PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards

Mahdi Nazm Bojnordi and Engin Ipek
University of Rochester
Rochester, NY 14627 USA
{bojnordi, ipek}@ece.rochester.edu

Abstract

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 paper 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 6-17% and reduces DRAM energy by 9-22% over four existing memory controllers.

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 multi-objective 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 [1, 2] to NAND Flash [3] and directory [4, 5, 6, 7, 8, 9] control. 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 paper 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.

978-1-4673-0476-4/12/$31.00(c)2012 IEEE

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 6-17$\%$ and DRAM energy by 9-22$\%$ 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 common 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. 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 comprises a *request processor*, a *transaction processor*, and *command logic*—three tightly-coupled processing elements that 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).

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.

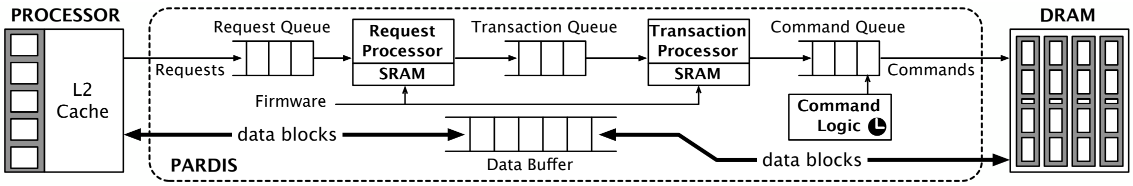

Figure 1. **Illustrative example of PARDIS in a computer system.**

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.

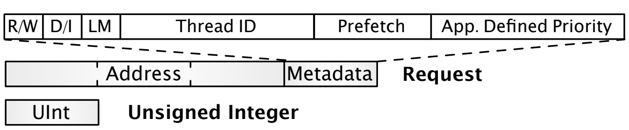

Figure 2. **Data types supported by the request processor.**

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.

4.1.3 Instructions

As depicted in Figure 3, the request processor supports 14 32-bit instructions of four different types.

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.

Control Flow. 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. The target address of a branch is a 16-bit immediate value, which is an absolute address to the instruction memory.

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

Queue Access. 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 copying different fields of the 64-bit request into general purpose registers with the R flag, operating on these fields to compute channel, rank, bank, row, and column IDs, and 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.

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`

Figure 3. **Instructions supported by the request processor.**

4.1.4 Example Firmware Code: 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 [10], or can operate on R1-R4 to implement more sophisticated address mapping heuristics. Figure 4 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.

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

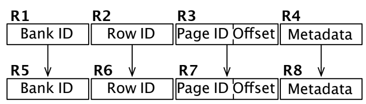

Figure 4. **Illustrative example of page interleaving on the request processor. The destination register in each line of code is the leftmost register.**

As a second example of address mapping at the request processor, an implementation of permutation based page interleaving [11] is shown in Figure 5. 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 in a transaction register. The remaining instructions copy source request registers to destination transaction registers, and enqueue a transaction at the transaction queue.

```
### Permutation Based Address Mapping
# Initialization
    LD    R10,R0(0)  # load bit mask for bank ID
    LD    R11,R0(1)  # load shift amount for alignment
# Main Loop
st: 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   st
```

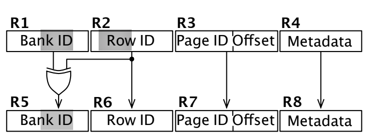

Figure 5. **Illustrative example of permutation based address mapping on the request processor. The destination register in each line of code is the leftmost register.**

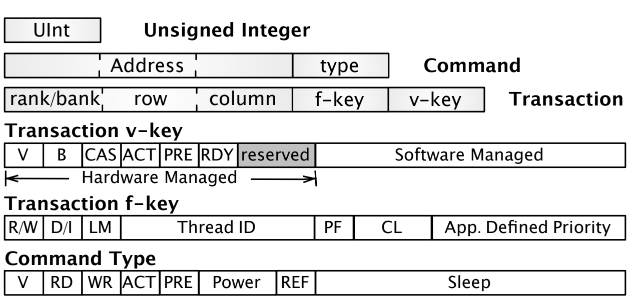

Figure 6. **Data types supported by the transaction processor.**

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 6), and a *variable key* (v-key in Figure 6). 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 5). 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 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 the 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 multi-cycle 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.

4.2.3 Instructions

The transaction processor provides 30 instructions comprising ALU, control flow, memory access, interrupt processing, and queue access operations (Figure 4.2.3).

