DESIGNING EFFICIENT MEMORY SCHEDULERS FOR FUTURE SYSTEMS

by

Niladrish Chatterjee

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

December 2013

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Niladrish Chatterjee**

has been approved by the following supervisory committee members:

Rajeev Balasubramonian	, Chair	**30-Sept-2013** Date Approved
Alan Davis	, Member	**30-Sept-2013** Date Approved
Erik Brunvand	, Member	**30-Sept-2013** Date Approved
Nuwan Jayasena	, Member	**30-Sept-2013** Date Approved
Naveen Muralimanohar	, Member	**20-Oct-2013** Date Approved

and by **Alan Davis** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

The internet-based information infrastructure that has powered the growth of modern personal/mobile computing is composed of powerful, warehouse-scale computers or datacenters. These heavily subscribed datacenters perform data-processing jobs under intense quality of service guarantees. Further, high-performance compute platforms are being used to model and analyze increasingly complex scientific problems and natural phenomena. To ensure that the high-performance needs of these machines are met, it is necessary to increase the efficiency of the memory system that supplies data to the processing cores. Many of the microarchitectural innovations that were designed to scale the memory wall (e.g., out-of-order instruction execution, on-chip caches) are being rendered less effective due to several emerging trends (e.g., increased emphasis on energy consumption, limited access locality). This motivates the optimization of the main memory system itself. The key to an efficient main memory system is the memory controller. In particular, the scheduling algorithm in the memory controller greatly influences its performance. This dissertation explores this hypothesis in several contexts. It develops tools to better understand memory scheduling and develops scheduling innovations for CPUs and GPUs. We propose novel memory scheduling techniques that are strongly aware of the access patterns of the clients as well as the microarchitecture of the memory device. Based on these, we present (i) a Dynamic Random Access Memory (DRAM) chip microarchitecture optimized for reducing write-induced slowdown, (ii) a memory scheduling algorithm that exploits these features, (iii) several memory scheduling algorithms to reduce the memory-related stall experienced by irregular General Purpose Graphics Processing Unit (GPGPU) applications, and (iv) the Utah Simulated Memory Module (USIMM), a detailed, validated simulator for DRAM main memory that we use for analyzing and proposing scheduler algorithms.

To My Parents, Wife, and the Taxpayers Who Subsidized My University Education

CONTENTS

LIST OF TABLES

ACKNOWLEDGEMENTS

A PhD is the culmination of frantic efforts over a long period of time. At times during these five years, things looked great; ideas were forming in dreams and in the shower, experiments were producing good results, bugs were fixing themselves, and papers were waltzing into conferences. But interspersed with these rosy times, there were times when, to quote Al Davis, the PhD experience felt like "setting your hair on fire for five years." Times like these are what make up the real PhD experience. I am extremely thankful for having people around me who reminded me, during these times, that there is a world outside my cubicle in the Utah Arch lab. My wife Anusua and my parents managed to find the right balance of sweet cajoling and brisk rebuke to set me straight whenever I tried to emulate the proverbial headless chicken during these five years. There is simply no way in which I could have made progress without their willingness to bear with my erratic behavior with unshakeable good demeanor.

Having a good PhD advisor might be the single-biggest factor in determining the quality and success of a graduate career. I had the best. Rajeev's technical acumen is known very well in academic circles. But more importantly, his quick sense of humor, calmness, balance in life, and innate humility have had the biggest effects on me, and I hope to emulate these aspects of his personality throughout my career as well.

My committee members have also had a big impact on the quality and scope of my work. I am thankful for the advice I received from Al and Erik that comes from their years of experience in dealing with complex problems and problematic grad students. Nuwan gave me the chance to intern for AMD Research, where I learned a great deal from him and made friends I would not have made otherwise. I would also like to take this opportunity to thank Mike O'Connor and Gabriel Loh at AMD, whose technical insight helped me to formulate a lot of the ideas in this thesis. Naveen was a senior PhD student in the Utah Arch lab when I joined, a mentor to me during my internship at HP Labs under Norm Jouppi, and finally became my committee member. He has provided invaluable advice on my research, on my course selection, on my job search, and on restaurants in the Bay Area.

My labmates from the Utah Arch labaratory are an “interesting” bunch, to put it mildly. Kshitij, Manu, Ani, Dave, Seth, Manju, Ali, Danny, and Aasheesh have been collaborators at work and partners in crime—catch me sometime to hear about some of the literally insane experiences we have shared together. My friends in Salt Lake City, Arijit, Protonu, Saurav, Meghana, Piyush, and Avishek increased my quality of life immensely and made it possible to enjoy the incredible landscape of this scenic state. My good friend Syamantak Das was the first to motivate me to get into a graduate program and a great pillar of support over instant-messenger.

Finally I would like to thank Karen and Ann at the School of Computing for their advice and help in keeping me in line with the plethora of official rules that dictate the lives of all international students.

CHAPTER 1

INTRODUCTION

1.1 Emerging Trends

Current trends indicate that the computing industry will bifurcate primarily along the following lines. The first kind of computers will be the relatively simple mobile devices that act as information consumption terminals. The other kind will be deployed in large server farms, also called warehouse-scale computers. The mobile devices are severely constrained by their form-factors and battery-life, and are thus reliant on a web-based information distribution infrastructure for their operation. The intensive computing required to provide these mobile devices with data will be carried out in powerful computers in large datacenters. On the other hand, scientific and high-performance computing tasks will be carried out on increasingly powerful machines which support a high degree of parallelism. One of the major challenges in architecting such powerful systems is the efficient delivery of data to the compute cores. The main memory system has always been a system bottleneck. In particular, the memory latency wall has been a well-recognized issue. Many important innovations in processor microarchitecture, e.g., out-of-order speculative processing, symmetric multithreading, and branch-prediction, have tried to leverage instruction-level-parallelism to mitigate the latency wall. On the other hand, multi-core processors and graphics processing units have looked to leverage thread-level parallelism to hide memory-induced delays. However, several emerging trends are threatening to reduce the efficiency of these established techniques, and, consequently, motivating the optimization of the main memory system itself.

1.1.1 Energy Constraints Forcing Simpler Cores

A large datacenter housing several thousands of servers can consume up to 30 MW of power [1] and the combined energy consumption of datacenters accounts for about

2% of the total energy production in the United States [2]. Processors are the highest consumers of energy in such systems. The large reorder buffers and other associated mechanisms that enable out-of-order processing have often been singled out for high power consumption. This has led to the use of simpler processor cores in servers, such as Intel's Atom [3] and more commonly, low-power Advanced RISC Machines (ARM) cores [4]. Many of these simpler cores offer limited to no support for out-of-order processing. Since the processors cannot effectively hide the Dynamic Random Accem Memory (DRAM) latency, performance-optimized main memory becomes a necessity.

1.1.2 Performance Demands of the Future

The performance demand from servers is always on the rise. The most exciting commercial applications of today are in the field of "big data." This model of computation typically requires the mining of useful insight and information from many thousands of petabytes of data. Thousands of threads mining this data can be run concurrently on the hardware thanks to the increasing on-chip core counts. Also, general purpose GPUs (GPGPUs) are being employed increasingly in high-performance-computing platforms to model complex natural phenomena and also in commercial compute systems as data-parallel accelerators. The high degree of thread-parallelism exerts tremendous pressure on the memory bandwidth. DRAM vendors have responded to this challenge by increasing the DRAM pin frequency. In spite of this, the aggregate available bandwidth is limited by the pin count on the processor socket. Compared to the 16X growth that is expected to take place in transistor count (and probably on-chip core count) over a period of 8 years, pin counts are estimated to grow by only 1.47X in the same period, according to the ITRS road-map [5]. With several cores competing for the restricted number of pins, it is more important than ever to increase the bandwidth utilization and reduce the high queuing delay encountered by memory requests. The bandwidth utilization is a function of the DRAM bank utilization which motivates intelligent memory controllers.

1.1.3 DRAM Core Speeds

The DRAM pin bandwidth (i.e., DRAM interface frequency) has increased significantly over different DRAM generations; the same cannot be said about the DRAM core latencies. Fig. 1.1 demonstrates how two of the most important timing parameters that determine DRAM latency have reduced only slightly over the years. As a consequence, while the latency for transmitting a set of bits over the DRAM interface to the processor has reduced (owing to the increasing data-rates shown on the x-axis of Fig. 1.1), the latency of accessing the DRAM core has not scaled. In high-traffic scenarios, this increases contention for the DRAM banks (see Chapter 2), leading to higher overall latency.

1.1.4 Emerging Application Trends

Applications of the future demand high performance and higher reliability. Modern GPUs allow developers to express the parallelism in their applications through programming paradigms like CUDA [6] and OpenCL [7]. While GPUs are well-suited for handling regular, structured code, it is still a large challenge to efficiently support irregular applications [8]. The GPU core and memory architecture are organized with expectations of regular compute and access patterns and can lead to significant performance penalties for irregular applications.

On the other hand, datacenters running business-critical applications require reliable memory systems. This has led to the deployment of chipkill-correct memory systems [9] where the memory system is able to tolerate the failure of a complete DRAM chip. A direct consequence of the chipkill feature is an increase in the write-traffic to the memory system. Most DRAM systems are designed to primarily accelerate reads (as writes are not on the critical path), but inefficient handling of writes can cause large slowdowns.

1.2 Dissertation Overview

It is clear from the preceding discussion that the memory latency wall continues to be a major concern. In this dissertation, we look at optimizations to the memory system that can satisfy the performance demands of future workloads. To accomplish this, we focus on the memory controller, which constitutes the “smarts” of the main

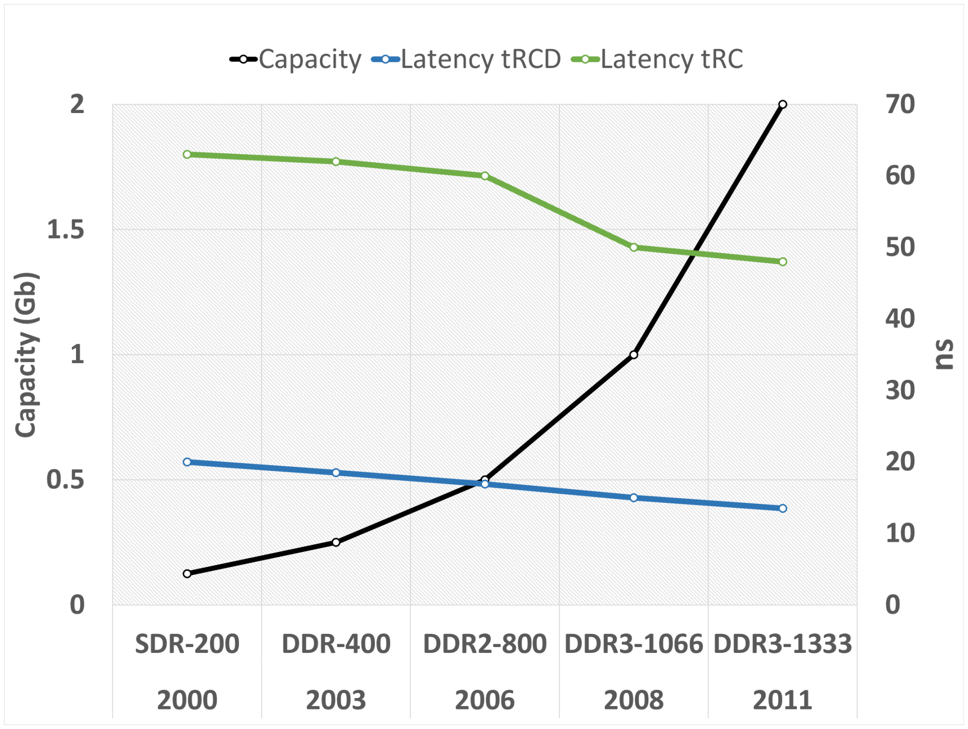


Figure 1.1. DRAM Trends

memory system and is also the most malleable part of the memory system where changes can be instituted with minimum cost impacts. The impact of memory scheduling algorithms on overall system throughput and power consumption have been demonstrated by previous studies [10], [11], [12], [13], [14], [15]. However, very few studies have looked at the impact of write scheduling on DRAM reads [16], [17]. We find that writes can be a significant bottleneck in current workloads and certainly in future systems that employ more stringent error checking and in systems that deploy nonvolatile memory chips. We design a memory architecture to mitigate this bottleneck. Similarly, memory scheduling techniques for GPU architectures have focused on improving the memory throughput with little consideration of the impact of DRAM latency on GPU performance. We show that in Single Instruction Multiple Thread (SIMT) cores, the memory system needs to be aware of the source of requests during the scheduling stage and invent mechanisms that provide high performance. Finally, we develop a detailed memory simulator that helps accelerate similar studies by the community at large. Our analysis with this simulator helps shed insight on memory scheduling bottlenecks.

1.2.1 Thesis Statement

Main memory performance is a key determinant of system throughput. The most malleable part of the memory system is the memory controller. The key to architecting efficient memory for the future lies in the design of intelligent memory scheduling algorithms that are aware of memory access patterns and the intricacies of DRAM chip microarchitecture as well as the architectural bottlenecks in the client cores. This thesis is aimed at developing scheduling strategies and tools to address these issues.

1.2.2 Write-Aware Main Memory

DRAM bandwidth is a precious system resource, and one of the factors that can prevent efficient utilization of the DRAM bandwidth is the draining of writes. Given that reads are on the critical path for CPU progress, reads are prioritized over DRAM writes by the memory scheduler, but writes have to be drained from the write queue buffers eventually and the write-drain process delays pending reads. In fact, a single channel in the main memory system offers almost no parallelism between reads and writes. This is because a single off-chip memory bus is shared by reads and writes, and the direction of the bus has to be explicitly turned around when switching from writes to reads. This is an expensive operation, and its cost is amortized by carrying out a burst of writes or reads every time the bus direction is switched. As a result, no reads can be processed while a memory channel is busy servicing writes even if the reads and writes are being serviced from different banks of the DRAM device. To alleviate this performance loss, we propose a novel mechanism to boost read-write parallelism and perform useful components of read operations even when the memory system is busy performing writes. If some of the banks are busy servicing writes, we start issuing reads to the other idle banks. The results of these reads are stored in a few registers near the memory chip's I/O pads. These results are quickly returned immediately following the bus turnaround. This reduces the queuing delay of the reads waiting for the write-queue drain to complete and also frees up banks faster for future reads. This process is referred to as a Staged Read because it decouples a single column-read operation into two stages, with the first step being performed in parallel with writes. This technique works well when there is bank imbalance in the write stream and there are pending reads on the banks that do not have many pending writes. To exploit this, we designed a write scheduling algorithm that artificially introduces bank imbalance and allows useful read operations to be performed during the write-drain. With a marginal chip area overhead (0.25%), we can gain a DRAM access latency improvement of 17% using staged-reads.

1.2.3 Warp-Aware Main Memory

Graphics Processing Units (GPUs) use the SIMT model of computation where a group of threads execute the same instruction on different data elements. A group of threads running in lockstep in such a setup is called a warp (or a wavefront)—with every thread in the warp executing the same instruction. A load instruction in such a SIMT system can generate many different memory requests and the warp becomes ready to run (referred to hereafter as runnable) only when all of the outstanding memory requests are returned to the compute unit. The compute units are simple and typically lack the ability to hide the latencies of pending memory requests through techniques commonly found in out-of-order, speculative, superscalar processors. To negotiate long memory access times, a GPU's compute unit uses thread level parallelism instead of instruction level parallelism as a CPU would. Thus, when a warp waits for its memory requests to return from the memory system, the thread scheduler in the GPU picks a different warp that is ready to run. The memory requests issued by a warp will typically encounter different memory latencies. In fact, modern memory controllers schedule incoming requests out-of-order to maximize memory system throughput, which can stall some requests from a warp for a long time, thereby hampering the progress of the warp. This introduces the problem of memory latency divergence where a warp is stalled until the last memory request from a vector load instruction is returned to the compute unit. Several studies have highlighted how memory divergence can be a significant performance bottleneck in GPUs [18], [19]. We observe that the effective DRAM latency for a warp is often lengthened at the main memory because one or more requests of that warp are returned with longer latencies than the rest. We propose DRAM scheduling strategies that attempt to reduce the intrawarp memory latency divergence by eliminating interwarp interference. We first propose schemes that reduce intrawarp latency divergence in a single controller through a DRAM bank-aware shortest-job-first policy (BASJF, Section 4.3.3). We then augment it to be implicitly multicontroller aware (BASJF-AB, Section 4.3.4) by introducing an age bias in the scheduling scheme. We then further optimize the scheduler to regain the lost bandwidth utilization by carefully orchestrating the scheduling of row-miss requests (MERB, Section 4.3.5.1). We then couple this scheduler with a write-drain mechanism that reduces write-induced stall times for warps (WAWD, Section 4.3.6). The combined techniques reduce the adverse effects of memory divergence, reduce intrawarp latency variation, and thus improve performance by 8.6% on average.

