row ID field using a bit mask. (The mask is defined based on DRAM parameters, such as the number of banks.) Then, a shift-right logical (SRL) instruction aligns the selected row ID bits with the least significant bits of the bank ID. An XOR instruction generates the new bank ID for the request, and stores the results

```
# Permutation Based Address Mapping
# Initialization
       LD    R10,R0(0)
       LD    R11,R0(1)
# Main Loop
start: AND-R R9,R2,R10
       SRL   R9,R9,R11
       XOR   R5,R1,R9
       ADD   R6,R2,R0
       ADD   R7,R3,R0
       ADD-T R8,R4,R0
       JMP   start
```

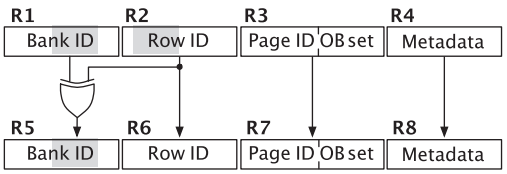


Fig. 8. Illustrative example of permutation-based address mapping on the request processor. The destination register in each line of code is the leftmost register.

```
### FCFS Scheduling Algorithm
# Initialization
       XOR    R1,R1,R1 # Reset the key mask
# Main loop
start: BTQE   start    # A: wait for a transaction
       LTQ-C  R9,R0,R0 # B: issue oldest command
       JMP    start    # Go to next transaction
```

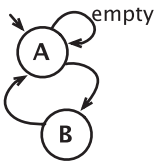


Fig. 9. Example transaction processing code for the FCFS scheduling algorithm. The leftmost register in each line of code is the destination register.

in a transaction register. The remaining instructions copy source request registers to destination transaction registers, and enqueue a transaction at the transaction queue.

5.2. Example Firmware Code for FCFS Command Scheduling Algorithm

As a simple example of transaction scheduling, the firmware can emit the next valid DRAM command of the oldest transaction, and can process all requests in the same order that they arrive at the request processor. The transaction processing code of this first-come first-serve (FCFS) algorithm is shown in Figure 9. The code snippet shows an infinite loop with three instructions. A BTQE instruction keeps checking the empty flag of the transaction queue until it reads a zero. The second instruction is a load from transaction queue (LTQ), which is annotated with the C flag. Since the key mask register (R1) that specifies which bits of the variable and fixed keys should be searched (Section 4.2.1) is initialized to zero, LTQ simply searches for a valid transaction in the transaction queue. Because of the annotation with the C flag, the LTQ instruction creates a command in the destination register (R9) and in the command address registers. Then, based on the valid bit of the command (now in R9), the LTQ instruction decides whether to enqueue the command in the command queue.

5.3. Example Firmware Code for FR-FCFS Command Scheduling Algorithm

An example code snippet for a higher-performance, first-ready, first-come, first-serve (FR-FCFS) [Rixner et al. 2000] policy is shown in Figure 10. FR-FCFS considers DRAM resource availability and the state of each transaction to reduce the overall latency of a DRAM access. The code uses an infinite loop to receive the next transaction and to generate the corresponding commands. In the body of the loop, a transaction is prioritized on the basis of the type of the next DRAM command it requires. A sequence

```
### FR-FCFS Scheduling Algorithm
# Initialization
       LD    R10,R0(0)       # key for ready CAS
       LD    R11,R0(1)       # mask for ready CAS
       LD    R12,R0(2)       # key for ready ACT
       LD    R13,R0(3)       # mask for ready ACT
       LD    R14,R0(4)       # key for ready PRE
       LD    R15,R0(5)       # mask for ready PRE
# Main loop
start: BTQE  start           # A: idle
       LTQ-C R1,R0,R10       # B: ready CAS
       BMSK  R1,R16, start   # restart
       LTQ-C R1,R0,R12       # C: ready ACT
       BMSK  R1,R16, start   # restart
       LTQ-C R1,R0,R14       # D: ready PRE
       JMP   start           # restart
```

Reg.	Value	Description
R10	4800	RDY and CAS flags
R11	4800	RDY and CAS flags
R12	2800	RDY and ACT flags
R13	2800	RDY and ACT flags
R14	1800	RDY and PRE flags
R15	1800	RDY and PRE flags
R16	8000	V flag

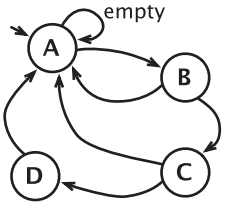


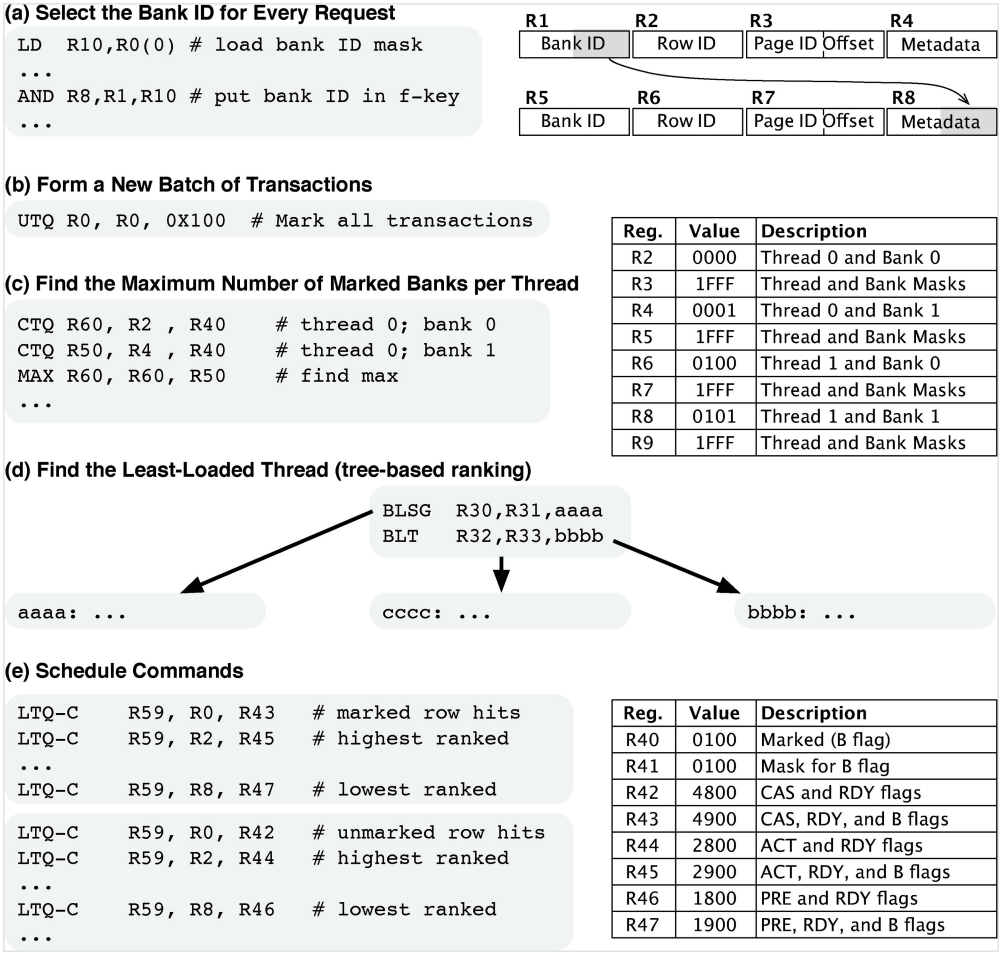
Fig. 10. Example transaction processing code for the FR-FCFS scheduling algorithm. The leftmost register in each line of code is the destination register.

of LTQ instructions are used to find matches for a specific variable key. The first LTQ instruction uses a pair of key and mask registers (R10, R11) holding a bit pattern that represents all transactions with a ready read or write command. (Recall from Section 4.2.3 that the register holding the bit mask is implicit, since the bit mask always resides in the next odd register following a key.) Therefore, this instruction searches for the oldest ready DRAM column access command, and issues the command to the command queue. The following instruction checks the valid bit of the command placed in R1, and starts scheduling the next command if a valid column access was found. If no ready read or write command was available, the next two instructions search for a valid activate command and issue it if found; otherwise, the code searches for a ready precharge command. Ready DRAM commands are prioritized over commands that are not ready by using the bit masks, while the order in which instructions are executed enforces a descending priority from column reads and writes to activate and precharge commands.