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 7. **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.

Control Flow. Ten 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 "return from an interrupt service routine" (RETI) instructions.

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

Interrupt Programming. 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 pre-programmed 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 to determine 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. 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 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 even numbered register, and the following odd numbered register is used to store a bit mask that determines which bits from the key should contribute to the search. 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 for the selected transaction (Figure 6) 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 6) 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 [12] 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.

4.2.4 Example Firmware Code: FCFS and FR-FCFS Scheduling

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 8. 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 4.3.3) 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.

```
### FCFS Scheduling Algorithm
# Initialization
    XOR    R1,R1,R1 # reset the key mask
# Main loop
st: BTQE   st       # A: wait for trans.
    LTQ-C  R9,R0,R0 # B: issue oldest command
    JMP    st       # goto next transaction
```

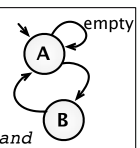

Figure 8. **Example transaction processing code for FCFS scheduling algorithm. The leftmost register in each line of code is the destination register.**

A second example code snippet for a higher-performance, first-ready first-come first-serve (FR-FCFS) [13] policy is shown in Figure 9. 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 based on the type of the next DRAM command it requires. A sequence 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.

```
### 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
st: BTQE   st           # A: idle
    LTQ-C  R1,R0,R10    # B: ready CAS
    BMSK   R1,valid, st # restart
    LTQ-C  R1,R0,R12    # C: ready ACT
    BMSK   R1,valid, st # restart
    LTQ-C  R1,R0,R14    # D: ready PRE
    JMP    st           # restart
```

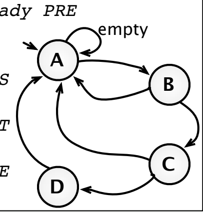

Figure 9. **Example transaction processing code for the FR-FCFS scheduling algorithm. The leftmost register in each line of code is the destination register.**

5 Implementation

This paper explores a scalar pipelined implementation of PARDIS as depicted in Figure 10. 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), and is issued to the DRAM subsystem at the next rising edge of the DRAM clock.

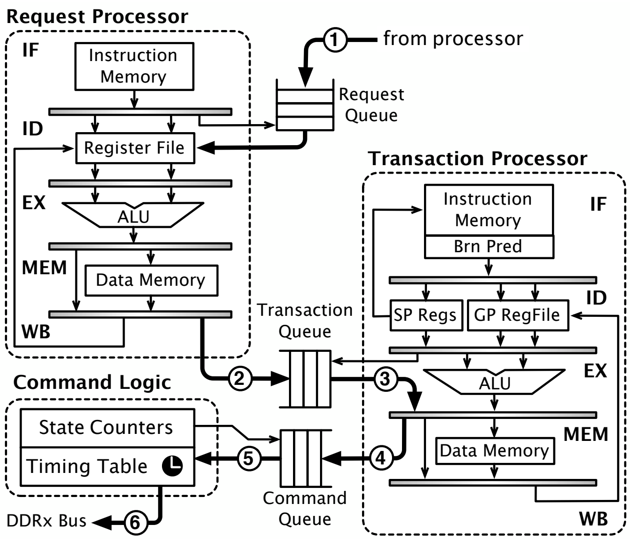

Figure 10. **Illustrative example of the proposed PARDIS implementation.**

5.1 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, 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.

5.2 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 11).

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 decode, a 16-bit ALU performs arithmetic and logic operations; in parallel, the transaction queue is accessed. Figure 12 shows the proposed architecture of the transaction queue comprising five components: 1) five 64-entry content-addressable 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 64×86 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 pro-

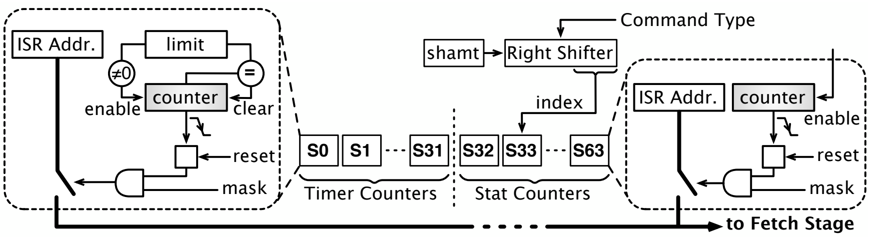

Figure 11. Interrupt counters in the proposed PARDIS implementation.