1.2.4 Utah Simulated Memory Module

The Utah Simulated Memory Module (USIMM) was developed as the simulation framework for the 3rd Journal of Instruction Level Parallelism' Computer Architecture Competition: the Memory Scheduling Championship [20]. The tool has the potential to accelerate memory system research by the community. The dissertation discusses the design of the tool, analyzes its accuracy, and identifies important areas of focus for memory system research.

CHAPTER 2

BACKGROUND

In this section, we briefly look at the architecture of a modern memory system, the general features of modern memory schedulers, and the DRAM access characteristics of a graphics processing unit. This helps provide a background for the subsequent discussions where we investigate memory scheduling techniques that are aware of the internal characteristics of memory devices as well as the different access patterns generated by the clients of the memory system.

2.1 Memory System Basics

2.1.1 DRAM System Organization

A typical, modern main memory system [21] employs JEDEC-style Dual Data Rate (DDR) SDRAM [22],[23]. Modern processors [24], [25] often integrate a plurality of memory controllers on the processor dies. Each memory controller is tasked with managing one or two (independent if two channels) off-chip main memory channels. Each channel is comprised of a 64-bit data bus and a 17-bit address and command bus. Multiple Dual Inline Memory Modules (DIMMs) are hosted on each channel. Each DIMM comprises multiple *ranks*, each rank being a collection of DRAM devices which work in unison to return data in response to a memory read request. A rank thus consists of the smallest number of chips that need to be activated to complete a read or write operation. Fig. 2.1 shows an example DIMM with 16 total DRAM chips forming two ranks. Ranks on the same channel share the data and command/address buses, but different ranks can work in parallel to service different requests. A schematic of the memory system that highlights its main constituent parts and subdivisions is shown in Fig. 2.1.

DRAM chips are often characterized by their output pin width. An *xN* DRAM chip has *N* output pins, and *N* bits of data go in/out of the chip on each clock-tick.

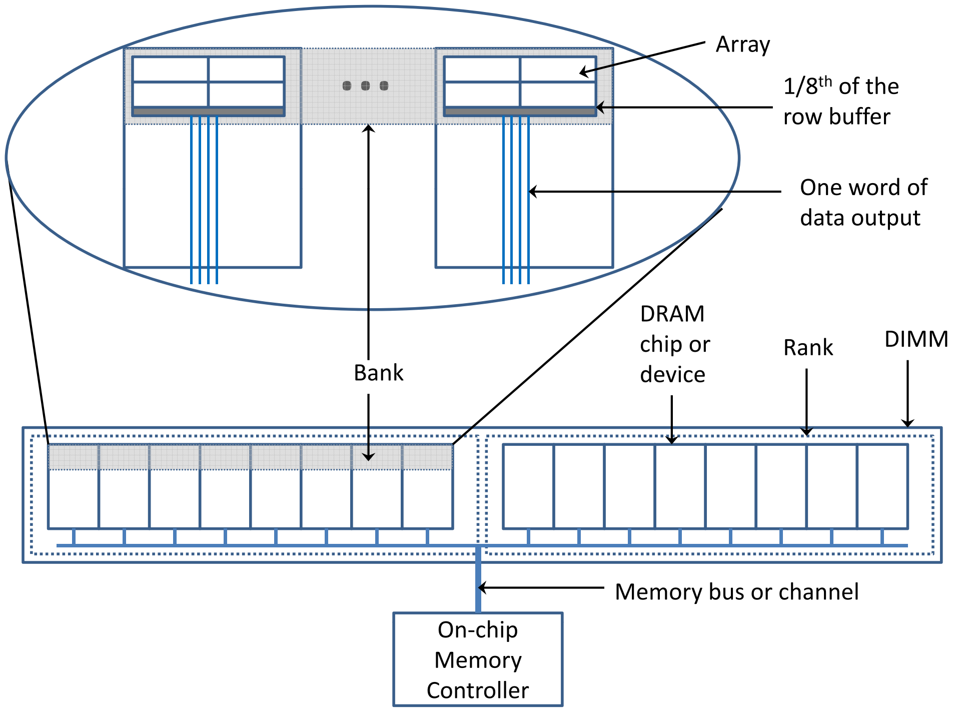


Figure 2.1. An example DDRx SDRAM architecture shown with one DIMM, two ranks, and eight x4 DRAM chips per rank.

In DDR chips, each pin transmits one bit at each edge of the clock signal. For a 64-bit data bus and x8 chips, a rank would require 8 DRAM chips (Fig. 2.1 only shows 8 x8 chips per rank to simplify the figure). Each DRAM chip is attached to a subset of the channel's data pins. When a rank is selected, all DRAM chips in the rank receive address and command signals from the memory controller on the corresponding shared buses. The rank selection is done by asserting chip-select signals, all chips on a rank being connected to the same chip-select signal.

Each rank is partitioned into multiple *banks*, typically numbering four to sixteen. Each bank can concurrently be processing a different memory request, although data transfers from/to the different banks have to be serialized over the shared data bus. Each bank is spread across the DRAM chips that constitute the rank. The same bank in each chip is involved in the transfer of a single cache-line request from the memory controller. Each cache-line is thus striped across the different DRAM chips on a rank and this allows the whole channel bandwidth to be utilized for the data transfer. In the example shown in Fig. 2.1, each DRAM chip contributes 1/8th of the data, i.e., 8 bytes for a 64 byte cache-line. With a data bus width of 64 bits and a cache-line size of 64 bytes, the data transfer requires 8 *bursts* on the channel.

Each DRAM bank can be logically thought of as comprising of a 2D array of 1T-1C DRAM cells. Physically, however, each array is split into several subarrays [26], [27] for managing the latency and current-draw. To access a set of bits, corresponding to a cache-line from the bank, the appropriate row of DRAM devices has to be first activated. This results in the data from the cells being read into a set of *sense-amplifiers*. These sense-amplifiers comprise the *row-buffer* of the DRAM bank. To service a cache-line request, a set of columns are then selected from the row-buffer and the bits are routed to the DRAM pins over a data bus that is shared by the different banks of the DRAM device. A row-buffer can retain the bits from the most recently accessed row, and the row is then considered "open." If subsequent accesses are to cache-lines in that open row, then the access is termed a *row-buffer hit*. If the requested data are not present in the bank's row buffer (a *row buffer miss*), the currently open row (if one exists) has to first be closed before opening the new row. Row-buffer hits consume less energy and take less time to complete, thus, memory architects often take special care to exploit such row-buffer locality.

As an example system, consider a main memory system of 4 GB capacity, organized on one channel serving a system that has a 64-byte cache-line size. If the system is comprised of 2Gb, x8 DRAM devices, then each rank will contain 8 DRAM devices, and there will be 2 such ranks on the channel. Each device has 8 banks, thus each bank is 256Mb in capacity and is split into 8 arrays. The cells in each bank will be organized in 65 536 rows and 1024 columns/row in each array. A row access thus brings down 1024 bits per array into the row-buffer of each chip—and the total row-buffer size across the 8 devices in the rank is thus 8 KB. This is often referred to as the page size of the DRAM system. Each cache-line access will require 64 bytes to be returned from these 8192 bytes.

2.1.2 Memory Controller

The memory-controller constitutes the "smarts" of the memory system and has the following main functions.

2.1.2.1 Address Translation

The physical address of a cache-line is converted into channel, rank, bank, row, and column addresses. Intelligent address translation can impact the performance and energy of the memory system. For example, parallelism is boosted by allowing consecutive cache-lines to be fetched from different channels, ranks, and/or banks. On the other hand, to exploit spatial locality and obtain higher row-hit rates, consecutive cache-lines may be placed in the same row of the same bank. Common address mapping techniques try to strike a balance between these approaches by mapping a group of consecutive cache-lines to the same row, and consecutive such groups are interleaved across channels, ranks, and banks. Several papers have investigated the impact of address mapping and proposed techniques for higher performance [28], [29], [30].

2.1.2.2 Memory Scheduling

This is arguably the most important function of the memory controller. The order in which memory requests are scheduled for service has a large impact on the performance and power consumption of the system. The memory controller has to balance several system-level considerations (e.g., thread priorities and read-write intensity) with the timing constraints associated with the DRAM devices while making scheduling decisions. We talk about this aspect in more detail in Section 2.1.3.

2.1.2.3 Error Management

The memory controller is also responsible for typically providing SECDED (*S*ingle *E*rror *C*orrect *D*ouble *E*rror *D*etect) error protection to data being read from the DRAM. For this, a 9th DRAM chip on a rank (consisting of eight x8 DRAM chips) is used to store parity information. The parity is read with the data and the memory controller does the necessary computation to detect and correct errors. In addition, in some critical scenarios, the DRAM systems are said to be chipkill-correct; that is, they can recover from the failure of a single DRAM chip on the rank in the worst case. The techniques for mitigating such errors are more involved [27] and require careful data placement and some computation from the memory controllers.

2.1.3 Main Memory Scheduling

The memory-controller's most important function is memory scheduling and a significant portion of the chip area is devoted towards the scheduling functionalities. Before we discuss the different memory scheduling techniques in some detail, it is beneficial to look at the basic DRAM access protocol (i.e. the sequence of DRAM commands that need to be issued to retrieve or write data to the DRAM devices.)

2.1.3.1 Memory Access Protocol

The ranks of memory devices are connected to the on-chip memory-controller via a 64-bit data bus and a narrower bus that transports commands and addresses to the ranks from the memory-controller. To read data from the DRAM system, the controller issues a column-read (*COL_RD*) command along with the appropriate bank and column addresses on the command/address bus. After a delay, the DRAM responds with the data on the data bus. If the data are not present in the open row-buffer of the bank, the controller needs to issue a precharge(*PRE*) command to set the bank's bitlines to an intermediate voltage value. This prepares the bank to receive an activate (*ACT*) command that brings a new row into the row-buffer. Writes are performed in the same way as reads, with the *COL_WR* command preparing the row-buffer to accept data from the data bus, which is then overdriven into the DRAM arrays. Each of these commands can be issued only after certain timing constraints are met, and the memory-controller is responsible for meeting these constraints before issuing each command [21]. For example, in response to the ACT command, the data are sensed and stored in the row-buffer and thereafter it is restored back in the DRAM arrays (DRAM-cell reads are destructive in nature). A COL_RD command can be issued only at the end of the sensing period, and thus the minimum gap between an ACT command and a COL_RD command to the same bank is the row-to-column-delay, *tRCD*. Similarly, a PRE command can be issued only after the completion of the restoration of the row to the arrays, and the minimum gap between the ACT and PRE command is given by the *tRAS* timing constraint. In addition, if there are multiple COL_RD commands to the same row that can effectively hide the tRAS delay, care has to be taken that the last COL_RD is separated from the subsequent PRE command by at least the read-to-precharge time or *tRTP* timing constraint. These three are just a subset of the timing constraints that the memory controller needs to be aware of. A command can be issued when multiple of these, possibly overlapping, timing constraints are met. A full discussion of these constraints is beyond the scope of this thesis, but the DRAM datasheets [31], [32] from manufacturers provide complete descriptions of the timing restrictions. In Chapter 5, we provide a more detailed account of the timing constraints and how they are modeled in our simulator.

2.1.3.2 Transaction Scheduling

Broadly, the scheduling task of the memory-controller can be split into transaction scheduling and command scheduling. The transaction scheduler picks a pending read or write command from the transaction queues, splits it into a series of DRAM commands (i.e. PRE + ACT + COL_RD for a row-miss and COL_RD for a row-hit), and enqueues the commands in the appropriate bank-level command queues. The transaction scheduler can pick requests out-of-order. Many studies have shown the importance of memory scheduling in determining the performance and power of the overall system [10], [12], [14], [33], [34]. Modern memory controllers process the incoming memory read requests out-of-order to extract high performance from the DRAM system. The most common optimization is the prioritization of row-hit requests through the *First-Ready First-Come-First-Served* (FR-FCFS) scheduling policy. Under this policy, the memory controller first schedules all row-hit requests (i.e., the requests that require only a COL_RD command to complete) before servicing row-miss requests. This yields lower DRAM latencies, higher data-bandwidth utilization, and lower power dissipation—leading to the popularity of this scheduling policy. This scheduling policy can be combined with either an open-row or closed-row page-management policy. In the open-row policy, the last accessed row is kept open in the row-buffer even when the DRAM request queue is empty in anticipation of row-hit requests that might arrive in the near future. This works well in applications with high spatial locality. The closed-row policy, however, precharges the bank as soon as the last request that hits in the row-buffer has been serviced. This removes the cost of a precharge from the critical path of a row-miss request and has been proposed for systems where many unrelated threads constantly conflict at the memory controller. Over the past few years, interest in memory scheduler design has seen many different scheduling algorithms being proposed. A significant majority of these deal with obtaining fairness and high performance when different threads in the system have different memory access characteristics [10], [11], [15], while some others have focused on allowing different memory controllers to coordinate their scheduling decisions for higher throughput [13]. Section 2.3 documents the major different scheduling strategies that have been used in processors or proposed in literature.

2.1.3.3 Write Scheduling

Besides scheduling read requests, the memory controller also has to handle incoming write requests to the DRAM. Most modern CPUs employ a write-back policy in their *Last Level Caches* (LLC). Consequently, writes to DRAM are the result of the eviction of dirty cache-lines from the LLC, and as such, they are not on the critical path for program execution. The writes are typically buffered in a write queue and are serviced when there are not performance-critical DRAM reads to service or when the write-queue nears full occupancy. When writes are being done to a bank, a different bank may service a read. However, every switch from a write to a read on the data bus of a chip requires a bus-turnaround penalty (tWTR) which reduces the bus utilization efficiency. To amortize the cost of this turnaround, writes are drained in batches. Writes are buffered until the write-queue reaches a high water mark, and the writes are drained till the queue occupancy is lowered to a low water mark. The bus is then turned around and reads are serviced [16], [35].

2.1.3.4 Command Scheduler

The part of the memory-controller that deals with issuing the DRAM commands (e.g., ACT, PRE, COL_RD, COL_WR) does so by ensuring the different timing constraints are met. The command scheduler scans the bank-level command-queues and picks a command that can be sent out on the address/command channel that cycle. The command scheduler typically will not reorder requests in a queue, but it interleaves requests from different ranks and banks to ensure high parallelism. The command scheduler is also responsible for scheduling periodic refresh commands. The refresh commands are needed for maintaining the data in the volatile memory cells.

The device specifies the maximum time that a cell can retain data (typically 64ms) and as a result, all the cells need to be refreshed before this time elapses.

2.2 Graphics Processing Unit Basics

Graphics Processing Units (GPUs) have emerged as an efficient alternative to traditional scalar processors for a large class of data parallel workloads. High-level programming models such as NVIDIA's CUDA [6] and OpenCL [7] allow the programmer to define the behavior of a single scalar thread, which is then replicated to run many threads on Single Instruction Multiple Data (SIMD) execution units (also called compute units). A group of threads running in lockstep in such a setup is called a warp (or a wavefront), with every thread in the warp executing the same instruction. A load instruction in such a SIMT (Single Instruction Multiple Thread) system can generate many different memory requests, and the warp becomes ready to run (referred to hereafter as runnable) only when all of the outstanding memory requests are returned to the compute unit. The compute units are simple and typically lack the ability to hide the latencies of pending memory requests through techniques commonly found in out-of-order, speculative, and/or superscalar processors. To negotiate long memory access times, a GPU's compute unit uses thread level parallelism instead of instruction level parallelism as a CPU would. Thus, when a warp waits for its memory requests to return from the memory system, the thread scheduler in the GPU picks a different warp which is ready to run.