5.4. Example Firmware Code for Par-BS Command Scheduling Algorithm

An example code snippet for a fair and high-performance DRAM command scheduler, Parallelism-Aware Batch Scheduling (ParBS) [Mutlu and Moscibroda 2008], is shown in Figure 11. Unlike FCFS and FR-FCFS, ParBS services each thread's concurrent requests in parallel, thereby reducing the average slowdown per thread. (This requires a dedicated queue per bank for command scheduling. PARDIS can implement this feature because the transaction queue supports associative search operations and can emulate multiple queues.) To avoid starvation of the memory requests pending in the queues, ParBS processes memory requests in batches. This guarantees that a memory request within an existing batch is serviced before forming a new batch. ParBS employs two rules, known as the "max rule" and the "total rule", to rank threads within a batch based on their resource requirements. ParBS uses the thread ranks and the state of DRAM banks to generate DRAM commands while ordering commands as follows: (1) marked requests first; (2) row hit first; (3) higher ranked first; and (4) oldest first. The proposed firmware for ParBS consists of four main parts.

(1) *Bank ID selection.* Part of the code running on the request processor copies the bank ID of each memory request from the original address to the least significant byte



Reg.	Value	Description
R2	0000	Thread 0 and Bank 0
R3	1FFF	Thread and Bank Masks
R4	0001	Thread 0 and Bank 1
R5	1FFF	Thread and Bank Masks
R6	0100	Thread 1 and Bank 0
R7	1FFF	Thread and Bank Masks
R8	0101	Thread 1 and Bank 1
R9	1FFF	Thread and Bank Masks

Reg.	Value	Description
R40	0100	Marked (B flag)
R41	0100	Mask for B flag
R42	4800	CAS and RDY flags
R43	4900	CAS, RDY, and B flags
R44	2800	ACT and RDY flags
R45	2900	ACT, RDY, and B flags
R46	1800	PRE and RDY flags
R47	1900	PRE, RDY, and B flags

Fig. 11. Example code for the Par-BS scheduling algorithm. The leftmost register in each line of code is the destination register.

of the metadata.[1] (These eight bits hold the prefetch ID and application-defined priority fields when a request is first written to the request queue; ParBS firmware overwrites these fields.) Since the request metadata is automatically copied to the f-key at the time a transaction is written to the transaction queue (recall from Section 4.2.1), the transaction processor can access the transaction queue by associative search operations on a specific bank ID. This allows the transaction processor to emulate a controller with dedicated queues for each bank.

(2) *Batch formation.* The firmware uses the busy flag (B) of the variable keys to mark the memory transactions as a new batch. When a transaction is written to the transaction queue, its busy flag is set to zero by default; thereafter, a UTQ instruction can change the busy flag by searching and updating the transaction queue. As shown in Figure 11(b), a UTQ instruction uses R0 for both search keys, and

[1]This example code uses eight bits for bank IDs that enables addressing into up to 256 banks per channel. Support for higher number of banks is possible at the cost of firmware complexity.

marks all valid transactions in the queue with a bit mask (0X100).[2] A new batch is formed after all entries in the current batch are processed and removed from the transaction queue.

(3) *Max rule application.* ParBS applies the max rule to each newly formed batch of transactions to find the most heavily demanded bank for each thread. Based on this rule, a thread with the lowest number of requested banks has the highest priority. The example firmware implements this capability using CTQ, MIN, and MAX instructions as shown in Figure 11(c).

(4) *Total rule application.* The total load rule is applied to all threads for fine-grained ranking. Among all threads that are assigned the same rank by the max rule, a thread with a lower total load is ranked higher. A comparison tree, as shown in Figure 11(d) can efficiently implement this capability using control flow instructions BLSG and BLT.

(5) *DRAM command generation.* Based on the decisions made by the comparison tree from the previous step, the execution flow ultimately reaches a subroutine that schedules DRAM commands. On the one hand, this results in a simple and high-throughput software implementation of the ParBS scheduler; on the other hand, it increases the size of firmware.[3] Figure 11(e), shows an example command scheduling subroutine that prioritizes commands based on batch, row-hit, rank, and age.

5.5. Example Firmware Code for TCM Command Scheduling Algorithm

An example firmware implementation of the thread cluster memory scheduling (TCMS) [Kim et al. 2010b] is shown in Figure 12. TCMS is a fair and high-throughput memory scheduler that groups the application threads into two *memory-intensive* and *memory-nonintensive* clusters. Since the memory-nonintensive threads are sensitive to latency, TCMS prioritizes them to improve overall throughput. This, however, may result in long service delay for the memory-intensive cluster, thereby making the scheduling policy unfair. TCMS improves fairness in the memory-intensive cluster by shuffling the threads periodically. The proposed firmware code for TCMS consists of four main parts, as follows.

(1) *Initialization.* All required regular and special registers are initialized at the beginning. Regular registers hold constant values required for search keys and bit masks, while special registers are programmed to monitor the state of the memory subsystem. (Among 64 special registers, S0-S31 are dedicated to counter interrupts, and S32-S63 are used as statistics counters for monitoring the commands issued to the command queue.) In the TCMS firmware, two interrupt counters are programmed for thread clustering and shuffling (Figure 12(a)).

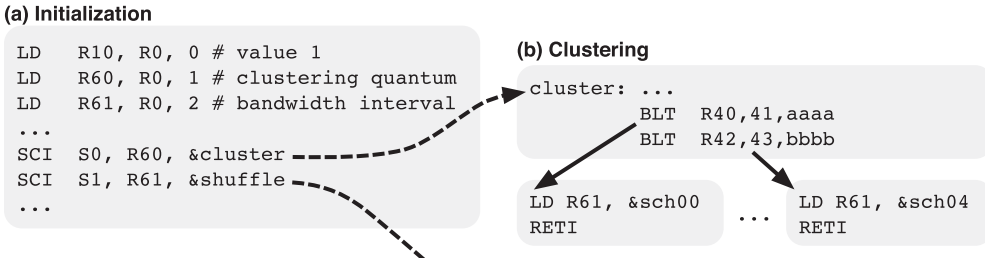
(2) *Thread clustering routine.* TCM requires observing the memory bandwidth usage and clustering threads periodically. As shown in Figure 12(b), the firmware provides an interrupt service routine called at "quantum" intervals to determine new clusters. First, all the required statistics are collected; based on the numbers, a decision tree then finds an appropriate order of threads. The outcome of the decision tree (stored in R61) is a pointer to a corresponding command scheduling routine.

(3) *Shuffling service routine.* Using simple ALU operations, the order of memory-intensive (bandwidth-sensitive) threads is shuffled. The outcome of this shuffling is a new value stored in R61 (Figure 12(c)).

(4) *Command scheduling routines.* Each scheduling routine is an infinite loop that schedules commands based on thread rank, row hit, and age. As shown in

[2]Updating the busy flag by a UTQ instruction does not affect other fields of the variable key.

[3]PARDIS supports transaction processing codes sizes up to 64KB.