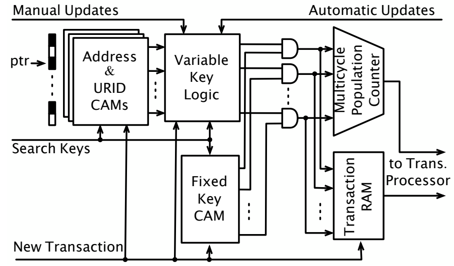

Figure 12. The proposed architecture of the transaction queue.

vide 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.
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.

5.3 Command Logic

The command logic (Figure 13) is implemented 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. 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.

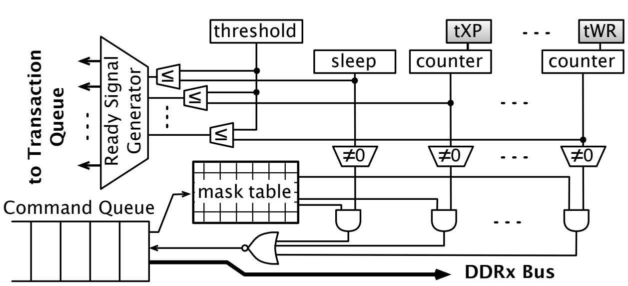

Figure 13. Illustrative example of the proposed command logic for PARDIS.

6 Experimental Setup

We evaluate the performance potential of PARDIS by comparing fixed-function hardware and PARDIS-based firmware implementations of FCFS [13], FR-FCFS [13], Par-BS [12], and TCMS [14] scheduling algorithms. We also implement in firmware a recent DRAM power management algorithm proposed by Hur and Lin [15], 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 [16] 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 [17]. We measure the area, frequency, and power dissipation of PARDIS by implementing the proposed system in Verilog HDL, and synthesizing the proposed hardware.

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 [19]	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

Table 1. Simulation parameters.

6.1 Architecture

We modify the SESC simulator [17] to model an eight-core system with a 4MB L2 cache and two on-chip memory controllers. Table 1 shows the simulation parameters. In the simulated configuration, memory channels are fully populated with DIMMs (typical of server systems [15]), which restricts the maximum channel data rate to 800MT/s for

DDR3-1066 [18, 19, 20]. 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[19]. 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 since policies are implemented in firmware.)

6.2 Applications

Evaluated parallel workloads represent a mix of 13 data-intensive applications from Phoenix [21], SPLASH-2 [22], SPEC OpenMP [23], NAS [24], and Nu-MineBench [25] suites. Table 2 summarizes the evaluated benchmarks and their input sets. All applications are simulated to completion.

Benchmarks	Suite	Input
Histogram	Phoenix	34,843,392 pixels (104MB)
String-Match	Phoenix	50MB non-encrypted file
Word-Count	Phoenix	10MB text 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 2. Applications and data sets.

6.3 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 [28] with FreePDK [29] at 45nm. The results are then scaled to 22nm (relevant parameters are shown in Table 3). Instruction and data memories are evaluated using CACTI 6.0 [30], while register files and CAMs are modeled through SPICE simulations with the FabMem toolset from FabScalar [31].

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

Table 3. Technology parameters [26, 27].

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

7.1 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 14. A fully synthesizable 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%).

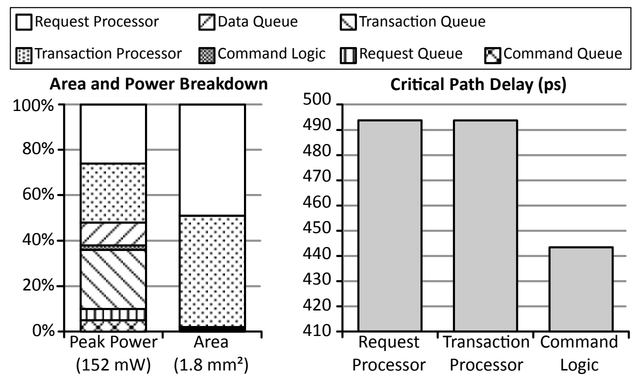

Figure 14. Delay, area, and peak power characteristics of the synthesized PARDIS implementation.