The DRAM system in GPUs is designed for very high bandwidth. The DRAM channel is run at a frequency of 3GHz, the DRAM chips (GDDR5) are heavily banked and allow multiple memory requests to proceed in parallel. To maximize throughput, the memory scheduler aggressively reorders requests to achieve very high row-hit rates. With traditional graphics workloads displaying significant spatial locality, such a technique is particularly well-suited for high-performance. However, many applications with irregular access patterns are now being re-architected to take advantage of the massive parallelism in GPUs, and their memory access behavior is not always suited for the high-throughput scheduling policies employed in GPUs. For example, in the recent past, a number of algorithms that use graphs as the primary data structure have been ported to OpenCL/CUDA [36], [37]. In addition, server workloads such as memcached [8] have also been implemented for execution on GPUs. These workloads often demonstrate lower locality than conventional GPU compute workloads and as a result are not handled very efficiently by the memory system.

2.3 Scheduling Background

In this section, we briefly review the state-of-the-art in memory scheduling.

2.3.1 First Read First-Come-First-Served (FR-FCFS)

Proposed by Rixner *et al.* [12], this is the most popular memory scheduling algorithm that has been explored in detail in academia and also implemented in almost all commercial memory schedulers today. The basic idea of this scheduler is to allow requests that require less time to be serviced, by virtue of being a row-hit, to preempt older row-miss requests. Evidently, this has higher performance and better energy characteristics than a first-come first-served (FCFS) policy.

2.3.2 Stall-Time Fair Memory-Scheduling (STFM)

Proposed by Mutlu *et al.*, STFM [10] introduces the concept of a *fair* memory system, one in which the memory-related slowdown experienced by each thread due to interference from other threads is minimized, without hurting the overall performance. The scheduler determines the memory stall-time for a thread when it runs alone (*T_alone*) and when it runs in conjunction with other threads (*T_shared*). The scheduler calculates the *memory_slowdown* of every thread. When the ratio of the maximum and minimum slowdown of threads in the system breaches a threshold, the scheduler prioritizes the threads with high slowdowns. Otherwise it uses FR-FCFS for regular scheduling.

2.3.3 Parallelism-Aware Batch Scheduling (PARBS)

Proposed by Moscibroda *et al.* [34], this scheme tries to maintain the fairness and quality-of-service notions introduced in STFM and, in addition, aims at improving the system throughput. The scheduler first forms batches of requests by grouping consecutive outstanding requests in the memory request buffers and services all requests in a batch before moving over to the next batch. By grouping requests into batches, the scheme avoids starvation of threads at a very fine granularity and ensures steady and fair progress across all threads. Within a batch, row-hits are prioritized over row-misses and threads with few requests or those that display high-bank-level parallelism are prioritized over others to minimize the service time of a batch.

2.3.4 Adaptive per-Thread Least-Attained-Service Memory Scheduling (ATLAS)

Proposed by Kim *et al.* [13], ATLAS is a scheme that allows multiple memory-controllers to coordinate their scheduling decisions to improve throughput. Execution time is split into long epochs. During each epoch, the memory-controllers keep track of the level of service received by each thread from the memory system. At the beginning of the next epoch, this information is accumulated at a central coordinator, which increases the priorities of the threads that received the least service in the previous epoch. This information is propagated to the memory-controllers and thereafter, the selected threads are prioritized.

2.3.5 Thread-Cluster Memory Scheduling (TCM)

Proposed by Kim *et al.* [14], TCM argues that techniques such as STFM, PAR-BS, and ATLAS are unable to provide adequate fairness and high throughput because they use the same policy for all threads. In contrast, TCM uses the memory behavior of the thread to decide its priority. First, it prioritizes requests from non-memory-intensive threads over memory-intensive ones during memory scheduling. After making the observation that unfairness in memory scheduling techniques stems from interference among memory-intensive threads, TCM periodically shuffles the priority order among such threads to increase fairness. However, not all threads get to enjoy all priority levels; instead, threads with higher bank-level parallelism are prioritized over streaming threads that have high row-buffer locality.

2.3.6 Scheduling with Processor-Side Load Criticality Information

In this scheme proposed by Ghose *et al.* [38], a load that has a large number of consumer instructions and/or has a history of reaching the head of the reorder-buffer (ROB) long before the data for the load arrives at the processor are deemed high priority. This information is propagated to the memory controller to aid the memory-controller in prioritizing such critical loads. The same study was performed by Prieto *et al.* [39] where the authors reported maximum improvement for a scheme that uses the position of the load in the ROB as a metric for criticality.

2.3.7 Staged Memory Scheduling (SMS)

As a consequence of the integration of GPUs and CPUs on a chip, memory controllers in modern systems have to manage the memory traffic from both the CPU and GPU. CPUs are typically latency-sensitive and GPUs are bandwidth limited. To enable the memory controller to balance the different needs of these clients, the SMS scheme was proposed by Ausavarungnirun *et al.* [15]. SMS uses the same principles as PAR-BS and TCM. First batches of row-hit requests are formed from each client. A batch scheduler prioritizes batches of requests based on the shortest-job-first policy. Thus, requests from the CPU are prioritized by default while those from the GPU are generally deprioritized.

CHAPTER 3

A DRAM SCHEDULER OPTIMIZED FOR WRITES

3.1 Impact of DRAM Writes on Reads

Main memory latencies have always been a major performance bottleneck for high-end systems. This bottleneck is expected to grow in the future as more cores on a chip must be fed with data. Already, many studies [10], [40] have shown the large contribution of queuing delays to overall memory latency. A number of studies have focused on memory scheduling and have tried to optimize throughput and fairness [10], [14], [34], [40], [41]. However, only a few optimizations have targeted writes; for example, the Eager Writeback optimization [17] tries to scatter writes so that write activity does not coincide with read activity, and the Virtual Write Queue optimization [16] combines memory scheduling and cache replacement policies to create a long burst of writes with high row buffer hit rates.

Generally, read operations are given higher priority than writes. When the memory system is servicing reads, the DIMMs drive the off-chip data bus and data are propagated from the DIMMs to the processor. Since writes are not on the critical path for program execution, they are buffered at the processor's memory controller. When the write buffer is nearly full (reaches a high water mark), writes have to be drained. The data bus is turned around so that the processor is now the data bus driver and data are propagated from the processor to the DIMMs. This bus turnaround delay (tWTR) has been of the order of 7.5 ns for multiple DDR generations [16], [21], [42]. Frequent bus turnarounds add turnaround latency and cause bus underutilization, which eventually impacts queuing delay. Therefore, to amortize the cost of bus turnaround, a number of writes are drained in a single batch until a low water mark is reached. During this time, reads have no option but to wait at the memory controller; the unidirectional nature of the bus prevents reads from opportunistically reading data out of idle banks. Thus, modern main memory systems offer nearly zero read-write parallelism within a single channel.

In this chapter, we look at an optimization that allows reads to perform opportunistic prefetches while writes are being serviced. This is not a form of speculation; the read operation is simply being decoupled into two stages and the stage that does not require the data bus is being performed in tandem with writes. We refer to this optimization as a *Staged Read*. The two stages are coupled via registers near the memory chip's I/O pads that store the prefetched cache line. This not only minimizes the latency for the more critical second stage (the second stage does not incur delay for memory chip global wire traversal) but is also less disruptive to memory chip design. Prior work [43] has identified the I/O pad area as being most amenable to change, and that area already accommodates some registers that help with scheduling.

With the proposed optimization, while writes are being serviced at a few banks, other banks can perform the first stage of read operations. As many prefetches can be performed as the prefetch registers provided at the I/O pads. After the bus is turned around to service reads, these prefetched results are quickly returned in the subsequent cycles without any idling. The Staged Read optimization is most effective when only a few banks are busy performing writes. We therefore modify the write scheduling algorithm to force bank imbalance and create opportunities for Staged Reads. This ensures that the memory system is doing useful read work even when it is busy handling writes.

Such read-write parallelism becomes even more important in future write-constrained systems when (i) writes are more frequent in chipkill systems [27], [44], (ii) writes take longer (because of new NVM cells [45], [46]), and (iii) turnaround delays are more significant [16]. Our results show an average improvement of 7% in throughput for our baseline modern system (along with an average DRAM access latency reduction of 17%) and this improvement can grow to 12% in future systems. Applications that are write-intensive (about half of the simulated benchmarks suite) show an 11% improvement in throughput with our innovation.

3.2 Background and Motivation

3.2.1 Main Memory Background

The main memory system is composed of multiple channels (buses), each having one or more DIMMs. For most of this study, we will assume that the DIMMs contain multiple DRAM chips, although the proposed design will apply for other memory technologies as well. When servicing a cache line request, a number of DRAM chips in a *rank* work in unison. Each rank is itself partitioned into multiple *banks*, each capable of servicing requests in parallel. Ranks and banks enable memory-level parallelism, although each data transfer is eventually serialized on the memory bus. The most recently accessed row of a bank can be retained in a row buffer, which is simply a row of sense-amps associated with each array. The row is then considered "open." If subsequent accesses deal with cache lines in an open row (a row buffer hit), they can be serviced sooner and more efficiently.

A memory chip is organized into many banks; each bank is organized into many arrays. The I/O pads for a chip are placed centrally on a memory chip [43]. From here, requests and data are propagated via tree-like interconnects to individual arrays involved in an access. To maximize density, the arrays have a very regular layout and are sized to be large. When a read request is issued, the bitlines for the corresponding row must be first PRECHARGED (if they have not already been previously precharged). An ACTIVATE command is then issued to read the contents of a row into the row buffer. There is significant overfetch in this stage: to service a single 64 byte cache line request, about 8 KB of data are read into a row buffer. It is prohibitively expensive to ship this overfetched data on global wires, so the row buffer is associated with the arrays themselves. Finally, a column-select or CAS command is issued that selects a particular cache line from the row buffer and communicates it via global wires to the I/O pads. In the subsequent cycles, the cache line is transmitted to the processor over the off-chip memory bus. Each of these three major components (PRECHARGE, ACTIVATE, CAS) take up roughly equal amounts of time, approximately 13 ns each in modern DDR3 systems [47], and the data transfer takes about 10 ns.

The memory scheduler has to consider resource availability and several timing constraints when issuing commands. Generally, the memory scheduler prioritizes reads over writes, accesses to open rows, and older requests over younger ones. DRAM writes are generated as a result of write-back operations from the LLC. Since writes are not on the processor's critical path, the memory-controller is not required to complete the write operation immediately and buffers the data in a write queue. One of the many timing constraints is the write turnaround delay (tWTR) that is incurred every time the bus changes direction when switching from a write to a read. Writes and reads are generally issued in bursts to amortize this delay overhead. Writes are buffered until the write queue reaches a high water mark (or there are no pending reads); the bus is then turned around and writes are drained until a low water mark is reached.

3.2.2 Simulation Methodology

We use the Wind River Simics [48] simulation platform for our study. Table 3.1 details the salient features of the simulated processor and memory hierarchy. We model an out-of-order processor using Simics' *ooo-micro-arch* module and use a heavily modified *trans-staller* module for the DRAM/PCM simulator. The DRAM simulator closely follows the model described by Gries in [47] and shares features with the DRAMSim framework [49].

In this work, we model a modest multicore (16 core) system with two channels to limit simulation time. The memory controller models a First-Ready-First-Come-First-Served (FR-FCFS) scheduling policy and models the timing parameters described in Table 3.2. The interplay of these timing parameters is crucial for evaluating DRAM bank management as maintaining the restrictions imposed by the parameters will significantly impact bank usage [21]. The parameters tRAS, tRRD, and tFAW are essential because they impose restrictions on how frequently accesses can be made to the same bank (or same rank) if the accesses are not row hits. In our simulator, our bank usage model adheres to these constraints.

The DRAM device model and timing parameters were derived from [21], [47]. We model multiple ranks per memory channel, each rank has several banks (each with its own row-buffer). The data and address bus models are accurately designed to simulate contention and bus turnaround delays. The DRAM pipeline model is equipped to handle both reads and writes. In the baseline model, writes are enqueued

Table 3.1. Simulator parameters.

Processor	
ISA	UltraSPARC III ISA
CMP size and Core Freq.	16-core, 3.2 GHz
Re-Order-Buffer	64 entry
Fetch, Dispatch, Execute, and Retire	Maximum 4 per cycle
Cache Hierarchy	
L1 I-cache	32KB/2-way, private, 1-cycle
L1 D-cache	32KB/2-way, private, 1-cycle
L2 Cache Coherence Protocol	4MB/64B/8-way, shared, 10-cycle Snooping MESI
DRAM Parameters	
DRAM Device Parameters	MT41J128M8 DDR3-800 [47], 8 banks/device 16384 rows/bank, x8 part
DRAM Configuration	2 64-bit Channels 1 DIMM/Channel (unbuffered, non-ECC) 2 Ranks/DIMM, 8 devices/Rank
Row-Buffer Size	8KB per bank
Active row-buffers per DIMM	8
Total DRAM Capacity	4 GB
DRAM Bus Frequency	1600MHz
DRAM Read Queue	48 entries per channel
DRAM Write Queue Size High/Low Watermarks	48 entries per channel 32/16

Table 3.2. Timing parameters.

Parameter	**DRAM / PCM**	**Parameter**	**DRAM / PCM**
tRCD	13.5ns / 55ns	tCAS	13.5ns
tRP	13.5ns	tWR	13.5ns / 125ns
tRAS	36ns / 55ns	tRRD	7.5ns
tRTRS	2 Bus Cycles	tFAW	40ns
tWTR	7.5ns	tCWD	6.5ns

in the write queue on arrival and the write queue gets drained by a specific amount upon reaching a high water mark. The simulator's command scheduling mechanism can overlap commands to different banks (and ranks) to take maximum advantage of the bank level parallelism in the access stream.

DRAM address mapping parameters for our platform (i.e., number of rows / columns / banks) were adopted from the Micron data sheet [47] and the open row address mapping policy from [21] is used in the baseline. We use this address mapping scheme because this results in the best performing baseline on average when compared to other commonly used address interleaving schemes [21], [49].

Our techniques are evaluated with full system simulation of a wide array of memory-intensive benchmarks. We use multithreaded workloads (each core running 1 thread) from the PARSEC [50] (*canneal, fluidanimate*), OpenMP NAS Parallel Benchmark [51] (*cg, is, ep, lu, mg, sp*), and SPECJVM [52] (*lu.large, sor.large, sparse.large, derby*) suites along with the STREAM [53] benchmark. We also run multiprogrammed workloads from the SPEC CPU 2006 suite (*bzip2, dealII, gromacs, gobmk, hmmer, leslie3d, libquantum, omnetpp, perlbench, soplex, xalancbmk*). We selected applications from these benchmark suites that exhibited last level cache MPKI greater than 2 and could work with a total 4 GB of main memory. Each of these single threaded workloads are run on a single core, so essentially each workload is comprised of 16 copies of the benchmark running on 16 cores. We also run a workload designated as specmix which consists of the following single threaded SPEC CPU 2006 applications: *bzip2, bwaves, milc, leslie3d, soplex, sjeng, libquantum,* and *gobmk.* We chose to model cache space per core (4 MB for 16 cores) and memory channels per core (2 channels for 16 cores) that are slightly lower than those in modern systems. This allows us to create the memory pressure per channel that may be representative of a future many-core processor without incurring the high simulation times of such a many-core processor.

For multithreaded applications, we start simulations at the beginning of the parallel-region/region-of-interest of the application, whereas for the multiprogrammed SPEC benchmarks, we fast forward the simulation by 2 billion instructions on each core before taking measurements. We run the simulations for a total of 1 million DRAM read accesses after warming up each core for 5 million cycles. One million DRAM read accesses correspond to roughly 270 million program instructions on average. For comparing the effectiveness of the proposed schemes, we use the total system throughput defined as $(IPC^{i}_{shared}/IPC^{i}_{alone})$ where IPC^{i}_{shared} is the IPC of program i in a multi-core setting. IPC^{i}_{alone} is the IPC of program i on a stand-alone single-core system with the same memory system.

3.2.3 Motivational Results

We start by characterizing the impact of writes on overall performance. Fig. 3.1 shows normalized IPC results for a few different memory system models. The leftmost bar represents the baseline model with write queue high/low water marks of 32/16. The rightmost bar *RDONLY* represents a model where writes take up zero latency and impose zero constraints on other operations. This represents an upper bound on performance that is clearly unattainable but shows that write handling impacts system performance by 36% on average for our memory-intensive programs. The bar in the middle represents an oracular scheme that is more realistic and closer to the spirit of the Staged Read optimization. It assumes that while writes are being serviced, all pending reads can be somehow prefetched (regardless of bank conflicts), and these prefetched values can be returned in successive cycles following the bus turnaround. This bar is referred to as *Ideal* in the rest of the chapter and shows room for a 13% average improvement.