Reg.	Description
R30	Bandwidth of thread 0 (no. transfers)
R31	Bandwidth of thread 1 (no. transfers)
R32	Bandwidth of thread 2 (no. transfers)
R33	Bandwidth of thread 3 (no. transfers)

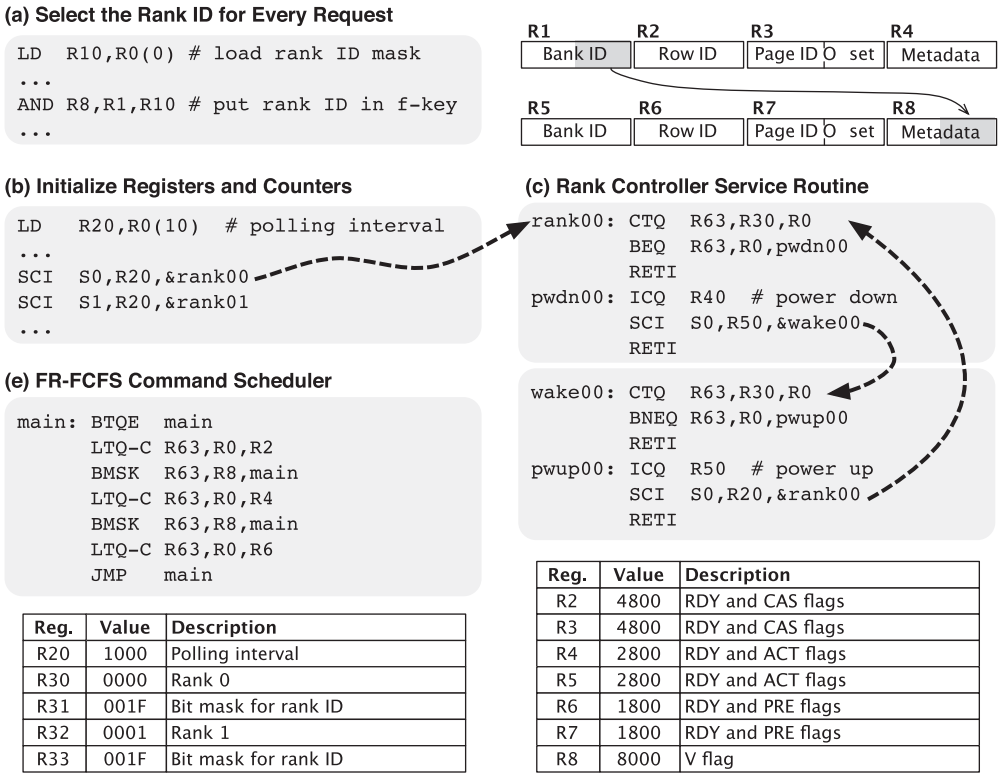
Reg.	Value	Description
R2	4800	RDY and CAS flags
R3	4800	RDY and CAS flags
R4	2800	RDY and ACT flags
R5	2800	RDY and ACT flags
R6	1800	RDY and PRE flags
R7	1800	RDY and PRE flags
R8	8000	V flag
R9	0001	Constant value 1
R10	0000	Thread 0
R11	1F00	Thread Mask
R12	0100	Thread 1
R13	1F00	Thread Mask
R14	0200	Thread 2
R15	1F00	Thread Mask
R16	0300	Thread 3
R17	1F00	Thread Mask

Fig. 12. Example code for the TCM scheduling algorithm. The leftmost register in each line of code is the destination register.

Figure 12(d), the last instruction of the loop is a JR instruction that jumps to the target stored in R61. Since this register is updated by the clustering and shuffling routines, the command scheduler can switch among different routines at the end of each loop iteration. Upon issuing a read or write command to the command queue, the scheduling routine increments a register (R30-R33) that tracks the bandwidth utilization of the corresponding thread. This register is used in the clustering routine to determine the memory-intensive and memory-nonintensive threads.

5.6. Example Firmware Code for DRAM Power Management

An example firmware for queue-aware DRAM power management, based on Hur and Lin's proposal [Hur and Lin 2008], is shown in Figure 13. This technique relies on transitioning to different power consumption modes to reduce the DRAM static power. Each DRAM rank is provided with a counter that records total idle time. For each rank, if the idle counter passes a predefined threshold and no pending memory request exists in the queue, the operational state changes to "low power". The rank remains in the low power state until a memory request in the queue requires access to the rank. In the



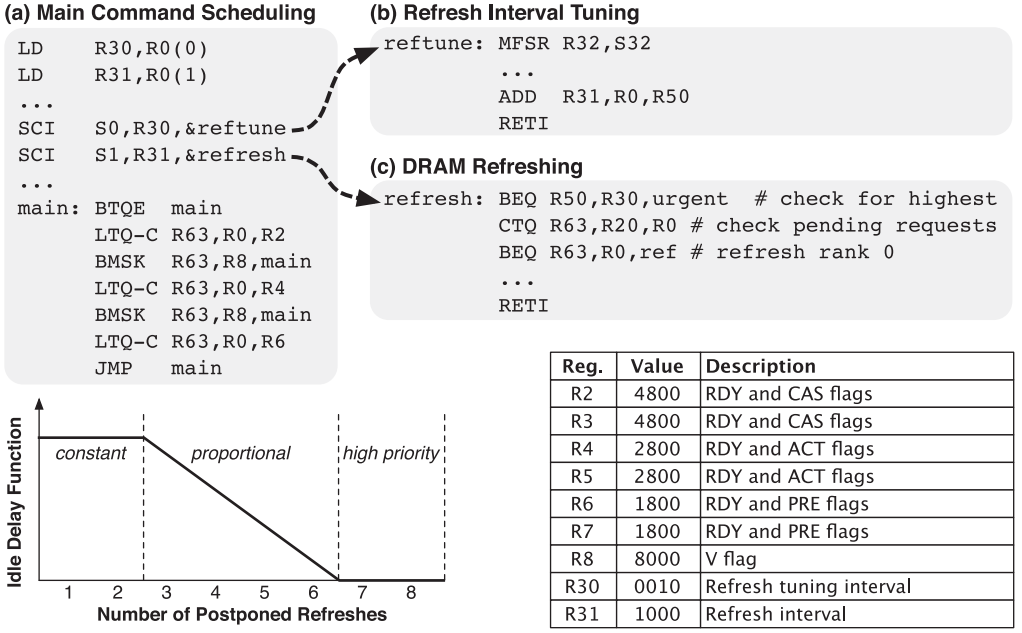
Reg.	Value	Description
R20	1000	Polling interval
R30	0000	Rank 0
R31	001F	Bit mask for rank ID
R32	0001	Rank 1
R33	001F	Bit mask for rank ID

Reg.	Value	Description
R2	4800	RDY and CAS flags
R3	4800	RDY and CAS flags
R4	2800	RDY and ACT flags
R5	2800	RDY and ACT flags
R6	1800	RDY and PRE flags
R7	1800	RDY and PRE flags
R8	8000	V flag

Fig. 13. Example transaction processing code for queue-aware power management. The leftmost register in each line of code is the destination register.

firmware example, queue-aware power management is applied on top of an FR-FCFS scheduling algorithm. The proposed code snippet has four main parts:

(1) *Rank ID selection.* Similar to ParBS firmware, part of the code running on the request processor copies the rank ID to the least significant byte of the metadata. This allows the transaction processor to access the transactions in the queue by associative search operations for specific bank IDs.
(2) *Register and counter initialization.* The transaction processing part of the code is initialized in Figure 13b, such that it supports a counter interrupt per DRAM rank. This allows the firmware to control each rank independently.
(3) *Rank controller service routine.* The power state of each DRAM rank is controlled by an interrupt service routine as shown in Figure 13(c). Each service routine periodically checks the state of the transaction queue and corresponding DRAM rank, and transitions to an appropriate power state. If no pending requests to the corresponding DRAM rank exists in the transaction queue, it transitions to a low power mode. A rank switches from a low power state to a high power state if it has at least one pending request in the transaction queue. In this example, a single counter interrupt is used to implement both power states and polling intervals. For example, the rank 0 controller consists of two subroutines labeled rank00 and wake00, which are accessed by a single counter interrupt (S0). At every transition



Reg.	Value	Description
R2	4800	RDY and CAS flags
R3	4800	RDY and CAS flags
R4	2800	RDY and ACT flags
R5	2800	RDY and ACT flags
R6	1800	RDY and PRE flags
R7	1800	RDY and PRE flags
R8	8000	V flag
R30	0010	Refresh tuning interval
R31	1000	Refresh interval

Fig. 14. Example transaction processing code for interrupt programming. The leftmost register in each line of code is the destination register.

to the low power state, S0 is reprogrammed for an appropriate polling interval and subroutine address.[4]

(4) *Command scheduling routine.* In this example we use the FR-FCFS command scheduling algorithm as the main routine; however, the rank service routines are general enough to apply to other command schedulers.