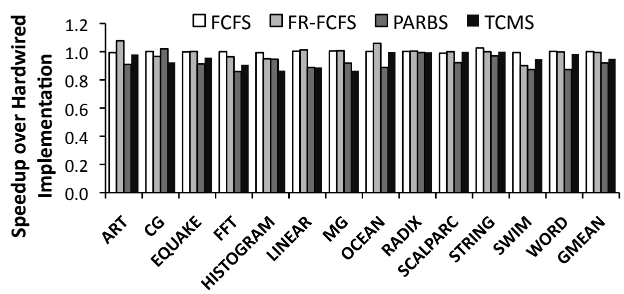

Figure 15. Performance of PARDIS-based and hardwired implementations for FCFS, FR-FCFS, PARBS, and TCMS scheduling algorithms.

7.2 Scheduling Policies

Figure 15 compares PARDIS-based firmware implementations of FCFS [13], FR-FCFS [13], Par-BS [12], and TCMS [14] 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.

7.3 Address Mapping

To evaluate the performance of different DRAM address mapping techniques on PARDIS, the permutation-based interleaving [11] technique was mapped onto PARDIS and compared to its fixed-function hardware implementation (Figure 16). The average performance of the two implementations differ by less than 1%; interestingly, PARDIS outperforms fixed-function hardware by a small margin on some applications. As explained in Section 7.2, 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 16). Other applications, such as ART and OCEAN, do not benefit from this phenomenon.

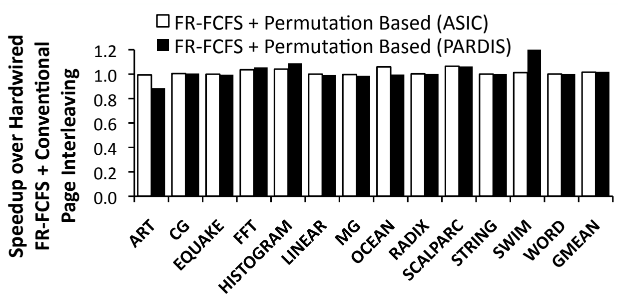

Figure 16. Performance of PARDIS-based and hardwired implementations of permutation based address mapping.

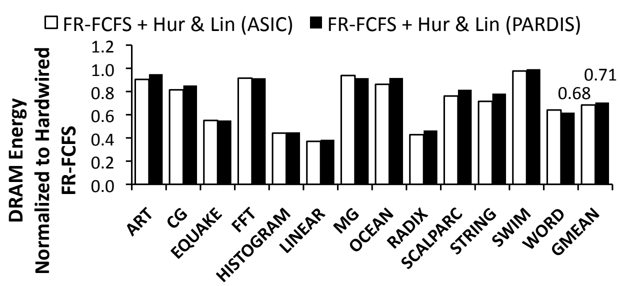

Figure 17. DRAM energy comparison between the PARDIS-based and hardwired implementations of the queue aware power management technique.

7.4 Power Management

DRAM power management with PARDIS was evaluated by implementing Hur and Lin's queue-aware power management technique [15] in firmware, and comparing the results to a fixed-function hardware implementation (Figure 17); 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

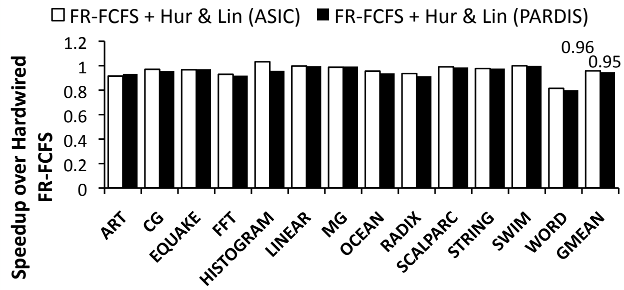

Figure 18. Performance of PARDIS-based and hardwired implementations for power management technique.

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 17 and 18).

7.5 Refresh

In order to evaluate DRAM refresh management on PARDIS, a conventional on-demand DDR3 refresh method [19] is considered as the baseline to which fixed-function hardware and PARDIS-based firmware implementations of the recently proposed Elastic Refresh algorithm [16] are compared (Figure 19). 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.

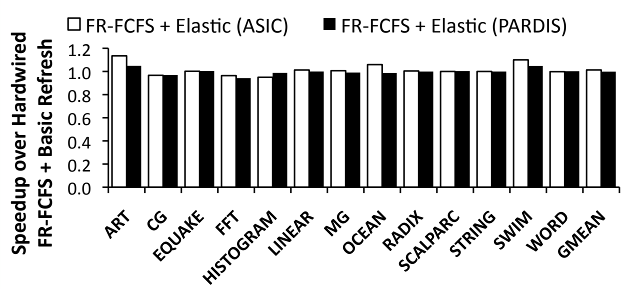

Figure 19. Performance of PARDIS-based and hardwired implementations of the elastic refresh scheduling algorithm.