Fig. 3.2 shows the break-up of the DRAM access latencies of the baseline and the Ideal cases. On the left of the graph, the two bars show the average latencies encountered by reads that have to wait for the write-queue to drain. By finishing

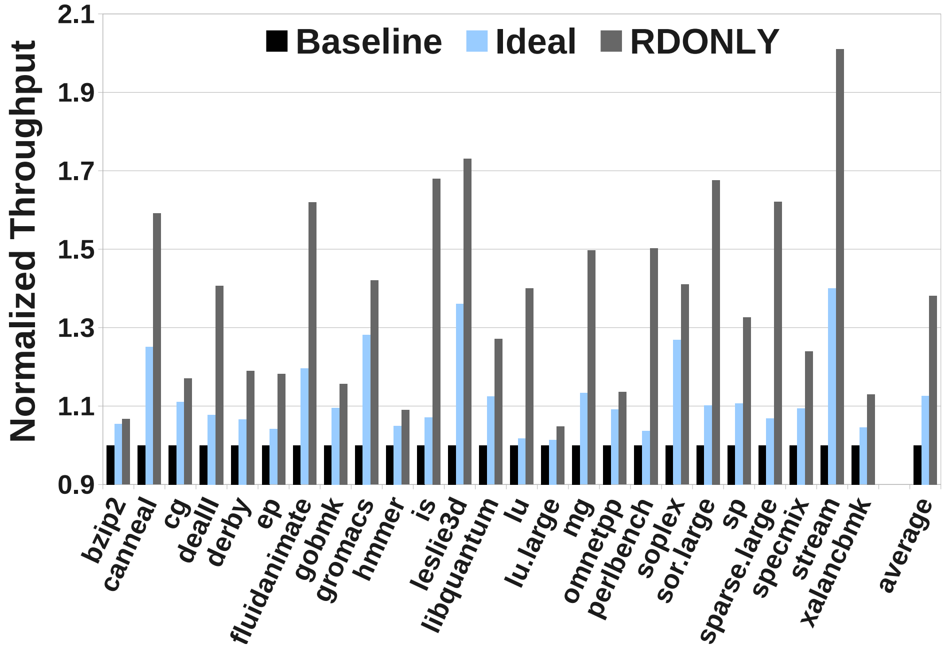


Figure 3.1. Room For Performance Improvement

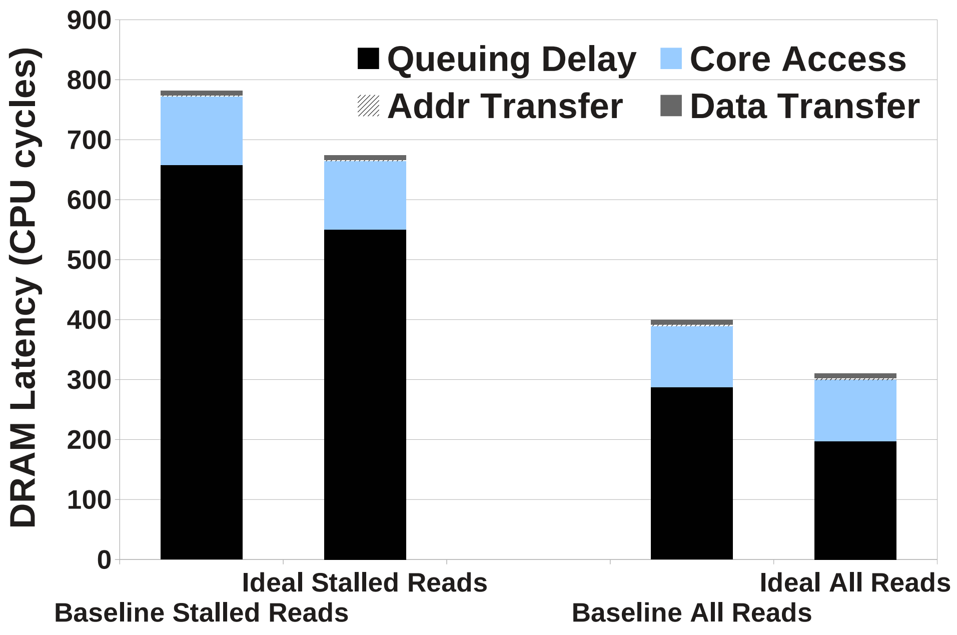


Figure 3.2. DRAM Latency Breakdown

the bank access of the reads in parallel with the writes, the Ideal configuration can substantially lower the queuing delay, showing that ramping up the read-pipeline after the write-to-read turnaround is inefficient in the baseline. The impact of this speed-up is noticed in the reduced overall queuing delay for all reads in the system, as shown in the two rightmost bars in Fig. 3.2.

3.3 Staged Reads

3.3.1 Proposed Memory Access Pipeline

3.3.1.1 Baseline Scheduling

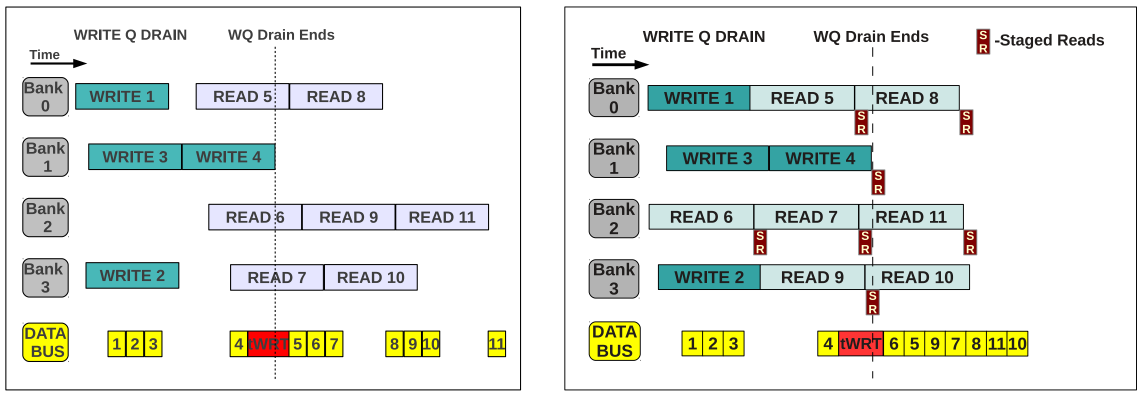
We assume a baseline scheduling process that is already heavily optimized. When the write queue is draining, we first schedule row buffer hits when possible and prioritize older writes otherwise while maximizing bank-level parallelism. For most of the write drain process, reads are not issued. As we near the end of the write drain process, as banks are released after their last write, we start issuing PRECHARGE, ACTIVATE, and CAS for the upcoming reads. These are scheduled such that the data are ready for transfer on the bus immediately after the bus is turned around. The pipeline is shown in Fig. 3.3(a). Note how the operations for READ-5 begin before the bus is turned around (shown in red by tWTR) and the data transfer for READ-5 happens immediately after the tWTR phase.

3.3.1.2 Optimization Opportunity

The key to the Staged Read optimization is that there are bus idle slots soon after the bus starts servicing reads (right after data transfer 7 in Fig. 3.3(a)) and bank idleness when servicing writes (Bank 2 in Fig. 3.3(a)). Bus idleness when servicing reads happens when the reads conflict for the same banks (Reads 6, 9, and 11 all conflict for Bank 2). These idle bus slots could have been filled if some of the reads for Bank 2 could have been prefetched during Bank 2's idle time during the write phase. The baseline scheduling policy can already start issuing up to one read per bank before the bus turnaround happens (for example, Reads 5, 6, and 7 in Fig. 3.3(a)). Thus, each bank already does some limited prefetch, whereby the bank access latency of some reads are hidden, with the prefetched lines remaining in the bank's row buffer. The Staged Read optimization is intended to provide prefetch beyond this single read per bank.

3.3.1.3 Timing with Staged Reads

Fig. 3.3(b) shows how Reads 6, 9, and 11 for Bank 2 (and all other reads except for Bank 1 reads) are moved further to the left. As soon as a bank is done servicing writes, it starts to service reads. If the read finishes before the bus is turned around, the



(a) (b)

Figure 3.3. Timing for reads and writes in (a) Baseline, and (b) Staged-Reads.

resulting cache line is saved in a Staged Read register near the I/O pads (shown by the SR box in Fig. 3.3(b)). After the bus is turned around, data blocks are either returned via the normal process (i.e., from the sense amplifier row-buffer through a traditional column-read command) or from Staged Read registers. As seen in Fig. 3.3(b), the bus is kept busy for a number of cycles following the turnaround. Many studies, including ours, show that bank conflicts are the source of bus idle cycles and consequently long queuing delays. Hence, such prefetch operations have a favorable impact on the latency of many subsequent reads.

3.3.2 Staged Read Implementation

As our results show later, most programs do well with 16/32 Staged Read registers. Assuming 64 byte cache lines spread across a rank of eight chips, this corresponds to a storage overhead of only 128/256 bytes per DRAM chip. This is much more efficient than prior proposals that have advocated the use of row buffer caches within DRAM chips [54], [55]. The proposed optimization is much less invasive than row buffer caches for several reasons. First, row buffer cache entries retain entire rows, each about 8 KB in size (this is the rank or DIMM level row-buffer with each constituent chip in a rank contributing 1KB). Second, if row buffer caches are placed centrally near the I/O pads, an enormous amount of overfetch energy and latency is incurred in moving the entire row to the central row buffer cache. If row buffer caches are distributed among arrays, the area and layout of highly optimized arrays is impacted. DRAM chips employ a limited number of metal layers and it is a challenge to introduce a latch or SRAM structure for the row buffer cache directly adjacent to the arrays themselves. Third, row buffer caches are speculative; entries are retained in the hope that future accesses will reuse data in these entries. The Staged Read optimization does not suffer from any of these problems. The registers only store the specific cache line that will be requested in the near future. As shown in Fig. 3.4, the registers can be located centrally near the I/O pads and be shared by all banks. This is feasible because their overall capacity is small and no additional data (compared to the baseline) are being shipped across global wires to the I/O pads. Thus, the prefetch does not impact overfetch energy on a DRAM chip and the only energy penalty is the cost of reading data in and out of Staged Read registers. The proposal also

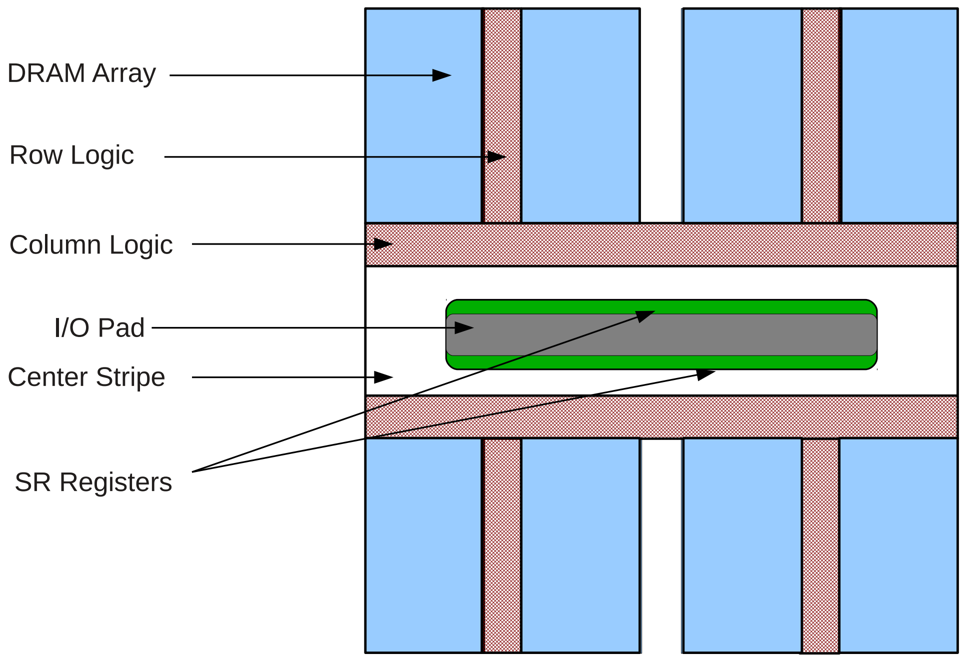


Figure 3.4. Floorplan of DRAM Chip

does not impact the layout of the dense array structures, which represent the bulk of the memory chip area footprint. The I/O pad area of a DRAM chip occupies a central strip on the DRAM chip. It contains an I/O gating structure that is shared by all banks and by reads and writes. In order to promote read-write parallelism with Staged Reads, the Staged Read registers must be placed directly before the on-chip global wires for data reaching the I/O gating structure. It is well known that changes to a DRAM chip must be extremely cost sensitive. Vogelsang [43] points out that changes to a DRAM chip are most costly when instituted in the bitline sense-amplifier stripe, followed by in the local wordline driver stripe, then in the column logic, and finally in the row logic and center stripe. We are therefore limiting our modifications to the least invasive portion of the DRAM chip.

3.3.2.1 Area Overhead

At a 32 nm process, a 256 byte register file (corresponding to 32 Staged Read registers) requires 1000 square microns [56]. Most of the overhead can be attributed to a new channel that must be implemented between each bank and the Staged Read register pool, as shown in Fig. 3.5. DDR3 has a burst length of eight, meaning that each bank within an x8 device will be sending 64 bits of data to the I/O pads.

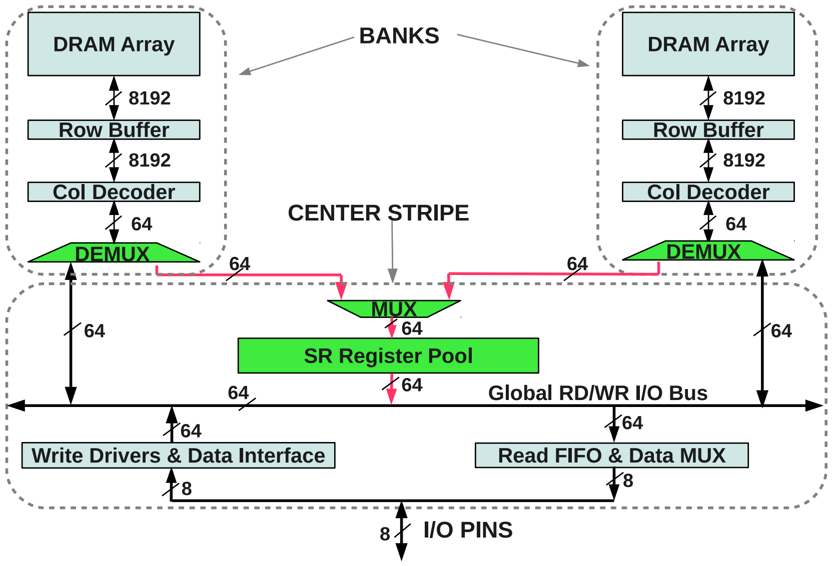


Figure 3.5. Organization of Staged Read Registers

As DRAM core frequency is less than off-chip DDR bus frequency, to sustain high bandwidth, all the 64 bits are sent from a bank to the I/O pads in parallel. These data are then serialized and sent 8 bits at a time through the DDR bus. For a wire pitch of 2.5F (where F is the feature size), a channel with 64 wires will have a pitch of 5.1 microns. Even after considering the overhead of eight such channels, for eight banks connected to the Staged Read register through a mux, the net area overhead is approximately less than 0.25% in a 50 square mm DRAM device. Note that DRAM layout is heavily optimized for area, and the actual overhead can deviate slightly from the above based on how transistors are laid and wire pitches are employed for buses and Staged Read registers.

3.3.2.2 Effect on Regular Reads

With our proposed implementation, after a row is activated and brought into the row-buffer and a cache-line is read from it, it has to choose one of two paths, i.e., either the regular bus to the I/O pins or the bus that feeds into the Staged Read registers. As shown in Fig. 3.5, this is accomplished by a simple demultiplexer to choose between one of the two paths, which introduces a 1FO4 gate delay to every read (regular and staged). However, this is less than 1% of the DRAM read latency and hence has negligible impact on the performance of nonstaged reads.