5.7. Example Firmware Code for DRAM Refresh Management

An example firmware for Elastic Refresh [Stuecheli et al. 2010] is shown in Figure 14. This technique exploits the flexibility of DRAMs ([Micron Technology, Inc. 2009a]) to postpone refresh operations. To mitigate refresh penalties due to conflicts with actual read and write accesses, the technique employs a predictive approach for refreshing DRAM devices in idle periods. This approach relies on an "Idle Delay Function" (IDF) shown in Figure 14, which defines three delay regions: *constant*, *proportional*, and *high priority*. As the number of postponed refresh commands increases, the delay between two consecutive check points for issuing refresh commands decreases; an urgent refresh becomes necessary after seven postponed refresh commands. The constant and proportional delay regions are dynamically tuned for different applications by monitoring online statistics, such as the number of idle intervals. The firmware consists of three routines:

(1) *Main command scheduling routine.* In the main routine, two interrupt counters (S0 and S1) are programmed for tuning the refresh interval and issuing refresh commands. The rest of the routine implements the FR-FCFS command scheduling algorithm.

[4]We studied the tradeoff between firmware execution overheads and state transition latency to find efficient polling intervals.

(2) *Tuning the refresh interval.* This routine implements IDF by periodically checking the registers, and counters to adjust the delay interval for calling the refresh management routine.
(3) *Refresh management routine.* This routine issues necessary DRAM refresh commands while trying to postpone refresh operations until no pending request exists in the queue. The routine records all postponed refresh operations for a rank in architectural registers that are read by the refresh interval tuner. It also defines two levels of priority for refreshing: *normal* and *urgent*. In the normal mode, refresh operations can be postponed; in the urgent mode, immediate refresh operations are required.

6. HARDWARE IMPLEMENTATION OF PARDIS

This article explores a scalar, pipelined implementation of PARDIS as depicted in Figure 15. The proposed implementation follows a six-step procedure for processing an incoming DRAM request, ultimately generating the corresponding DRAM command stream. A unique request ID (URID) is assigned 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. Although transactions enter the transaction queue in FIFO order, a queued transaction is typically prioritized based on fixed and variable keys. (3) After which the processor issues the next command of the transaction to the command queue. (4) Commands that are available in the command queue are processed by the command logic in FIFO order. (5) A DRAM command is only dequeued when it is ready to appear on the DDRx command bus. (6) Is issued to the DRAM subsystem at the next rising edge of the DRAM clock.

6.1. 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 of the pipeline, an instruction is fetched from the instruction memory. All branches are predicted taken,[5] and on a branch misprediction, the over-fetched wrong-path instruction is nullified. In the second stage, the fetched instruction is decoded to extract control signals, operands are read from the register file, and the next request is dequeued from the request queue if the instruction is annotated with an 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-R4) can only be written from the request queue side (on a dequeue), and are read-only to the request processor. In the third pipeline 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 of the pipeline, the result of every instruction is written back to the register file, and if the T flag of the instruction is set, a new transaction is enqueued at the transaction queue.

[5]Note that the request processor supports only a direct addressing mode for control flow instructions, which does not require address calculation.

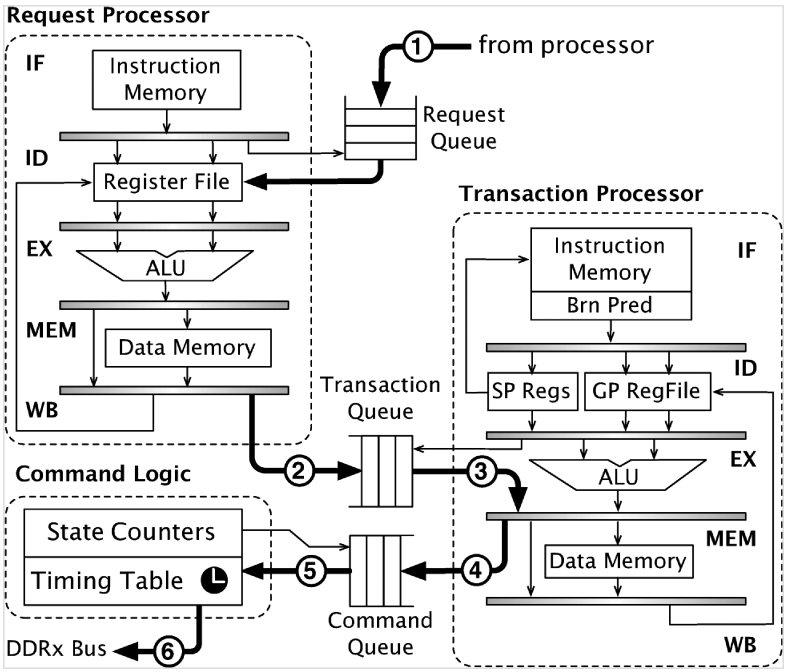


Fig. 15. Illustrative example of the proposed PARDIS implementation.

6.2. Transaction Processor

The transaction processor is a 16-bit, five-stage pipelined processor. In the first stage of the pipeline, the processor fetches the next instruction from a 64KB instruction memory. In the implementation, branch and jump instructions are divided into two categories: *fast* and *slow*. Fast branches include jump and branch on queue status instructions (BTQE and 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 8K-entry g-share branch predictor. Critical branches in the transaction processor are usually coded using the fast branch instructions (e.g., infinite scheduling loops, or queue state checking).

In the second pipeline stage, the instruction is decoded, general- and special-purpose registers are read, and special-purpose interrupt registers are set. Special purpose registers are implemented using a 64-entry array of programmable counters. In the proposed implementation of PARDIS, 32 of these programmable counters (S0-S31) are used for timer interrupts, and the remaining 32 programmable counters (S32-S63) are used for collecting statistics to aid in decision-making (Figure 16).

For every timer, there are two registers holding the interrupt service routine address and the maximum counter value after which an interrupt must fire. Every time the counter resets, an interrupt is fired and latched in an interrupt flop. There is a descending priority from S0 to S63 among all interrupt timers. To prevent nested interrupts, a busy flag masks all other interrupts until the current interrupt finishes with a RETI instruction, which resets the busy flag and the corresponding interrupt flop.

After decoding, a 16-bit ALU performs arithmetic and logic operations the transaction queue is accessed; in parallel Figure 17 shows the proposed architecture of the transaction queue comprising five components: (1) five 64-entry content-addressable

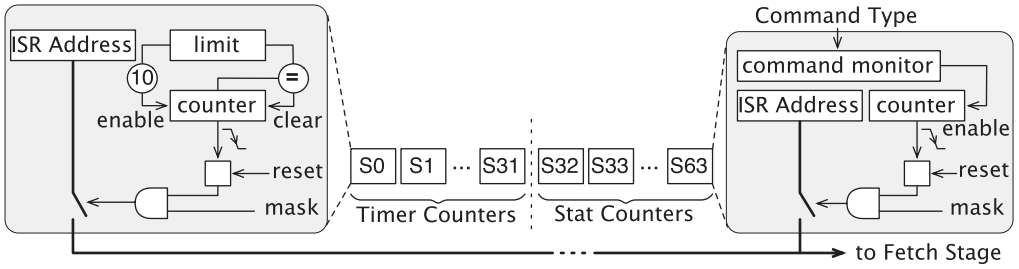


Fig. 16. Interrupt counters in the proposed PARDIS implementation.