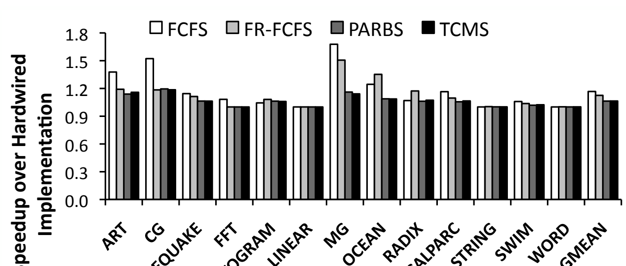

Figure 20. Speedup over hardwired permutation-based interleaving [11] using application-specific address mapping on PARDIS.

7.6 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. [1] As shown in Figure 20, application-specific DRAM indexing improves performance by 17%, 14%, 7%, and 6% over permutation-based interleaving [11] for FCFS, FR-FCFS, Par-BS and TCMS, respectively; corresponding DRAM energy savings are 22%, 14%, 9%, and 9% (Figure 21).

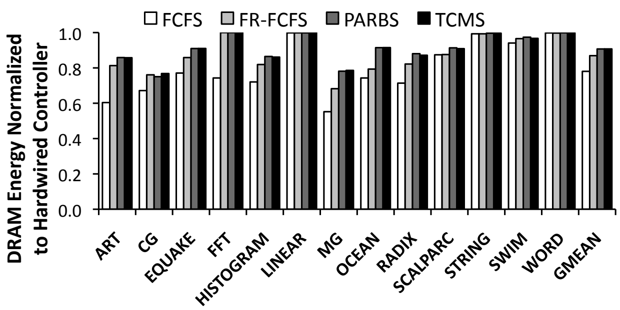

Figure 21. **DRAM energy savings over hardwired permutation-based interleaving [11] using application-specific address mapping on PARDIS.**

8 Related Work

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

8.1 DDRx Controller Optimizations

Numerous DDRx controller optimizations for improving performance, energy, and QoS have been published in the literature [12, 13, 14, 15, 16, 32, 33, 34, 35, 36, 37, 41]. 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, wearout, 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.

[1] 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 [5] are left for future work.

8.2 Programmable Cache and Directory Controllers

Programmability is a well-known concept that has been broadly applied to memory systems. FLASH [4] is a multiprocessor platform that introduces a general-purpose processor, called MAGIC, for executing directory protocols. Typhoon [5] is a programmable architecture that supports Tempest—a message passing protocol. Alewife [8] allows performance tuning through configuration of the cache coherence protocol. Smart Memories [9] 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.

8.3 Intelligent DRAM Controllers

Intelligent memory controllers have been proposed to provide a degree of configurability to the memory system. Impulse [6] 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 [1, 2], or allow choosing among pre-defined QoS-aware scheduling algorithms for a DDRx memory controller [38]. Recently proposed RL-based memory controllers [39, 40] 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.

9 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.

10 Acknowledgments

The authors would like to thank Arun Jagatheesan and anonymous reviewers for useful feedback. This work was supported in part by a Samsung contract.

References

[1] J. Martin, C. Bernard, F. Clermidy, and Y. Durand. A micro-programmable memory controller for high-performance dataflow applications. In *Proc. European Solid-State Circuits Conference*, 2009.

[2] G. Kornaros, I. Papaefstathiou, A. Nikologiannis, and N. Zervos. A fully programmable memory management system optimizing queue handling at multi gigabit rates. In *Proc. Design Automation Conference*, 2003.

[3] Micron Technology, Inc., *TN-29-14: Increasing NAND Flash Performance Functionality*, 2009. `http://www.micron.com/get-document/?documentId=140`.

[4] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proc. International Symposium on Computer Architecture*, 1994.

[5] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and typhoon: user-level shared memory. In *Proc. International Symposium on Computer Architecture*, 1994.

[6] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *In Proc. International Symposium on High-Performance Computer Architecture*, 1999.

[7] M. Browne, G. Aybay, A. Nowatzyk, and M. Dubois. Design verification of the S3.mp cache coherent shared-memory system. *IEEE Transactions on Computers*, 1998.