3.3.2.3 New Memory Commands

The results in a Staged Read register must be accessed with a new low latency instruction. In addition to the conventional CAS instruction, we now have CAS-SR and SR-Read instructions. For Staged Reads, the CAS-SR brings a specified cache line to a specified Staged Read register. The SR-Read moves the contents of the specified Staged Read register to the processor. Assuming a common pool of Staged Read registers that are shared by all banks, both new instructions must specify a few bits to identify the Staged Read register being handled. The memory controller must track in-progress reads and their corresponding Staged Read registers. The address/command bus is never oversubscribed because each cmd/address transfer is a single cycle operation compared to a 4 cycle data burst, and we observe that the address bus has an average utilization of 15% in the baseline. In the baseline, a CAS command is accompanied by a single address transfer (column address). For Staged Reads, this is replaced by a CAS-SR command and two address transfers (column address and destination SR register-identifier) and a SR-Read command accompanied by a source SR register-identifier at the end of the WQ drain. The increased activity on the address/command bus while performing Staged Reads pushes the utilization up to 24%.

3.3.2.4 Implementability

A sign that such a proposal is implementable is the fact that some high-performance DRAM chips have introduced buffering at the I/O pads [21], [57]. Since the I/O gating structure is shared by reads and writes and since reads can begin only after the last write has moved past the I/O gating structure, Rambus Direct RDRAM devices introduced a write buffer at the I/O pads so that the incoming data could be quickly buffered and the I/O gating structure can be relinquished sooner for use by reads [21]. Our optimization is similar in structure, but the logical behavior is very different. In Rambus devices, the buffering is happening for writes on their way in so they can get out of the way of important reads. In our Staged Read optimization, buffering is happening for reads on their way out so they can get out of the way of an on-going write burst.

3.3.2.5 Targeting Niche Markets

Given that commodity DRAM chips are highly cost-constrained, there is a possibility that such innovations, in spite of their minimal cost impact, may be rejected for the high-volume commodity market. However, there are several DRAM memory products that are produced for niche markets where either performance or energy is given a higher priority than cost. Such memory products may either be used in a supercomputer or datacenter setting or in the mobile market. Samsung's LP-DRAM [58], [59] is an example of a low-power chip and Micron's RLDRAM [60] is an example of a high-performance memory chip. It is expected that the marketplace for such niche DRAM products might grow as the memory hierarchy starts to incorporate multiple memory technologies (DRAM, eDRAM, PCM, STT-RAM, Memristors, etc.). In such hybrid memory hierarchies, the focus on cost may shift to the PCM subsystem, while the DRAM subsystem may be expected to provide low latency with innovations such as Staged Reads. Recent papers [61], [62], [63] also advocate the use of a 3D stack of memory chips and a logic die. The interface die can be used for many auxiliary activities such as scheduling, refresh, wear leveling, interface with photonics, row buffer caching, etc. If such a design approach becomes popular, Staged Read registers could be placed on the logic die, thus further minimizing their impact on commodity DRAM chip layouts.

3.3.3 Exploiting Staged Reads—Memory Scheduler

From the description in Section 3.3.2, we see that for Staged Reads to be beneficial, there have to be enough opportunities for the controller to schedule Staged Reads to idle banks. The opportunity is high if there are some banks that are not targeted by the current write stream and those same banks are targeted by the current pending reads. While we see in Section 4.5 that such opportunities already exist in varying amounts for different benchmarks, we devise a memory scheduler policy that actively creates such bank imbalance. This best ensures that useful read work is performed during every write drain phase.

The write scheduler first orders all banks based on the simple metric:*pending writes minus pending reads.* Banks are picked in order from this list to construct a set of writes that, once drained, will help the write queue reach its low water mark. Thus, we are draining writes to banks that have many pending writes; banks that have many pending reads are being kept idle. With the above scheduling policy, referred to as the *Write Imbalance (WIMB)* scheduler, during every write drain phase, a bank will roughly alternate between primarily handling writes or primarily handling Staged Read operations. For example, in a 4-bank system, in one write drain phase, a number of writes may be sent to banks 0 and 1, while banks 2 and 3 are busy handling Staged Reads. In the next write drain phase, banks 2 and 3 are favored for write drains (because their pending write queue has grown), while banks 0 and 1 may handle Staged Reads.

3.4 Evaluation

3.4.1 Results

In this section, we analyze the performance impact of our innovation and also present a sensitivity analysis of Staged Reads. We present results for the following different configurations.

- **Baseline :** These experiments model the baseline DRAM pipeline and memory controller described in Section 3.2.2 (Table 3.1). The memory controller has a 48-element write queue (for each channel), which is drained once it reaches a high water mark of 32, until the occupancy drops to 16. In the baseline model, there is no provision for Staged Reads, which means that following a column-read command, the data are read out from the sense amplifiers and sent out over the I/O pins.
- **SR_X :** These configurations refer to systems that consist of DRAM chips and controllers that are, at a maximum, equipped to handle **X** Staged Read requests per rank. The timing specifications for Staged Reads are as described in Section 4.3. We consider the following values of X: 16, 32, and infinite.
- **SR_32+WIMB :** This refers to the configuration where the memory controller's write-scheduling policy is modified to direct writes at a small subset of banks in a rank. This exposes more free banks that can be used by the Staged Read mechanism.
- **Ideal :** As described earlier in Section 3.2.3, we also show a bar for the Ideal case as a reference. The Ideal case assumes that all pending reads can be

prefetched into an infinite set of Staged Read registers while they wait for a write drain cycle to finish.

The key difference between SR_Inf and Ideal is that SR_Inf continues to faithfully model bank conflicts and other timing constraints. Therefore, some pending reads in SR_Inf may not have the opportunity to issue their prefetch before the write drain is complete.

Fig. 3.6 shows the impact of Staged Reads on average DRAM latency. Fig. 3.7 shows the impact on normalized weighted throughput. The benchmarks are ordered from left to right based on the throughput improvement caused by the SR_32 configuration. Some applications, such as *ep, dealII,* and *lu.large*, do not exhibit much improvement with Staged Reads, even with an infinite register pool, whereas applications such as *stream, leslie3d,* and *fluidanimate* show marked improvements. The sensitivity of applications to our innovation is dependent on whether during a write drain cycle, there are enough reads that can be parallelized using Staged Reads and whether these reads would have introduced data bus bubbles in the baseline because of bank conflicts. To help understand the performance characteristics of the different configurations, we plot the average number of reads stalled during write queue drain cycles and the number of Staged Reads completed with 32 Staged Read registers in Fig. 3.8.

The best indicator for Ideal performance is the number of pending reads during each write induced stall period. The performance of the Ideal configuration is high in all cases where there are a large number of pending reads (the first series in

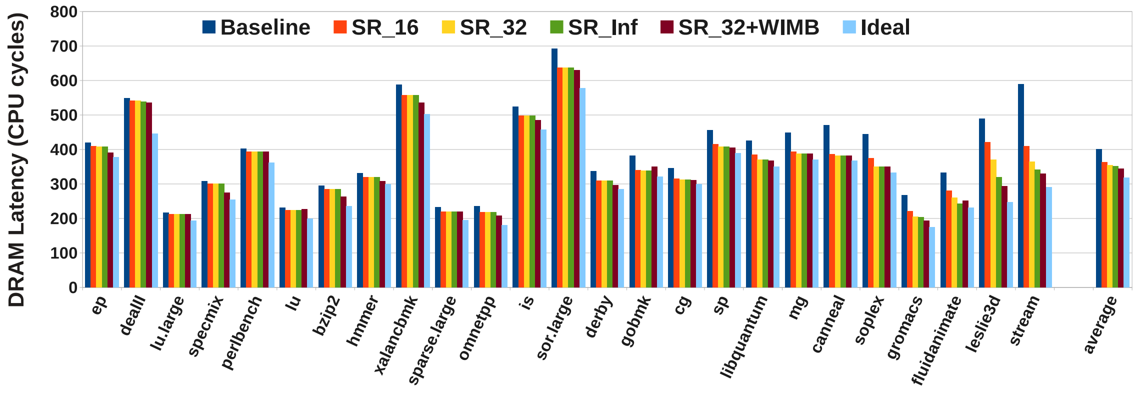


Figure 3.6. DRAM Latency Impact of Staged Reads

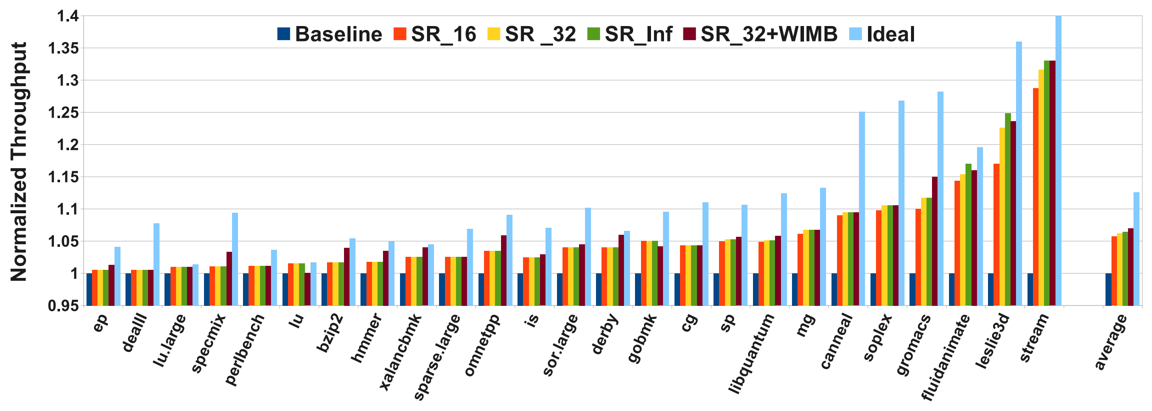


Figure 3.7. Performance Impact of Staged Reads

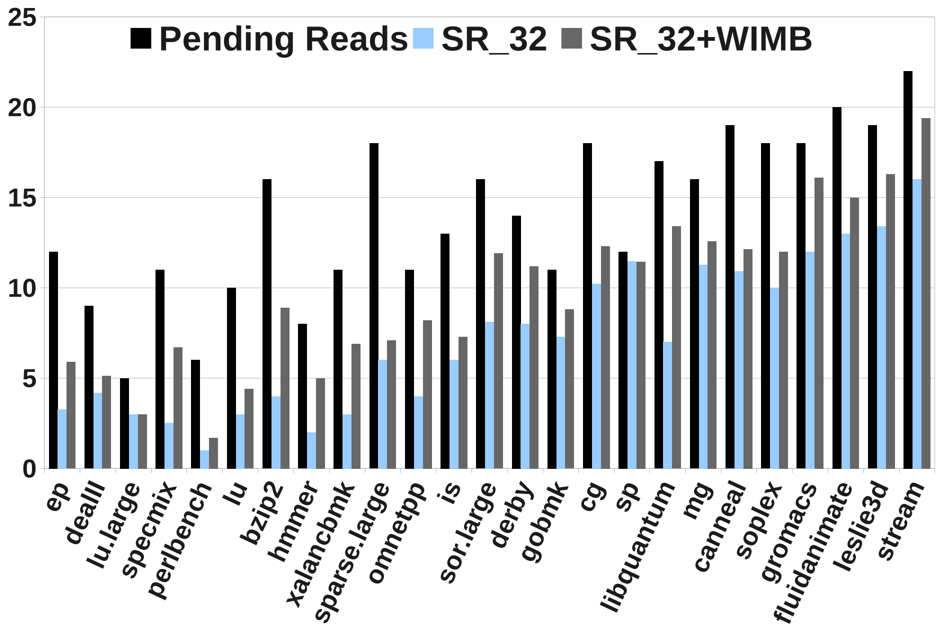


Figure 3.8. Average Number of Reads Stalled By Write Drains and Number of Staged Reads Completed Per Rank

Fig. 3.8). In a practical setting, however, it might not be possible for the Staged read mechanism to drain all these pending reads. There might not be enough bank imbalance between reads and writes for the Staged Reads to schedule read prefetches. Thus, an application like *sor.large* shows marked improvement with Ideal configuration (Fig. 3.7), because it has a high number of pending reads (Fig. 3.8). In reality, these reads can not be drained by Staged Reads as the writes in *sor.large* also touch a large number of banks (Fig. 3.9), thereby reducing the opportunity to

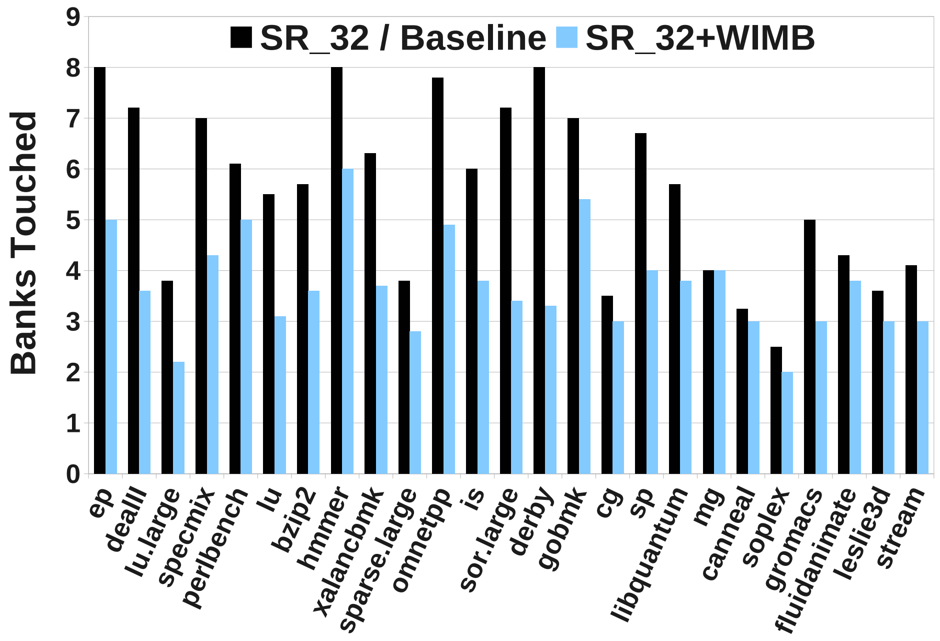


Figure 3.9. Average number of banks touched by writes per drain cycle per rank

carry out Staged Reads. In Fig. 3.9, we show the average number of banks that are engaged by writes during write queue drains by all applications.

Applications that have a high read bandwidth demand would naturally see many reads queuing up during write drain cycles - but the MPKI alone can not explain the response of an application to Staged Reads. The improvements obtained are influenced by bank imbalance between writes and reads. For the best performing applications with the SR_32 scheme, such as *stream, leslie3d,* and *fluidanimate*, there are relatively larger number of queued reads during each write-queue drain cycle (Fig. 3.8). A large fraction of these can be drained by Staged Reads, because few banks are touched by the writes during write queue drains in these applications (Fig. 3.9), leaving other banks ready to service Staged Reads. Applications that have a favorable combination of a large number of pending reads and a small number of banks touched by writes benefit the most from regular Staged Reads with the baseline scheduler. On the other hand, applications such as *lu.large* and *perlbench* have very few outstanding reads during write drains, while applications like *omnetpp* and *is* have the writes spread evenly over a large number of banks, leading to lower benefits.

Increasing the number of Staged Read registers improves the latency for some benchmarks (Fig. 3.7). For example, by going from SR_16 to SR_32, the average latency of accesses drops by approximately 8% for applications like *stream* and *leslie3d*. For most other applications, 16 registers are enough to handle all pending reads, an observation verified by Fig. 3.8. For the benchmarks we evaluated, SR_32 performs as well as SR_Inf on all occasions except for *stream*, *leslie3d*, and *fluidanimate*, where during some drain cycles, more than 32 SR registers can be useful, but this is not a common occurrence.