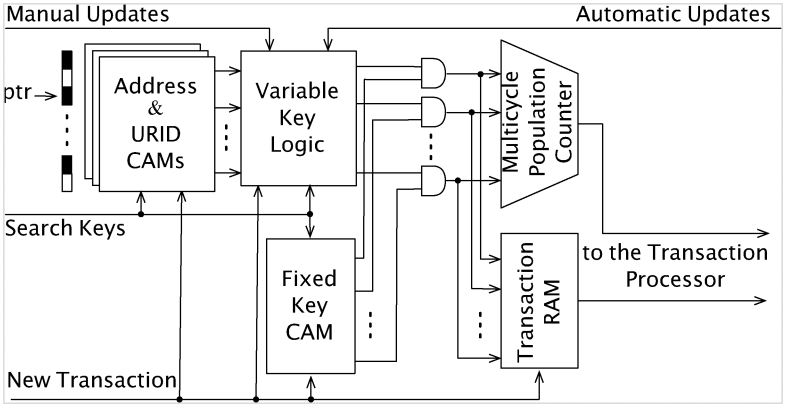


Fig. 17. The proposed architecture of the transaction queue.

memories (CAMs), one each for the rank, bank, row, column, and unique request IDs; (2) a 64-entry CAM storing variable keys; (3) a 64-bit population counter; (4) a 64-entry CAM holding fixed keys; and (5) a 64 $\times$ 86 bit RAM holding a copy of the fixed data for the transaction (i.e., the address, the fixed key, and the URID). The transaction queue is accessible in four ways:

(1) *Adding a new transaction.* If the transaction queue is not full, a new transaction is written to the transaction queue by updating the content of the address and URID CAMs, variable keys, fixed keys, and the transaction data. Even though transactions are allowed to leave the transaction queue out of order, the transaction queue employs a circular enqueuing technique that maintains an oldest-first order among occupied entries.
(2) *Searching for a transaction.* For all instructions that need to search the transaction queue, the fixed and variable key CAMs are accessed with the corresponding search keys. Every key is accompanied by a mask indicating which subset of the bits within the key should contribute to the search (other bit positions are ignored by hardware). The fixed and variable key CAMs provide match results to the transaction RAM (for retrieving the DRAM address to be accessed by the selected transaction) and to the population count logic (for counting the number of matches).
(3) *Updating the variable keys.* The variable key logic receives updates to the variable key from the transaction processor and command logic. Updates to the software-managed region of the variable key are generated by a UTQ instruction, whereas the hardware managed region is automatically updated after every state change.

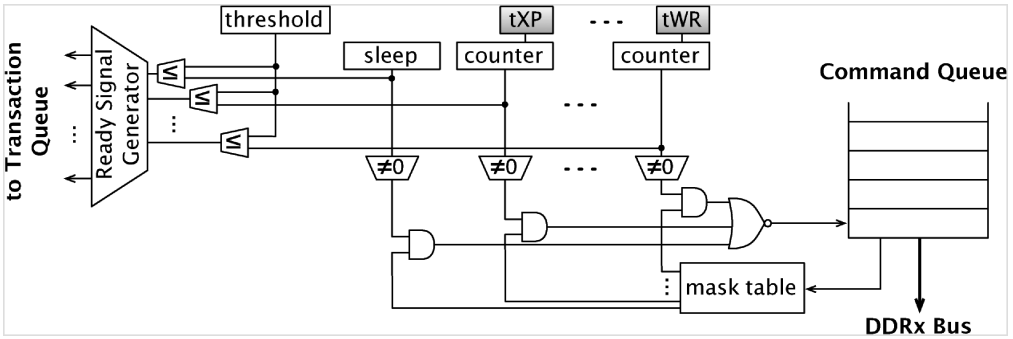


Fig. 18. Illustrative example of the proposed command logic for PARDIS.

(4) *Reading search results.* After a search, the number of matching transactions can be obtained from a population counter, and the DRAM address of the highest-priority matching transaction can be obtained from a transaction RAM.

Command queue and data memory accesses occur in the fourth stage of the pipeline, and the result of the instruction is written back to the register file in the fifth stage.

6.3. Command Logic

The command logic (Figure 18) is implemented by using 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. At every DRAM cycle, the command at the head of the command queue is inspected, and a bit mask is retrieved from the masking table to mask out timing constraints that are irrelevant to the command under consideration (e.g., tCL 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.

7. EXPERIMENTAL SETUP

We evaluate the performance potential of PARDIS by comparing fixed-function hardware and PARDIS-based firmware implementations of FCFS [Rixner et al. 2000]; FR-FCFS [Rixner et al. 2000]; Par-BS [Mutlu and Moscibroda 2008]; and TCMS [Kim et al. 2010b] scheduling algorithms. We also implement in firmware a recent DRAM power management algorithm proposed by Hur and Lin [2008], 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 [Stuecheli et al. 2010] to its firmware implementation. 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 SESC simulator [Renau et al. 2005]. We measure the area, frequency, and power dissipation of PARDIS by implementing the proposed system in Verilog HDL, and synthesizing the proposed hardware.

7.1. Architecture

We modify the SESC simulator [Renau et al. 2005] to model an eight-core system with a 4MB L2 cache and two on-chip memory controllers. Table III shows the simulation parameters. In the simulated configuration, memory channels are fully populated

Table III. Simulation Parameters

Core	8 4-issue cores, 2.0 GHz
Functional units	Int/FP/Ld/St/Br units 2/2/2/2/2, Int/FP Mult 1/1
IQ, LSQ, ROB size	IssueQ 32, LoadQ/StoreQ 24/24, ROB 96
Physical registers	Int/FP 96/96
Branch predictor	Hybrid, local/global/meta 2K/2K/8K, 512-entry direct-mapped BTB, 32-entry RAS
IL1 cache (per core)	32KB, direct-mapped, 32B block, hit/miss delay 2/2
DL1 cache (per core)	32KB, 4-way, LRU, 32B block, hit/miss delay 3/3, MESI protocol
L2 cache (shared)	4MB, 8-way, LRU, 64B block, hit/miss delay 24/24
PARDIS	request/transaction/command queue size: 64/64/64
DRAM Subsystem [Micron Technology, Inc. 2009a]	8Gb DDR3-1066 chips, 2 Channels, 4 Ranks/Channel, 8 Banks/Rank, tRCD: 7, tCL: 7, tWL: 6, tCCD: 4, tWTR: 4, tWR: 8, tRTP: 4, tRP: 7, tRRD: 4, tRAS: 20, tRC: 27, tBURST: 4, tFAW: 20, IDD0: 1314, IDD1: 1584, IDD2P: 288, IDD2N: 1620, IDD3P: 1080, IDD3N: 1800, IDD4R: 2304, IDD4W: 2304, IDD5B: 3297, IDD6: 216

with DIMMs (typical of server systems [Hur and Lin 2008]), which restricts the maximum channel data rate to 800MT/s for DDR3-1066 [Micron Technology, Inc. 2009c, 2009a; Hewlett-Packard Development Company, L. P. 2010]. This results in a core-to-DRAM clock ratio of five. Energy results for the DRAM subsystem are generated based on DDR3-1066 product data from Micron[Micron Technology, Inc. 2009a]. Evaluated baseline controllers have the same queue sizes as PARDIS (64 entries each); they observe pending requests at the beginning of a DRAM clock cycle, and make scheduling decisions by the end of the same cycle. (In PARDIS, this is not always the case because policies are implemented in firmware.)

7.2. Applications

Evaluated parallel workloads represent a mix of thirteen data-intensive applications from Phoenix [Yoo et al. 2009]; SPLASH-2 [Woo et al. 1995]; SPEC OpenMP [Dagum and Menon 1998]; NAS [Bailey et al. 1994]; and Nu-MineBench [Narayanan et al. 2006] suites. Table IV summarizes the evaluated benchmarks and their input sets. All applications are simulated to completion.

7.3. Synthesis

We evaluate the area and power overheads of the proposed architecture by implementing it in Verilog HDL and synthesizing the design using Cadence Encounter RTL Compiler [Cadence] with FreePDK [FreePDK] at 45nm. The results are then scaled to 22nm (relevant parameters are shown in Table V). Instruction and data memories are evaluated using CACTI 6.0 [Wilton and Jouppi 1996], while register files and CAMs are modeled through SPICE simulations with the FabMem toolset from FabScalar [Choudhary et al. 2011].