[8] A. Agarwal, R. Bianchini, D. Chaiken, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine: architecture and performance. In *Proc. International Symposium on Computer Architecture*, 1995.

[9] A. Firoozshahian, A. Solomatnikov, O. Shacham, Z. Asgar, S. Richardson, C. Kozyrakis, and M. Horowitz. A memory system design framework: creating smart memories. In *Proc. International Symposium on Computer Architecture*, 2009.

[10] B.L. Jacob, S.W. Ng, D.T. Wang, and D.T. Wang. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann, 2008.

[11] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proc. International Symposium on Microarchitecture*, 2000.

[12] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proc. International Symposium on Computer Architecture*, 2008.

[13] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens Memory access scheduling. In *Proc. International Aymposium on Computer Architecture*, 2000.

[14] Y. Kim, M. Papamichael, O. Mutlu, and M.H. Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proc. International Symposium on Microarchitecture*, 2010.

[15] I. Hur and C. Lin. A comprehensive approach to DRAM power management. In *Proc. International Symposium on High-Performance Computer Architecture*, 2008.

[16] J. Stuecheli, D. Kaseridis, H.C. Hunter, and L.K. John. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *Proc. International Symposium on Microarchitecture*, 2010.

[17] J. Renau et al. SESC simulator, Jan. 2005. `http://sesc.sourceforge.net`.

[18] Micron Technology, Inc., *TN-41-08: Design Guide for Two DDR3-1066 UDIMM Systems Introduction*, 2009. `http://www.micron.com//document\_download/?documentId=4297`.

[19] Micron Technology, Inc., *8Gb DDR3 SDRAM*, 2009. `http://www.micron.com//get-document/?documentId=416`.

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

[21] R.M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proc. International Symposium on Workload Characterization*, 2009.

[22] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. Interntional Symposium on Computer Architecture*, 1995.

[23] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 1998.

[24] D.H. Bailey et al. NAS parallel benchmarks. Technical report, NASA Ames Research Center, 1994. Tech. Rep. RNR-94-007.

[25] R. Narayanan et al. Minebench: A benchmark suite for data mining workloads. In *International Symposium on Workload Characterization*, 2006.

[26] ITRS. *International Technology Roadmap for Semiconductors: 2010 Update*. `http://www.itrs.net/links/2010itrs/home2010.htm`.

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

[28] Encounter RTL compiler. `http://www.cadence.com/products/ld/rtl_compiler/`.

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

[30] S. Wilton and N. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 1996.

[31] N.K. Choudhary, S.V. Wadhavkar, T.A. Shah, H. Mayukh, J. Gandhi, B.H. Dwiel, S. Navada, H.H. Najaf-abadi, and E. Rotenberg. Fabscalar: composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *Proceeding of the International Symposium on Computer Architecture*, 2011.

[32] Y. Kim, D. Han, O. Mutlu, and M.H. Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proc. International Symposium on High-Performance Computer Architecture*, 2010.

[33] B. Diniz, D.O.G. Neto, W. Meira, and R. Bianchini. Limiting the power consumption of main memory. In *Proceeedings of International Symposium on Computer Architecure*, 2007.

[34] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. In *Proc. International Symposium on Microarchitecture*, 2008.

[35] C. Isen and L. John. Eskimo: energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem. In *Proc. International Symposium on Microarchitecture*, 2009.

[36] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. In *Proc. Architectural Support for Programming Languages and Operating Systems*, 2010.

[37] S. Liu, K. Pattabiraman, T. Moscibroda, and B.G. Zorn. Flikker: saving DRAM refresh-power through critical data partitioning. In *Proc. Architectural Support for Programming Languages and Operating Systems*, 2011.

[38] K.-B. Lee, T.-C. Lin, and C.-W. Jen. An efficient quality-aware memory controller for multimedia platform SOC. In *IEEE Transactions on Circuits and Systems for Video Technology*, 2005.

[39] E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self-optimizing memory controllers: a reinforcement learning approach. In *Proc. International Symposium on Computer Architecture*, 2008.

[40] J. Mukundan and J.F. Martinez. Morse: multi-objective reconfigurable self-optimizing memory scheduler. In *Proc. International Symposium on High-Performance Compouter Architecture*, 2012.

[41] J. Stuecheli, D. Kaseridis, D. Daly, H.C. Hunter, and L.K. John. The virtual write queue: coordinating DRAM and last-level cache policies. In *Proc. Interntional Symposium on Computer Architecture*, 2010.