On average, the best performing scheme is the SR_32+WIMB configuration that yields a 7% improvement in throughput. By actively creating idle periods for some banks that have a lot of pending reads, this configuration offers a good opportunity for these reads to be completed using Staged Reads. As seen in Fig. 3.9, the average number of banks touched by writes decreases due to the biased write-scheduling policy for almost all cases. For applications like *gromacs, derby,* and *omnetpp*, the bank imbalance is increased favorably, and this is reflected in the additional benefit obtained by SR_32+WIMB over regular SR_32; more Staged Reads are completed for these applications (Fig. 3.8) with the novel write-scheduling policy. Applications like *cg* and *mg* that already touched a small number of banks do not derive any additional benefits over SR_32 with the modified write-scheduling policy. There are also applications such as *gobmk* and *lu* where the improvement with SR_32 and a regular write-scheduling policy is greater than the SR_32+WIMB policy. This can be explained by the fact that in these benchmarks, the explicitly reduced bank footprint of writes leads to a lengthening of the average write-queue drain cycle, which diminishes the benefits of Staged Reads. However, in none of the cases do we see any performance degradation compared to the baseline.

The gap between the Ideal configuration and SR_Inf in Fig. 3.7 cannot be bridged by parallelizing reads and writes. This results from reads waiting on the same banks as targeted by the writes—a problem that can not be alleviated with regular Staged Reads, but which is abstracted away in the Ideal configuration. By using the SR_32+WIMB configuration, this is ameliorated to a certain extent for most applications.

Overall, we witness a 17% reduction in average DRAM latency across the benchmark suite resulting in a 6.2% improvement in overall system throughput with 32 Staged-Read registers; this grows to 7% with the modified write-scheduling policy. Half of the simulated benchmarks yield an improvement higher than 3%, and for these write-intensive benchmarks, the average throughput improvement is 11% with SR_32.

Figs. 3.10 and 3.11 shed further light into the DRAM latency impact of Staged Reads. Fig. 3.10 shows the DRAM latency breakdown for DRAM read requests. We see that read requests waiting for a write queue drain to finish have very long queuing delays in the baseline which are brought down substantially by using Staged Reads (Fig. 3.10) leading to lowering of overall DRAM latency. However, even after a read goes through the Staged-Read phase, it has to wait in the Staged-Read register for some amount of time before it can be sent out over the bus, which we refer to as the staged-wait latency. Recall that when a read request goes into the staged-read register, it has already finished its bank activity and the next request to the bank can start immediately. By reducing the bank-wait for pending read requests, the bus utilization after the write-queue drain is increased.

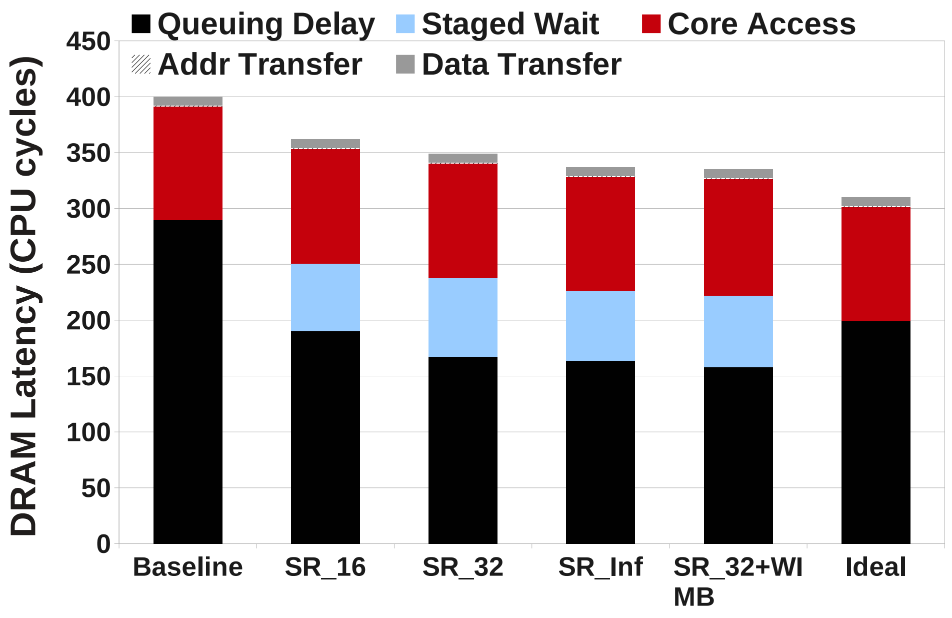


Figure 3.10. DRAM Latency Breakdown For All Read Requests

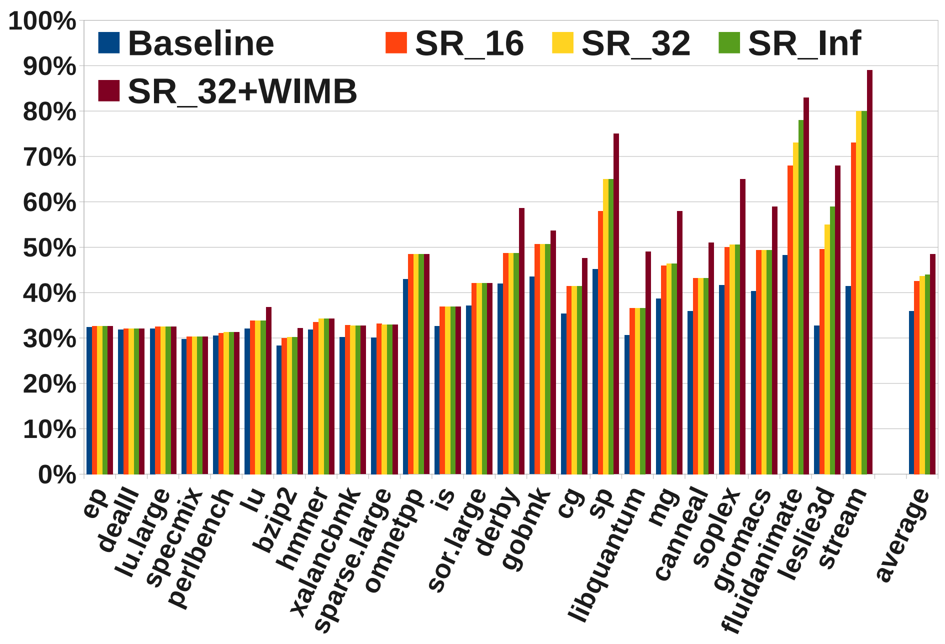


Figure 3.11. Bus Utilization Immediately After WQ Drain

Fig. 3.11 shows the bus utilization in the period following the write queue drain until all the reads that had arrived before the end of the write-queue drain are completed. With Staged Reads, the utilization in this period increases by as much as 35% for STREAM and about 22% on average. Applications that demonstrate the maximum bus utilization in this period with Staged Reads also derive the maximum benefit. We do not show the bus utilization in this window achieved by the Ideal configuration because it is 100% for all applications by design.

3.4.2 Sensitivity Analysis

To estimate the extent of the influence of our choice of DRAM parameters on the performance of Staged Reads, we tested the following factors that can potentially impact the efficacy of Staged Reads: write queue high and low water marks and a higher number of banks.

3.4.2.1 Write Queue Parameters

The choice of the high and low water marks for the write queue will determine the duration and frequency of write drain cycles. This in turn determines for how long (and how many) reads are stalled due to the write queue drain and also how frequently this disruption occurs, respectively. It also determines if enough bank imbalance is observed, both during and after the write drain.

We ran simulations with various values of the write queue drain parameters. With large values of the high water mark, draining of writes can be delayed—potentially reducing the adverse impacts of writes. In such a case, it is also important to drain the write queue by a large amount each time because otherwise the gap between write-queue drains will not be reduced. We present results for two different configurations with high water marks of 16 and 128 and low water marks of 8 and 64, respectively. Fig. 3.12 shows the results for Staged Reads with a high/low water mark of 16/8.

However, it is important to note that a baseline system (not capable of Staged Reads) which employs high/low water marks of 32/16 performs better on average compared to the 16/8 and 128/64 configurations. We find that by frequently initiating write-queue drains (i.e., a smaller high water mark) bandwidth hungry applications get penalized so that the IPC drops by about 2% on average. Again, by using a large value for the high water mark, we risk stalling some critical reads at the head of the out-of-order core's reorder buffer for a long duration. Thus, with a high value for the high water mark, some applications perform better than the baseline (i.e.,

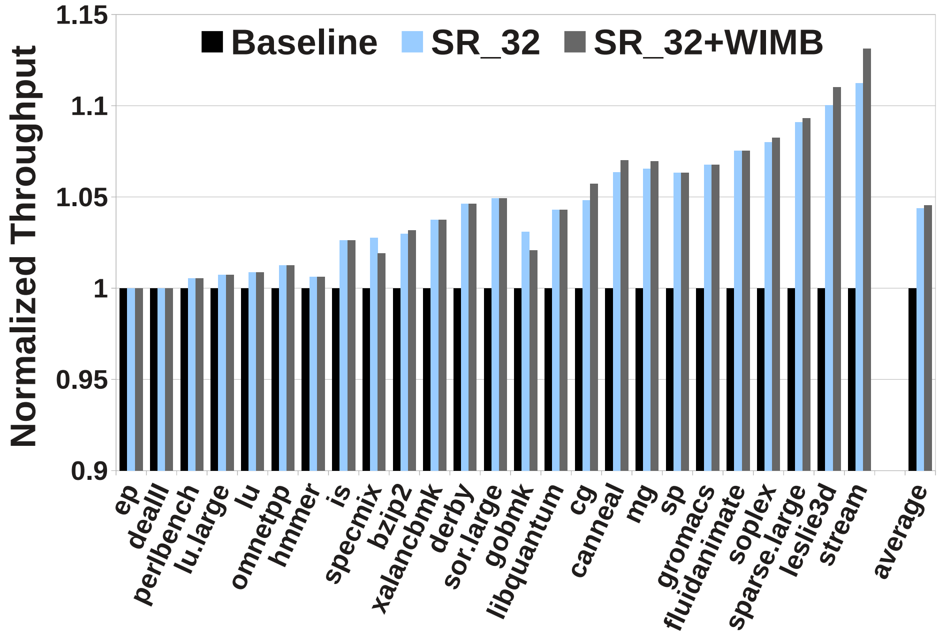


Figure 3.12. Staged Reads with High Water Mark = 16, Low Water Mark = 8

high/low water mark 32/16) and some applications perform worse, leading to a 1.4% performance degradation on average. Therefore, for the workloads we simulated, the best performing write queue configuration is the one used for the baseline.

We see that in both the 16/8 and 128/64 cases, Staged Reads can offer performance improvements as shown in Fig. 3.12 and Fig. 3.13. The results show the same trends as before, but the improvements are slightly lower (just under 5% on average) with SR_32 (which has no scheduler induced write-imbalance). With low water marks, there are just not enough pending reads that can be expedited with Staged Reads. In fact, even with the write-scheduler creating write-imbalance, there is no extra improvement due to the dearth of writes. On the other hand, with high water marks, there is less write imbalance with a regular scheduler, which can be alleviated by SR_32+WIMB as it has a much larger pool of writes to choose from. This leads to the SR_32+WIMB creating more oppurtunities for Staged Reads—finally yielding a 7.2% improvement over the baseline.

3.4.2.2 More Banks

The efficacy of Staged Reads increases if the reads pending on a write-drain cycle are directed at banks that do not have many writes going to them. With a larger

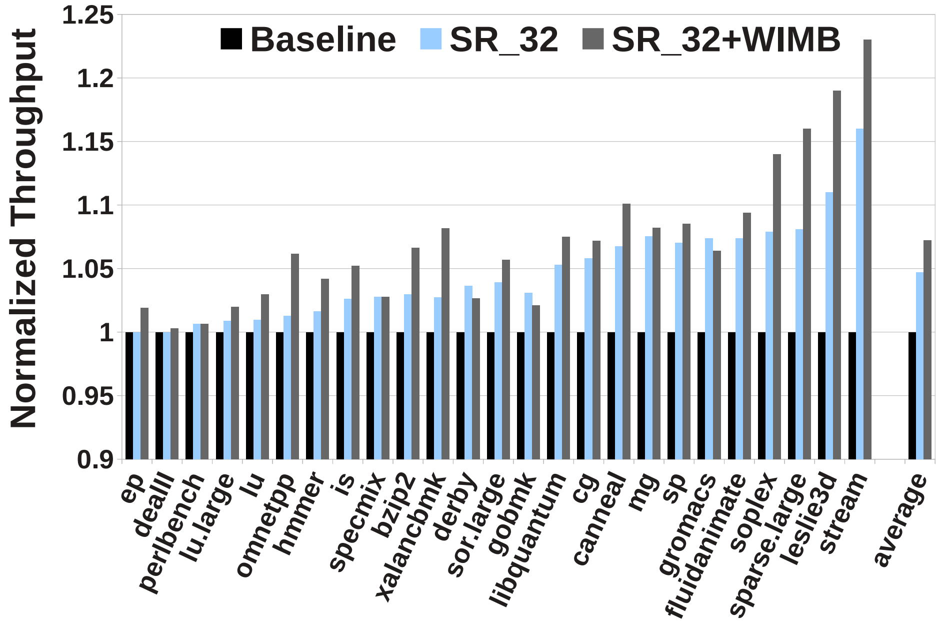


Figure 3.13. Staged Reads with High Water Mark = 128, Low Water Mark = 64

number of banks, the possibility of the memory controller being able to find more opportunities for scheduling Staged Reads increases. On the other hand, if more banks exist, the baseline suffers from fewer bank conflicts for reads and fewer data bus bubbles following the write drain. So there are competing trends at play with more banks. With queueing delays being the most dominant portion of the DRAM latency encountered by read requests, it is intuitive that adding more banks to the DRAM system will alleviate the situation. We carry out simulations where the same 4GB capacity as the baseline system is split into twice the number of banks (i.e., 16 banks/rank). We observe that a regular DRAM system (without Staged Reads) with more banks performs better than the baseline by about 3.1%. Applying our SR_32 configuration on this improved system yields an average improvement of about 4.3%. Due to increased bank parallelism, the performance benefits with SR_32+WIMB are comparable to regular staged reads. Thus, as a performance optimization for next generation memories, it is more effective to add Staged Read registers than to double the number of banks.

3.4.3 Projecting for Future Main Memory Trends

In this section, we evaluate if the Staged Read optimization will be more compelling in future memory systems. We examine a number of processor configurations that might represent these future trends: memory systems with reliability support, nonvolatile memories, and fewer channels per core.

3.4.3.1 Higher Write Traffic

With errors in DRAM becoming a major source of concern [44], [64], DIMMs equipped with error protection measures are being employed in datacenters. In one possible chipkill implementation (to overcome the failure of one DRAM chip on a rank), each DIMM can have a separate chip that stores either parity information per byte or an Erorr Correcting Code (ECC) word per 64-bit word. On each read, the information in the extra chip can help with error detection and possibly correction. If the information is not enough to correct the error (as may happen with multibit errors), a second-tier protocol is invoked. In RAID-like fashion, parity is also maintained across DIMMs. If a DIMM flags an uncorrectable error, information from all DIMMs is used to reconstruct the lost data. Such RAID-like schemes have been implemented in real systems [65] and will likely be used more often in the future as error rates increase near the limits of scaling. As is seen in any RAID-5 system, every write to a cache line now requires us to read two cache lines and write two cache lines. This causes a significant increase in write traffic.

We simulate a RAID-5 like system where a fifth DIMM stores the parity information for the other 4 DIMMs. The baseline system in such a scenario encounters double the write traffic as seen in a non-ECC scenario. This, coupled with the increased number of reads makes Staged Reads more compelling. We see that the throughput increases by an average of 9% (Fig. 3.14) by using 32 Staged-Read registers. Compared to a unreliable baseline, we see a shorter gap between write-queue drain cycles which improves the benefits of Staged Reads. Using SR_32+WIMB, the average throughput does not increase beyond what is provided by SR_32, since a data write and its corresponding ECC code write have to be completed together, the write-scheduler is not able to reorder the writes to expose any additional staged read opportunities.