8. EVALUATION

We first present synthesis results on the area, power, and delay contributions of various hardware components in PARDIS. Next, we compare fixed-function hardware and PARDIS-based firmware implementations of existing scheduling policies, address-mapping techniques, power management algorithms, and refresh scheduling

Table IV. Applications and Data Sets

Benchmarks	Suite	Input
Histogram	Phoenix	34,843,392 pixels (104MB)
String Match	Phoenix	50MB non-encrypted file
Word Count	Phoenix	10MB text file
Linear Regression	Phoenix	50MB key file
ScalParC	NU-MineBench	125K pts., 32 attributes
MG	NAS OpenMP	Class A
CG	NAS OpenMP	Class A
Swim-Omp	SPEC OpenMP	MinneSpec-Large
Equake-Omp	SPEC OpenMP	MinneSpec-Large
Art-Omp	SPEC OpenMP	MinneSpec-Large
Ocean	SPLASH-2	514 × 514 ocean
FFT	SPLASH-2	1M points
Radix	SPLASH-2	2M integers

Table V. Technology Parameters
[ITRS; Zhao and Cao 2006]

Technology	Voltage	FO4 Delay
45nm	1.1 V	20.25ps
22nm	0.83 V	11.75ps

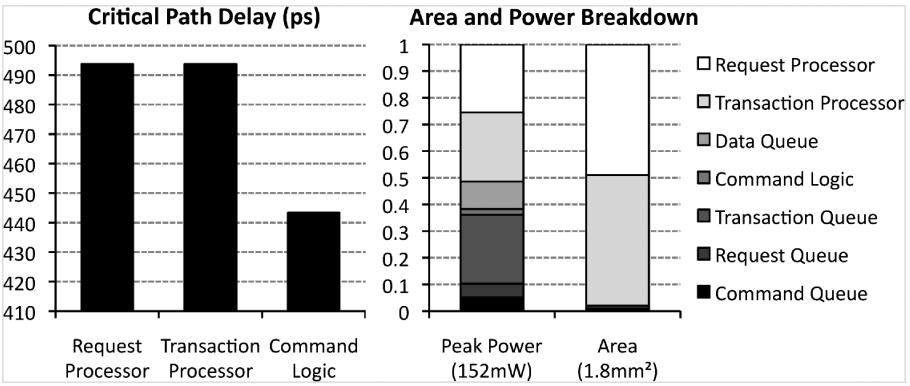


Fig. 19. Delay, area, and peak power characteristics of the synthesized PARDIS implementation.

mechanisms. We then evaluate the impact of a set of application-specific address-mapping heuristics enabled by PARDIS.

8.1. Area, Power, and Delay: Where Are the Bottlenecks?

Synthesis results on the area, power, and delay contributions of different hardware components are shown in Figure 19. A fully synthesized implementation of PARDIS operates at over 2GHz, occupies 1.8mm^2 of die area, and dissipates 152mW 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 area is occupied by the request and transaction processors because of four 64KB instruction and data SRAMs; however, the transaction queue, which implements associative lookups using CAMs, is the most power-hungry component (29%). Other major consumers of peak power are the transaction processor (29%) and the request processor (28%).

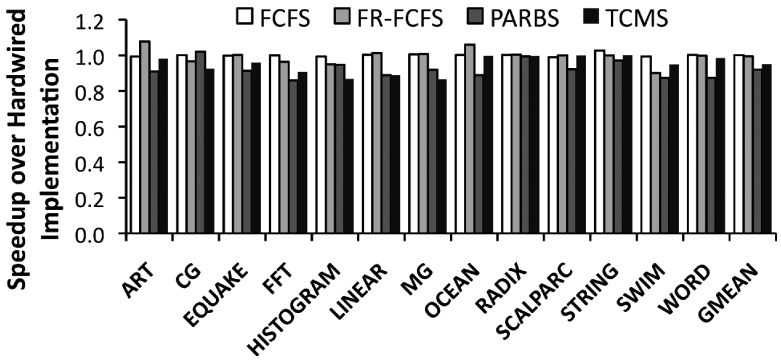


Fig. 20. Performance of PARDIS-based and hardwired implementations for FCFS, FR-FCFS, PARBS, and TCMS scheduling algorithms.

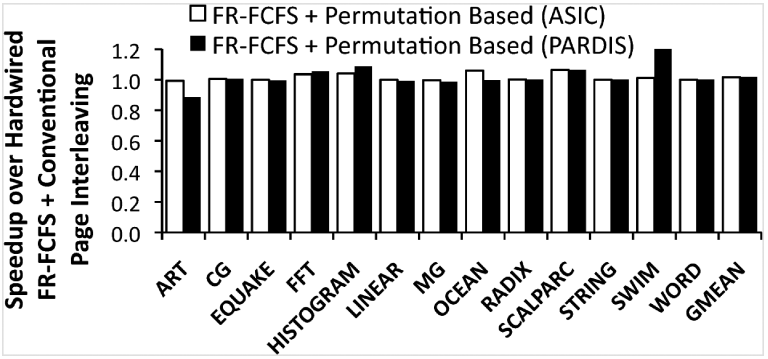


Fig. 21. Performance of PARDIS-based and hardwired implementations of permutation-based address mapping.

8.2. Scheduling Policies

Figure 20 compares PARDIS-based firmware implementations of FCFS [Rixner et al. 2000]; FR-FCFS [Rixner et al. 2000]; Par-BS [Mutlu and Moscibroda 2008]; and TCMS [Kim et al. 2010b] 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 (e.g., 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 decision-making when using PARDIS, which generally results in greater queue occupancies. As a result of having more requests to choose from, the scheduling algorithm is able to exploit bank parallelism and row buffer locality more aggressively under the PARDIS implementation. However, for Par-BS and TCMS—two compute-intensive scheduling algorithms—PARDIS suffers from higher processing latency, and hurts performance by 8% and 5%, respectively.

8.3. Address Mapping

To evaluate the performance of different DRAM address-mapping techniques on PARDIS, the permutation-based interleaving [Zhang et al. 2000] technique was mapped onto PARDIS and compared to its fixed-function hardware implementation (Figure 21). The average performance of the two implementations differ by less than

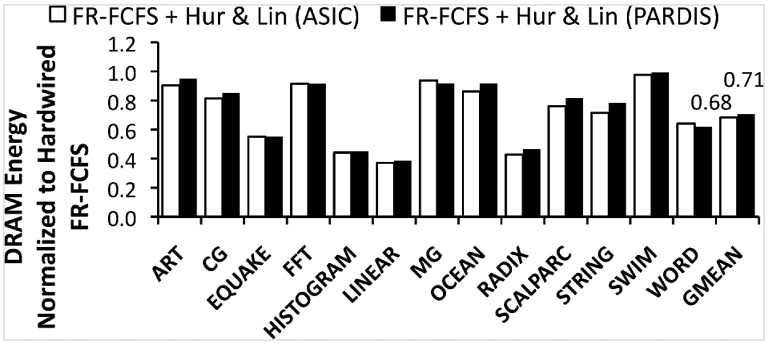


Fig. 22. DRAM energy comparison between the PARDIS-based and hardwired implementations of the queue-aware power management technique.

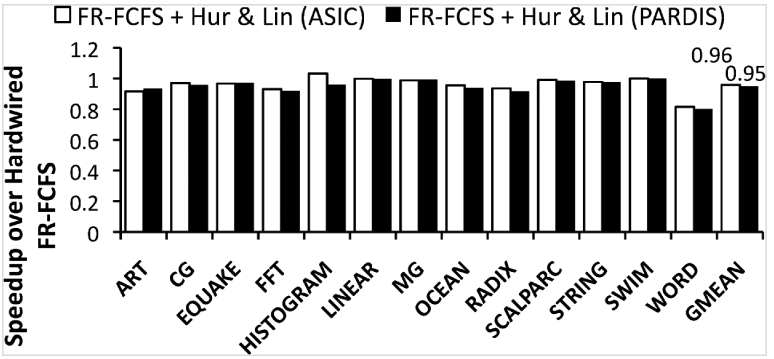


Fig. 23. Performance of PARDIS-based and hardwired implementations for power management technique.