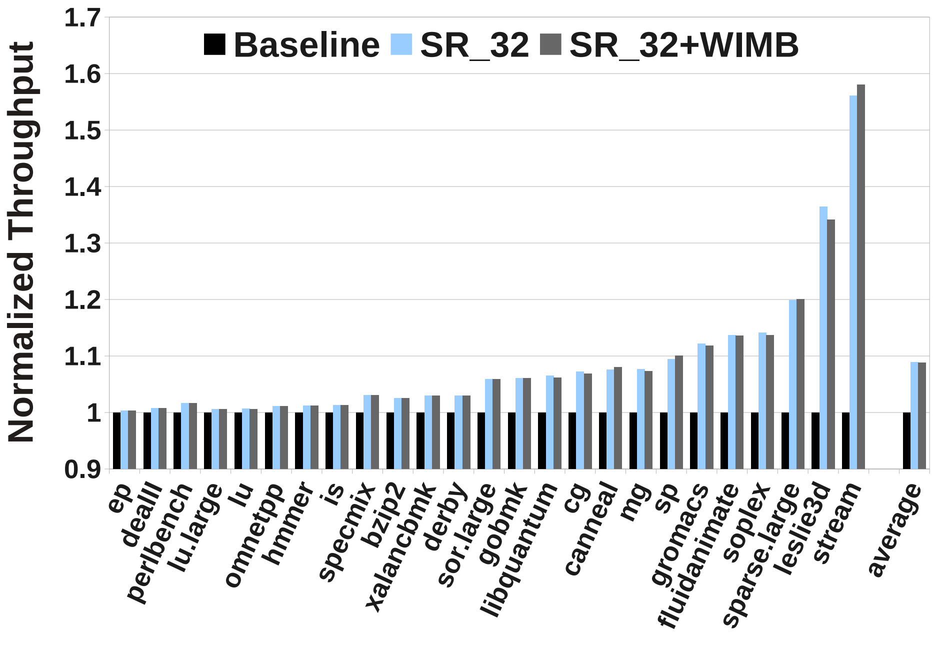


Figure 3.14. Staged Reads with RAID-5 like Chipkill Protection

3.4.3.2 Phase Change Memory (PCM)

Recent work [46], [66], [67] advocates the use of PCM as a viable main memory replacement. Similarly, other nonvolatile memory (NVM) technologies with read and write latencies longer than those of DRAM are also being considered [68]. We assume that the PCM main memory is preceded by a 16 MB L3 eDRAM cache (L3 average latency of 200 cycles). PCM chip timing parameters are summarized in Table 3.2; the PCM read and write latencies are approximately 2X and 4X higher than the corresponding DRAM latencies due to the high values of the row-activation (tRCD) and write-recovery (tWR) timing parameters.

The higher read and write latencies make the Staged Read optimization more compelling. We observe an average 26% reduction in memory latency because of a sharp drop in queuing delay. This translates to an average 12% throughput improvement (Fig. 3.15), with the Stream benchmark showing a 58% improvement. On the other hand, SR_32+WIMB does not offer as much advantage as regular SR_32—the performance improvement is 9.5% on average. The performance drop (compared to SR_32) is due to the high penalty of lengthening the write-queue drain cycle-time

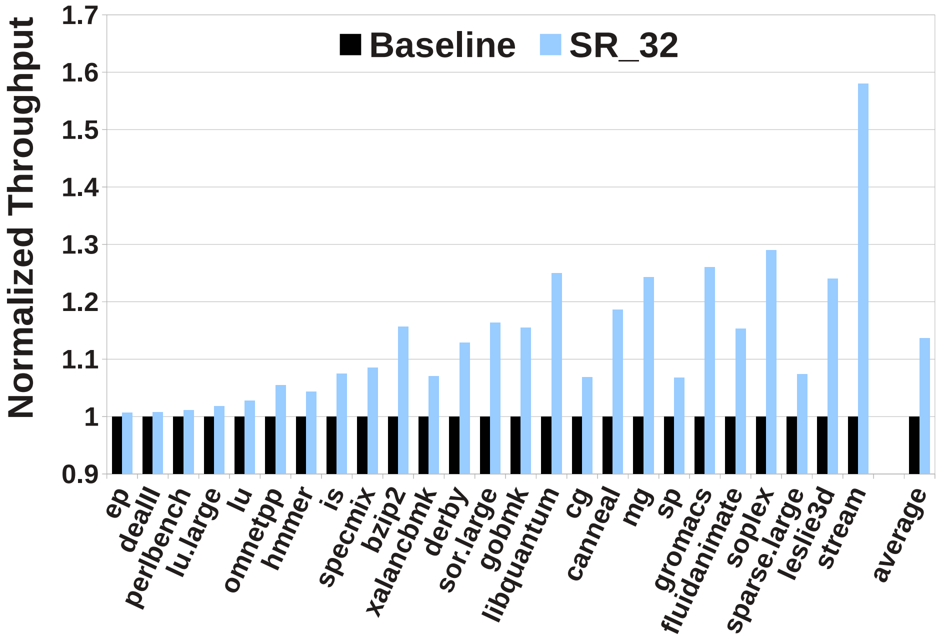


Figure 3.15. Staged Reads with PCM Main Memory

in PCM. By restricting bank-parallelism within writes, SR_32+WIMB ends up with bank conflicts on the targeted banks.

3.4.3.3 Number of Channels

With a greater number of channels, the pressure of pending reads on each channel would reduce and the benefits of Staged Reads would also diminish. With a quad-channel configuration, for our workload suite, the benefits of SR_32 is 3.3% and that of SR_32+WIMB is about 3.0%. However, the ITRS [69] projects an increase in the number of cores, but no increase in the number of off-chip pins. Hence, we expect that the number of channels per core will actually decrease in the future. If we instead assume that 16 cores share a single channel, the improvement with SR_32 jumps up to 8.4% because of the greater role played by queuing delays, while introducing write imbalance improves throughput by 9.2%.

3.5 Conclusions

We show that write handling in modern DRAM main memory systems can account for a large portion of overall execution time. This bottleneck will grow in future memory systems, especially with more cores, chipkill support, or nonvolatile main memories. This requires that mechanisms be developed to boost read-write parallelism. We show that the Staged Read optimization is effective at breaking up a traditional read into two stages, one of which can be safely overlapped with writes. We show average improvements of 7% in throughput for modern memory systems (accompanying a 17% reduction in DRAM access latency) and up to 12% for future systems. The proposed implementation has been designed to cater to the cost sensitivity of DRAM chips. We introduce less than a kilobyte of buffering near the I/O pads, similar to structures that have been employed for other functionalities in some prior high performance DRAM chips. We therefore believe that the Staged Read optimization is worth considering for DRAM chips designed for the high-performance segment.

CHAPTER 4

A DRAM SCHEDULER OPTIMIZED FOR GPUS

4.1 Introduction

Graphics Processing Units (GPUs) have emerged as an efficient alternative to traditional scalar processors for a large class of data parallel workloads. High-level programming models such as NVIDIA's CUDA [6] and OpenCL [7] allow the programmer to define the behavior of a single scalar thread, which is then replicated to run many threads on Single Instruction Multiple Data (SIMD) execution units (also called compute units or CUs). The viability of executing highly-threaded parallel workloads on general-purpose GPUs (GPGPUs) has led to the development of GPU implementations of algorithms that are considered "irregular" for GPUs [8], [36], [70], [71]. A study of these applications on GPU hardware demonstrates significant memory-access irregularity (MAI) [37]. The memory-access patterns are data dependent and consequently have less locality, thereby differing from the streaming access patterns typical of graphics and many compute workloads.

The GPU architecture is most efficient for executing graphics and compute workloads with regular patterns. The SIMD cores in a GPU can run many threads in parallel. A group of threads running in lockstep in such a setup is termed a warp (or a wavefront) with every thread in the warp executing the same instruction. A load instruction in such a SIMT (Single Instruction Multiple Thread) system can generate many different memory requests and the warp becomes ready to run (becomes "runnable") only when all of the outstanding memory requests are returned to the compute unit. The compute units are simple and typically do not employ the latency-hiding techniques commonly used in out-of-order, speculative, superscalar processors. GPUs use thread-level parallelism to tolerate memory access delays. Thus, when a warp waits for its memory requests to return from the memory system, the thread scheduler in the GPU picks a different warp that is ready to run. A GPU thus must maintain a large pool of warps (each with its own register file state) to mitigate the delays encountered by a warp stalled on a load. However, previous studies [72], [73] have shown that in spite of having a large number of thread contexts to choose from, the GPU's execution units often sit idle as all the warps are stalled on memory access. For instance, recent NVIDIA GPUs support at most 48 to 64 warps within a compute unit [74], while main memory latencies have been measured to exceed 400 cycles [75]. Thus, it is clear thread-level parallelism cannot always hide the memory latency completely.

The memory requests issued by a warp will also typically encounter different memory latencies. A subset of the requests might register hits in the different cache levels and the rest will be serviced by the DRAM. While the latencies of accessing data in the different levels of caches is obviously different, different requests to the DRAM will also encounter different latencies owing to the load on the DRAM channel, the complexities of the DRAM pipeline, and the memory scheduling algorithm. In fact, modern memory controllers schedule incoming requests out-of-order to maximize memory system throughput, which can stall a subset of the requests from a warp while memory requests for other warps (and other GPU functions) are serviced. This introduces the problem of *memory latency divergence* where a warp is stalled until the last memory request from a vector load instruction is returned to the compute unit. Several studies have highlighted how memory latency divergence can be a significant performance bottleneck in GPUs [18], [19].

In this chapter, we look at techniques to reduce the negative impact of memory latency divergence in GPUs by making the memory system hardware warp-aware. We propose main memory scheduling schemes that try to reduce the average memory-stall time for a warp (i.e., the time it takes for *all* requests of a warp to finish). We observe that the effective DRAM latency for a warp is often lengthened at the main memory because one or more requests of that warp are returned with longer latencies than the rest. We propose a DRAM scheduling strategy that attempts to reduce the intrawarp memory latency divergence by eliminating interwarp interference. The scheduling policy isolates requests from a warp in their own warp-group (a batch of requests) and takes scheduling decisions at the granularity of warp-groups. We evaluate different strategies for the scheduling of warp-groups. These strategies are motivated by the need to coordinate between different memory schedulers when a single warp sends requests to multiple memory controllers and also by the need to maintain good bandwidth utilization for some applications while minimizing latency. We first propose schemes that reduce intrawarp latency divergence in a single memory controller (BASJF, Section 4.3.3), then augment it to be implicitly multicontroller aware (BASJF+AB, Section 4.3.4). We then further optimize the scheduler to regain lost bandwidth utilization (MERB, Section 4.3.5.1). We couple the scheduler with a write-drain mechanism that reduces write-induced stall times for warps (WAWD, Section 4.3.6). The combined techniques reduce the adverse effects of memory latency divergence and improve performance by 8.6% on average for a set of GPGPU workloads.

4.2 Background

In this section, we take a brief look at the typical architecture of a modern GPU and that of the main memory system, including scheduling policies.

4.2.1 GPU Cores

Fig. 4.1 shows the basic architecture of a modern GPU. The GPU consists of a number of shader cores (Streaming Multiprocessor or SM in NVIDIA parlance). Each shader core executes a group of threads in Single Instruction Multiple Thread (SIMT) fashion. We model 32 SIMD lanes in each SM. A group of threads executing on the different lanes of the SIMD processor in lockstep is referred to as a warp (in NVIDIA parlance) or a wavefront (in AMD parlance). A cluster of such warps (or groups of threads), called a Cooperative Thread Array (CTA) or thread block, is assigned to each core. A kernel consists of one or more CTAs and there is a global CTA scheduler that issues CTAs to the cores taking into account the total number of CTAs required by the application and the resources available. The SMs are in-order cores; thus, when a warp is blocked waiting for the result of a load instruction, the warp scheduler within the SM will pick a ready warp to schedule on the SM. A warp

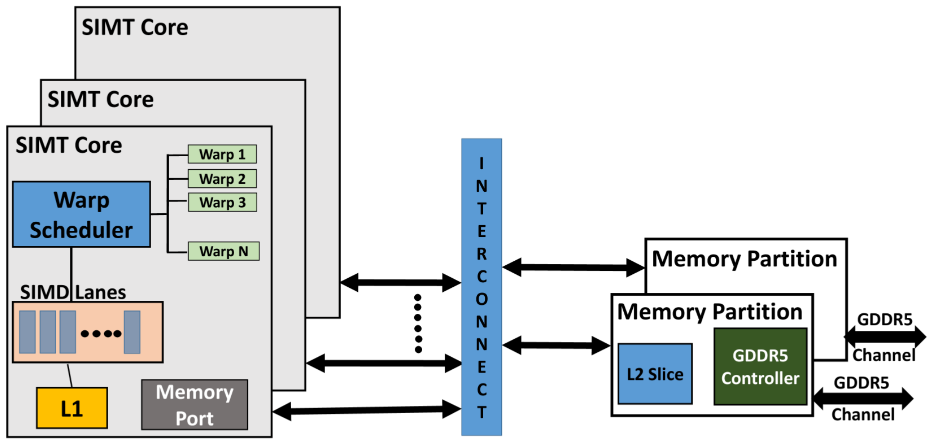


Figure 4.1. Simulated GPU Architecture

will remain blocked until all of its memory requests are returned by the memory system.

4.2.2 Memory System

The SMs have private L1 caches and are connected to different memory partitions through a crossbar interconnect. Each memory partition consists of a slice of the shared L2 and a GDDR5 channel. The L1 and L2s employ LRU replacement and write-evict strategies. Each GDDR5 [76] channel is typically 64-bits wide with the command and address bus running at 1.5GHz. The data bus runs at twice the frequency of the address/command bus and it is dual-data-rate (i.e., it transmits data at both the rising and falling edges of the data bus clock). Each GDDR5 chip has 16 banks and we consider 2 GDDR5 devices per channel that are operated in tandem (as one rank). The GDDR5 chip architecture is specialized for high bandwidth. Apart from higher bank counts and increased operating frequency, there are other architectural improvements such as a more robust power delivery network which allows a higher frequency of activates compared to DDR3 (i.e., a lower tFAW parameter) that leads to the higher performance of GDDR5. In the following two sections, we discuss the features of the baseline throughput-optimized memory controller.

4.2.3 Baseline Memory Controller Organization

Memory controllers in throughput processors are optimized to provide high bandwidth. Fig. 4.2 shows the important components of a typical memory controller that we model in this study.

Ⓐ The **Read Queue** buffers the read requests received from the interconnect. Typically, each entry contains the address of the request, the source SM identifier, and the source warp identifier.

Ⓑ The **Write Queue** buffers similar information as the read queue, but for write requests.

Ⓒ The **Transaction Scheduler** is the heart of the memory controller. The scheduler can pick transactions based on locality considerations (row-buffer hits vs row-buffer misses), request age, thread priorities, and other such high-level considerations. It is also responsible for interleaving write and read requests. The transaction scheduler picks a request each cycle from the read or write queues and enqueues a series of DRAM commands (ACT, PRE, COL_RD, etc.) into the appropriate command queue (labeled Ⓓ) in Fig. 4.2. The main throughput-optimizing features of the baseline scheduler are the address mapping policy and the scheduling policy. In our baseline, these two features are as follows.

4.2.3.1 Address Mapping

A high throughput memory-address scheme has to preserve row-buffer locality and maintain high pin utilization. The translation of cache-line addresses to the DRAM channel, bank, row, and column addresses used by the memory controller is a key element of the memory controller design. First, consecutive cache-lines are mapped to the same row in the same bank to promote row-buffer locality. Blocks of consecutive cache-lines are interleaved across the memory channels at a granularity of 256 bytes. To prevent pathological channel camping, where unusual access strides lead to excessive contention on one or few channels, the channel address is formed by XOR-ing a subset of higher-order bits with some lower-order bits to allow better spread across channels. Similarly, to prevent strided accesses from camping on the same bank, the bank address is formed by XOR-ing the bank address bits with a portion of higher-order address bits [29].