1%; interestingly, PARDIS outperforms fixed-function hardware by a small margin on some applications. As explained in Section 8.2, PARDIS incurs a higher transaction processing latency, which results in a higher transaction queue occupancy. In a scheduling algorithm that searches for specific commands (e.g., FR-FCFS, which searches for row hits), increasing the number of candidate commands sometimes improves performance (SWIM, FFT, and HISTOGRAM in Figure 21). Other applications, such as ART and OCEAN, do not benefit from this phenomenon.

8.4. Power Management

DRAM power management with PARDIS was evaluated by implementing Hur and Lin's queue-aware power management technique [Hur and Lin 2008] in firmware, and comparing the results to a fixed-function hardware implementation (Figure 22); in both cases, the underlying command scheduling algorithm is FR-FCFS. The hardwired implementation reduces average DRAM energy by 32% over conventional FR-FCFS at the cost of 4% lower performance. The firmware implementation of queue-aware power management with PARDIS shows similar results: 29% DRAM energy savings are obtained at the cost of a 5% performance loss (Figures 22 and 23).

8.5. Refresh

In order to evaluate DRAM refresh management on PARDIS, a conventional on-demand DDR3 refresh method [Micron Technology, Inc. 2009a] is considered as the baseline to which fixed-function hardware and PARDIS-based firmware

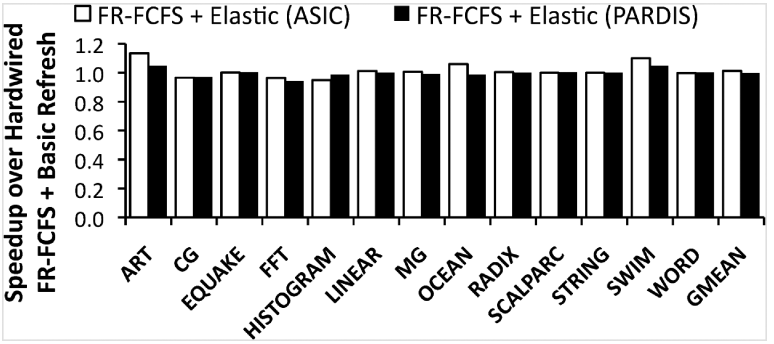


Fig. 24. Performance of PARDIS-based and hardwired implementations of the elastic refresh scheduling algorithm.

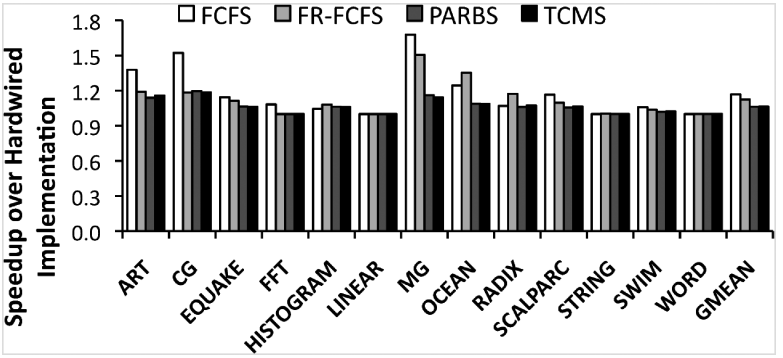


Fig. 25. Speedup over hardwired permutation-based interleaving [Zhang et al. 2000] using application-specific address mapping on PARDIS.

implementations of the recently proposed Elastic Refresh algorithm [Stuecheli et al. 2010] are compared (Figure 24). 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 1% of fixed-function hardware.

8.6. Application Specific Optimizations

A hardwired address-mapping scheme uses a fixed-mapping function to distribute memory accesses among DRAM banks; however, higher bank-level parallelism and row buffer hit rates can be achieved by defining a custom mapping function for each application based on profiling analysis. We define a profiling dataset for each application and evaluate the execution time when using different bit positions to index DRAM channels, ranks, and banks. (To cull the design space, we require each DRAM coordinate to comprise a set of adjacent bits.) After finding the best scheme for each application, we run new simulations based on the reference data sets to report the execution time and DRAM energy consumption.[6] As shown in Figure 25, application-specific DRAM indexing improves performance by 17%, 14%, 7%, and 6% over permutation-based interleaving [Zhang et al. 2000] for FCFS, FR-FCFS, Par-BS, and TCMS, respectively; corresponding DRAM energy savings are 22%, 14%, 9%, and 9% (Figure 26).

[6]We assume that the firmware is provided by the system and configured according to the needs of each application by the OS. User-level programming interfaces [Reinhardt et al. 1994] are left for future work.

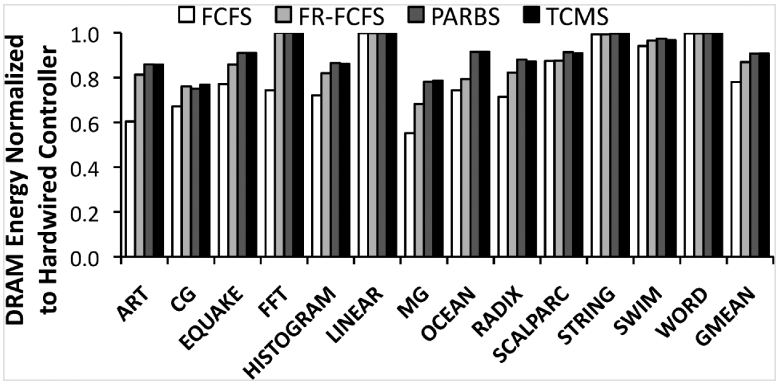


Fig. 26. DRAM energy savings over hardwired permutation-based interleaving [Zhang et al. 2000] using application-specific address mapping on PARDIS.

9. RELATED WORK

PARDIS builds upon existing work in high-performance memory systems.

9.1. DDRx Controller Optimizations

Numerous DDRx controller optimizations for improving performance, energy, and QoS have been published in the literature [Mutlu and Moscibroda 2008; Rixner et al. 2000; Kim et al. 2010b; Hur and Lin 2008; Stuecheli et al. 2010; Kim et al. 2010a; Diniz et al. 2007; Zheng et al. 2008; Isen and John 2009; Sudan et al. 2010; Liu et al. 2011; Stuecheli et al. 2010]. Unlike PARDIS, these proposals address specific workload classes (multiprogrammed, parallel, or real-time); yet a hardwired memory controller is neither able to meet the requirements of a diverse set of applications optimally, nor can it change its objective function for a new optimization target. In addition, the emergence of new memory technologies (e.g., PCM) creates new opportunities for energy, wear-out, and performance optimization, which are difficult to exploit within an existing hardwired controller. On the other hand, PARDIS provides significant flexibility in supporting a diverse set of capabilities through firmware-based programmable control, ease of applying revisions to the implemented memory controllers through firmware patches, and configurability in interfacing to different media.

9.2. Programmable Cache and Directory Controllers

Programmability is a well-known concept that has been broadly applied to memory systems. FLASH [Kuskin et al. 1994] is a multiprocessor platform that introduces a general-purpose processor, called MAGIC, for executing directory protocols. Typhoon [Reinhardt et al. 1994] is a programmable architecture that supports Tempest—a message-passing protocol. Alewife [Agarwal et al. 1995] allows performance tuning through configuration of the cache coherence protocol. Smart Memories [Firoozshahian et al. 2009] is a framework for designing cache coherent memory components connected via an on-chip network. The focus of these proposals is on caches and directories, not on managing internal DRAM resources. In contrast, PARDIS proposes a fully programmable framework that provides application-specific control of the DRAM subsystem.

9.3. Intelligent DRAM Controllers

Intelligent memory controllers have been proposed to provide a degree of configurability to the memory system. Impulse [Carter et al. 1999] is a memory controller

that provides configurable access to memory blocks via physical address remapping to accelerate special functions (e.g., matrix transpose). Other proposals introduce programmability into controllers for on-chip SRAMs and DMA engines [Martin et al. 2009; Kornaros et al. 2003], or allow choosing among predefined QoS-aware scheduling algorithms for a DDRx memory controller [Lee et al. 2005]. Recently proposed RL-based memory controllers [Ipek et al. 2008; Mukundan and Martinez 2012] introduce the new concept of self-optimization to DRAM command scheduling, exploiting reinforcement learning techniques. An RL-based memory controller successfully implements a hardwired but adaptive algorithm for DDR2 memory controllers. To the best of our knowledge, PARDIS is the first fully programmable DRAM memory controller that allows for managing the request and command streams in software.

10. CONCLUSIONS

We have presented PARDIS, a programmable memory controller that can meet the performance requirements of a high-speed DDRx interface. We have seen that it is possible to achieve performance within 8% of a hardwired memory controller when contemporary address-mapping, command scheduling, refresh management, and DRAM power management techniques are mapped onto PARDIS. We have also observed 6–17% performance improvements and 9–22% DRAM energy savings by using application-specific address-mapping heuristics enabled by PARDIS. We conclude that programmable DDRx controllers hold the potential to significantly improve the performance and energy-efficiency of future computer systems.

REFERENCES

Agarwal, A., Bianchini, R., Chaiken, D., Kranz, D., Kubiatowicz, J., Hong Lim, B., MacKenzie, K., and Yeung, D. 1995. The MIT alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 2–13.

Bailey, D. H. et al. 1994. NAS parallel benchmarks. Tech. rep. RNR-94-007, NASA Ames Research Center.

Browne, M., Aybay, G., Nowatzyk, A., Dubois, M., and Member, S. 1998. Design verification of the s3.mp cache coherent shared-memory system. *IEEE Trans. Comput.*

Cadence. Encounter RTL compiler. http://www.cadence.com/products/ld/rtl-compiler/.

Carter, J., Hsieh, W., Stoller, L., Swanson, M., Zhang, L., Brunvand, E., Davis, A., Kuo, C.-C., Kuramkote, R., Parker, M., Schaelicke, L., and Tateyama, T. 1999. Impulse: Building a smarter memory controller. In *Proceedings of the International Symposium 5th HPCA. High-Performance Computer Architecture*. 70–79.

Choudhary, N. K., Wadhavkar, S. V., Shah, T. A., Mayukh, H., Gandhi, J., Dwiel, B. H., Navada, S., Najaf-Abadi, H. H., and Rotenberg, E. 2011. Fabscalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, 11–22.

Dagum, L. and Menon, R. 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng. 5*, 1, 46–55.

Diniz, B., Guedes, D., Meira,W., Jr., and Bianchini, R. 2007. Limiting the power consumption of main memory. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 290–301.

Firoozshahian, A., Solomatnikov, A., Shacham, O., Asgar, Z., Richardson, S., Kozyrakis, C., and Horowitz, M. 2009. A memory system design framework: Creating smart memories. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, 406–417.

FreePDK. Free PDK 45nm open-access based PDK for the 45nm technology node. http://www.eda.ncsu.edu/wiki/FreePDK.

Hewlett-Packard Development Company, L. P. 2010. DDR3 memory technology. http://h20195.www2.hp.com/v2/GetPDF.aspx/c01750914.pdf.

Hur, I. and Lin, C. 2008. A comprehensive approach to dram power management. In *Proceedings of HPCA'08*. 305–316.

Ipek, E., Mutlu, O., Martinez, J., and Caruana, R. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the International Symposium on Computer Architecture*.

ISEN, C. AND JOHN, L. 2009. Eskimo - Energy savings using semantic knowledge of inconsequential memory occupancy for dram subsystem. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*. 337–346.

ITRS. International Technology Roadmap for Semiconductors: 2010 Update. http://www.itrs.net/links/2010itrs/home2010.htm.

JACOB, B. L., NG, S. W., WANG, D. T., AND WANG, D. T. 2008. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.

KIM, Y., HAN, D., MUTLU, O., AND HARCHOL-BALTER, M. 2010a. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*. 1–12.

KIM, Y., PAPAMICHAEL, M., MUTLU, O., AND HARCHOL-BALTER, M. 2010b. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. IEEE, Los Alamistos, CA, 65–76.

KORNAROS, G., PAPAEFSTATHIOU, I., NIKOLOGIANNIS, A., AND ZERVOS, N. 2003. A fully programmable memory management system optimizing queue handling at multi gigabit rates. In *Proceedings of the Design Automation Conference*. 54–59.

KUSKIN, J., OFELT, D., HEINRICH, M., HEINLEIN, J., SIMONI, R., GHARACHORLOO, K., CHAPIN, J., NAKAHIRA, D., BAXTER, J., HOROWITZ, M., GUPTA, A., ROSENBLUM, M., AND HENNESSY, J. 1994. The Stanford flash multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*. IEEE, Los Alamitos, CA, 302–313.

LEE, K.-B., LIN, T.-C., AND JEN, C.-W. 2005. An efficient quality-aware memory controller for multimediaplatform soc. *IEEE Trans. Circuits Syst. Video Technol. 15*, 5, 620–633.

LIU, S., PATTABIRAMAN, K., MOSCIBRODA, T., AND ZORN, B. G. 2011. Flikker: Saving DRAM refresh-power through critical data partitioning. In *Proceedings of ASPLOS*, R. Gupta and T. C. Mowry, Eds., ACM, New York, 213–224.

MARTIN, J., BERNARD, C., CLERMIDY, F., AND DURAND, Y. 2009. A microprogrammable memory controller for high-performance dataflow applications. In *Proceedings of ESSCIRC (ESSCIRC'09)*. 348–351.

MICRON TECHNOLOGY, INC. 2009a. 8Gb DDR3 SDRAM. Micron Technology, Inc. http://www.micron.com//getdocument/?documentId=416.

MICRON TECHNOLOGY, INC. 2009b. TN-29-14: Increasing NAND flash performance functionality. Micron Technology Inc. http://www.micron.com/getdocument/?documentId=140.

MICRON TECHNOLOGY, INC. 2009c. TN-41-08: design guide for two DDR3-1066 UDIMM systems introduction. Micron Technology, Inc. http://www.micron.com//document download/?documentId=4297.

MUKUNDAN, J. AND MARTINEZ, J. F. 2012. Morse: Multi-objective reconfigurable self-optimizing memory scheduler. In *Proceedings of the IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA'12)*. IEEE, Los Alamitos, CA, 1–12.

MUTLU, O. AND MOSCIBRODA, T. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ACM, New York, 32–41.

NARAYANAN, R., ET AL. 2006. Minebench: A benchmark suite for data mining workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*.

REINHARDT, S. K., LARUS, J. R., AND WOOD, D. A. 1994. Tempest and typhoon: User-level shared memory. In *Proceedings of ISCA-21*. 325–336.

RENAU, J., ET AL. 2005. SESC simulator. http://sesc.sourceforge.net.

RIXNER, S., ET AL. 2000. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*.

STUECHELI, J., KASERIDIS, D., HUNTER, H. C., AND JOHN, L. K. 2010. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *Proceedings of MICRO*. 375–384.

SUDAN, K., CHATTERJEE, N., NELLANS, D., AWASTHI, M., BALASUBRAMONIAN, R., AND DAVIS, A. 2010. Micro-pages: increasing dram efficiency with locality-aware data placement. In *Proceedings of ASPLOS'10*. 219–230.

WILTON, S. AND JOUPPI, N. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE J. Solid-State Circuits 31*, 5, 677–688.

WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of ISCA-22*.

Yoo, R. M., Romano, A., and Kozyrakis, C. 2009. Phoenix rebirth: Scalable MapReduce on a large-zscale shared-memory system. In *Proceedings of the IEEE International Symposium on Workload Characterization.*

Zhang, Z., Zhu, Z., and Zhang, X. 2000. A permutation-based page interleaving scheme to reduce row buffer conflicts and exploit data locality. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*. ACM, New York, 32–41.

Zhao, W. and Cao, Y. 2006. New generation of predictive technology model for sub-45nm design exploration. In *Proceedings of the International Symposium on Quality Electronic Design*.

Zheng, H., Lin, J., Zhang, Z., Gorbatov, E., David, H., and Zhu, Z. 2008. Mini-rank: Adaptive dram architecture for improving memory power efficiency. In *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*. IEEE, Los Alamitos, CA, 210–221.

Received December 2012; revised June 2013; accepted June 2013