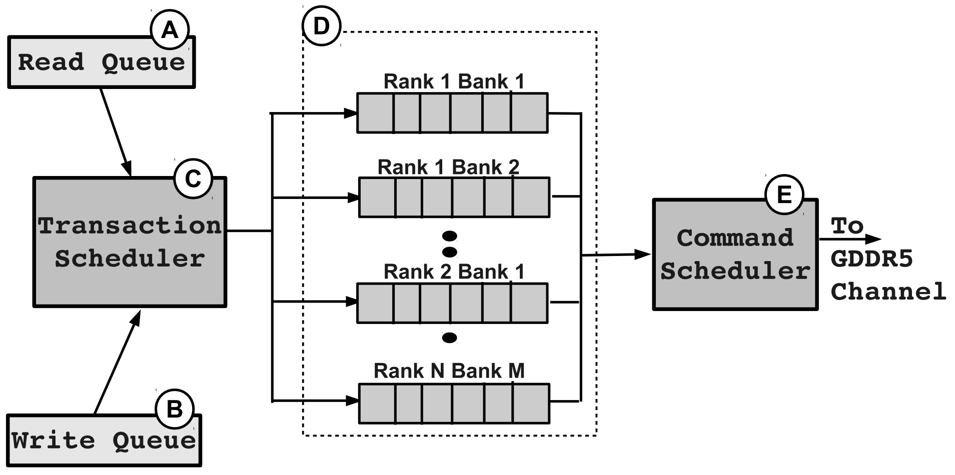


Figure 4.2. Baseline Memory Controller Structure

4.2.3.2 Baseline Scheduling Policy

The transaction scheduler picks a request each cycle from the read queue or write queue and enqueues the appropriate commands in the corresponding command queue. The transaction scheduler's primary responsibility is to exploit row-buffer locality. It thus aggressively reorders read requests to match the open row in each bank similar to the commonly used FR-FCFS policy [12]. Since the command schedulers processes commands from the command queues in order, the transaction scheduler maintains a table of the open-rows in each bank and scans the entire length of the read queue in order to select row-buffer hits. However, to prevent row-miss requests from being starved, it uses a combination of an age threshold based prioritization of old requests as well as a maximum row-hit streak control mechanism. Write requests to DRAM are the result of write-back from the last level cache and are thus not on the critical path for program execution. Writes are thus buffered in the write queue and are drained when the write queue occupancy rises above a high water mark or when there are no requests on the read queue [16]. The write requests are typically drained until the write queue has fewer than a fixed number of elements (the low water mark) or when too many read requests are stalled by the writes. Owing to the cost of the DRAM bus-turnaround delay ($tWTR$), reads are not interleaved with writes on the same rank [35].

Ⓓ The **Command Queues** in the memory controller store the low-level DRAM commands that need to be issued to the different banks to complete the transactions selected by the transaction scheduler. The queues are on a per-bank basis.
Ⓔ The **Command Scheduler** is responsible for issuing the DRAM commands to the GDDR5 chips. The command scheduler is cognizant of the low-level command timing restrictions and the state of the different DRAM banks. The command scheduler iterates over the different queues to interleave requests to different ranks/banks so as to leverage bank-level parallelism. However, within a bank, it orders requests in order to avoid disrupting the scheduling decisions taken by the transaction scheduler. The command scheduler is also responsible for issuing refresh commands in addition to the commands enqueued by the transaction scheduler. The command scheduler has a big impact on bank-level parallelism. It scans the head of the bank-level command queues in a round-robin fashion. If Ⓔ is not able to issue the command at the head of the queue in a cycle (because of timing constraints), it moves to the next bank's command queue. This ensures that while long-latency operations are being executed in one bank, the other banks are quickly engaged to serve other requests. The bank-group architecture of GDDR5 has the advantage of lower intercommand delays when the commands are issued to different bank groups [76] (e.g., the gap between consecutive column-read commands issued to different bank groups, *tCCDs*, is smaller than the gap required between column-read commands to the same bank group, *tCCDL*). The command scheduler thus tries to interleave requests between different bank groups first and then within each bank group through a multilevel round-robin policy.

4.3 Warp-Aware Memory Schedulers

As mentioned earlier, the efficacy of the DRAM scheduler will depend on the following two factors.

- The scheduling scheme's ability to return all of a warp's requests in close succession. This will also require the schedulers in different channels to have an efficient coordination mechanism amongst themselves.
- The scheduler's ability to maintain good bandwidth utilization and overall low memory latency by exploiting row-hits and bank-level parallelism.

With the above two goals, we examine a few different memory scheduling policies, each improving the preceding one in one or more aspects.

4.3.1 Warp-Aware Memory Controller Organization

First, we look at the changes that are needed to the memory controller hardware for enabling warp-aware scheduling. Fig. 4.3 shows the structure of the proposed warp-aware memory controller with the new addition, the **Batching Unit** Ⓕ, highlighted in blue. This is the batch formation stage that scans Ⓐ every cycle to form batches of requests from a warp. Requests from a warp may arrive at a memory controller interleaved with requests from other warps. The main job of the batch unit is to decide when a group of requests from a warp (called a *warp group*) is complete. We use a time-based threshold (T) to decide when a group can be considered complete. After time T has elapsed since the arrival of the first request for a warp, the batch unit considers the group complete and indicates this to the transaction scheduler. We observe that less than 3% of the average memory latency is required to completely collect the requests from a warp in the worst case (i.e., in the case when the interconnect is highly congested introducing variability in the arrival time of the warp's requests at the memory controller). Ausavarungnirun [15] *et al.* discuss the formation of batches of requests from each CPU. However, in their scheme, a batch is considered complete as soon as a request to a different row is received. We can afford to delay the formation of a batch because in a GPU the latency of the last request is the determinant of the performance. This is not the case in a CPU, where stalling a performance-critical memory request to group it with other memory requests might cause severe performance degradation. All the proposed memory schedulers described hereafter use Ⓕ.

4.3.2 Warp-Aware FCFS (WAFCFS)

The WAFCFS policy is designed for simplicity. The transaction scheduler picks the oldest complete warp-group and issues the corresponding requests in order to the different bank command queues. The transaction scheduler thus requires very little complexity, but is not cognizant of the state of the DRAM. The rationale behind this scheme is that it allows requests from a warp to be scheduled to the banks without

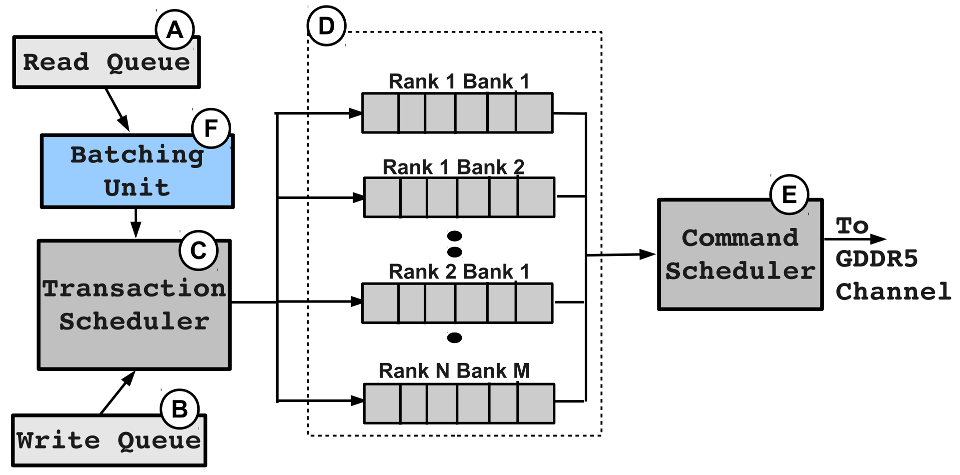


Figure 4.3. Memory Controller Organization

any interference from other warps. Also, the other important benefit of the WAFCFS scheme is that the different memory controllers service a single warp's requests in parallel. Since the requests from a warp show up at very similar times to the different memory controllers, selecting warp-groups based on their arrival order generally leads to simultaneous processing of a warp's requests in different memory controllers. While these two features of the WAFCFS scheme, i.e., warp-aware scheduling inside and across channels, can provide some performance advantage (by reducing memory divergence), it can severely degrade performance by being noncognizant of DRAM characteristics and thus needs to be improved upon.

This policy is similar to the complexity-effective memory scheduling scheme described by Yuan *et al.* [77]. In their work, the authors propose an intelligent interconnect which does not interleave requests from different SMs to allow a single warp's SM's requests to arrive at the controller in close succession. The scheduler uses a simple FCFS scheme to pick individual requests with the hope of being able to exploit row-locality within a SM's requests without the need of complex FR-FCFS-style reordering. With the batching unit forming warp-groups, a WAFCFS policy sees similar row-locality as the one proposed by Yuan *et al.*

4.3.3 Bank-Aware Shortest Job First (BASJF)

To improve upon the WAFCFS scheme, while remaining warp-aware, we propose the Bank-Aware Shortest Job First (BASJF) scheme. This warp-aware scheduler is basically a shortest-job-first (SJF) scheduler that arbitrates between the different warp-groups of memory requests with the aim of minimizing the service time for a warp (i.e., the time it takes to service all requests for the warp). The main distinction between the BASJF policy and a simple SJF policy that just considers the number of requests from each warp is that the BASJF considers the expected service time for the entire warp-group. BASJF uses a scoring system (described in detail in the following section) that accounts for the locality and bank-level parallelism of the requests in the group (besides the total number of requests), the state of the DRAM banks and bank groups, and the occupancy of each of the bank-level command queues. The scoring system effectively calculates the total service time of each completely formed warp group. The scheduler then picks the warp-group with the lowest total score each cycle and enqueues its requests in the bank queues (more details in Section 4.3.3.2).

4.3.3.1 Scoring System

The aim of the scoring system is to estimate the expected overall service time for a warp's memory request. Clearly, this is dependent on the number of DRAM requests issued by the warp as well as the latency of the request that would finish last. The latency of a request will depend on the occupancy of the bank queue and whether it would require a new row to be activated. The first factor contributes to the queuing delay of the request and the second to the core delay [27, 35]. The scoring system accounts for both.

Servicing a row-miss incurs a delay of 36 ns compared to 12 ns for a row-hit. The scoring system thus assigns a cost of 1 unit to a row-hit and 3 to a row-miss. To determine the hit or miss status of a request, the scoring system first ascertains what would be the open row in the target DRAM bank when the request under consideration gets to the front of the bank-level command queue or in other words, the row address of the request that gets serviced immediately before this request. Thus if a request is the first request in its group, it checks the row-address of the last request in the corresponding bank queue. Subsequent requests from a group also need to check the row address of the last preceding request from the same group that is directed at its bank. At the end of this exercise, each request has a score of either 1 or 3.

In addition to assigning the core delay score, the scoring system also adds the total score of all the pending requests in the target bank queue to each memory request. This accounts for the queueing delay of the request. The final score for the warp is the maximum score assigned to its requests. The scoring system thus accounts for the row hit/miss status of a request as well as the queuing delay resulting from the state of the bank queues. This is more accurate than a SJF policy based on counting the number of requests or the number of row hits or misses in predicting the completion time of the warp-group.

4.3.3.2 BASJF Transaction Scheduling Policy

Every cycle, the transaction scheduling policy looks at the completed warp-groups to pick a warp using the scoring system above and then issues a request from the warp with the smallest score to the appropriate bank queue. In the case of a tie, the warp with the highest number of row-hits is picked since row-hits help minimize DRAM power consumption. By scheduling from a warp-group together, BASJF achieves warp-awareness, and at the same time, the bank-aware scoring system allows high utilization of the banks. This effectively leads to higher bandwidth utilization and lower average DRAM access latencies compared to WAFCFS.

Note that once a warp is selected, its requests will take several cycles to be enqueued in the bank queues. It is quite possible that in the meantime, another warp group shows up with fewer requests or has a lower overall score. For example, when the transaction scheduler is in the process of sending out the 4 requests from a warp, a warp with a single request shows up at the memory controller. At this point, the transaction scheduler compares the score of the in-service warp to the score of the new warp, and if it is sufficiently lower (a predefined threshold), then it suspends the servicing of the in-service warp and issues the requests from the new warp. To reduce complexity, the transaction scheduler can override its current decision a maximum of 4 times in succession. After that, it has to drain all the previously selected warps' requests before it can move on to a new warp.

4.3.4 Scheduling for Multiple Memory Controllers

The main drawback of the BASJF policy as described above is that it does not promote any implicit management of request scheduling across different memory controllers. Warp-groups corresponding to the same warp in different memory channels will have different scores and will be positioned differently in the scoring charts. Thus, memory requests from a warp to different memory channels will see varying completion times. The different memory controllers therefore need to coordinate the scheduling of warp-groups to reduce latency-divergence.

The most obvious architecture for such coordinate scheduling will entail some manner of information exchange between the different memory controller. In fact, some previous memory management schemes [13], [40] in the CPU space have looked at exchanging scheduling information and even at the movement of data pages between memory controllers. These techniques, however, are designed to transfer information periodically after long time epochs. The overheads and complexity of an explicit communication mechanism for coordinating between memory controllers for fine-grained request scheduling are too high. Such a coordination scheme will need to exchange informations over the network-on-chip between the memory controllers and possibly a centralized arbiter that enforces coordinated scheduling.

To avoid such complexity, we try to modify the BASJF scheduler so that there is some implicit coordination between the different memory controllers. Since different requests from a warp arrive at the different memory controller within a short window of time, it is conceiveable that a transaction scheduler that is cognizant of the arrival order of warp-groups (similar to the WAFCFS scheme) will be implicitly coordinated with other schedulers in servicing requests from the same warp. Thus to improve warp-awareness of the aspect of the BASJF scheduler, we incorporate a simple age-bias in the scheduling policy. This leads to the BASJF+AB (BASJF with Age Bias) scheduling policy. This is an addition to the BASJF scheme, i.e., it uses the same bank-aware scoring scheme and selects the warp-group with the lowest score in the common case. However, if a warp-group's age goes past an age threshold (K), then the BASJF+AB scheduler picks the oldest warp. By bounding the latest service time for a warp, the different memory controllers are implicitly coordinated in their scheduling of a warp's requests. The value of the age threshold is set statically and is the same across all the channels, thus ensuring an upper bound on the interchannel latency variation for a warp.

In Section 4.5.1, we examine the efficacy of the BASJF+AB scheme against an idealized, explicitly coordinated warp-aware scheduler.

4.3.5 Improving the Performance of BASJF+AB

BASJF+AB will differ significantly from the BASJF policy in its scheduling decision when there are warp-groups with high scores whose age exceed the threshold K. With such a policy, it is possible that the scheduler frequently selects older warp-groups that take a long time to finish primarily because they have long latency row-miss requests. This will negatively impact the bandwidth utilization and thus runs the risk of negating the expected benefits of warp-aware scheduling. Even without the age-bias, warp-groups with row-misses will need to be scheduled to prevent starvation.

The negative impact of a row-miss request in one bank can be alleviated if the row-miss could be coscheduled with row-hit requests in other banks—effectively the overhead of precharging and activating a bank can be hidden by the data transfer of row-hit requests in other banks. The transaction scheduler needs to be cognizant of the timing parameters of the GDDR5 DRAM memory to estimate the minimum number of row-hits it needs to schedule to different banks before it can schedule a row-miss request from a warp-group that has crossed the age-threshold. This leads to bandwidth utilization that is very close to what is achieved by the BASJF scheduler, but at the same time, it allows time-bounded servicing of warp-groups with row-misses like in the BASJF+AB system. Central to this scheme is a metric called the Minimum Efficient Row Burst (*MERB*) that can be precomputed based on the GDDR5 timing numbers and be used by the transaction scheduler to decide how many row-hit requests need to be scheduled at a minimum before issuing row-miss requests from an older warp.

4.3.5.1 Estimating MERB

The sequence of events that take place to transfer a sequence of cache-lines in a DRAM bank are as follows. First, an activate command command with the row and bank addresses is issued to bring data from the DRAM arrays to the bank-level sense-amplifiers (also called the row-buffer). tRCD time after the ACT command, a column-read command can be issued which starts sending the data from the row-buffer to the output pins. tCAS time after the issuance of RD, data are found on the data pins, and it takes tBurst number of cycles to complete the transfer of once cache-line. The bank can be precharged (i.e., made ready for another activate) after tRC time has elapsed since the ACT command by issuing a precharge command. Other RD commands can be issued to the activated row before precharging. Also, there needs to be a gap of at least tRTP between a RD and a subsequent PRE command.

Now, each GDDR5 channel is 64-bits wide (with 2 x32 GDDR5 chips on the channel); thus, 4 DRAM command clock cycles (8 DRAM data clock cycles) are needed to transfer one 128B cache-line.

First, consider that the scheduler has a single bank of GDDR5 to schedule requests to. With 6 or less RDs, the time period is tRC ($tRCD + tCAS + 6 * tBURST = tRC$) and is more otherwise.

Thus if n, the number of row-hit requests issued per bank, is less than 6, then the bandwidth utilization is given by

$$utilization = \frac{tBurst * n}{tRC} \tag{4.1}$$

On the other hand, with 7 or more column reads, the utilization is

$$utilization = \frac{tBurst * n}{tRCD + tBurst * n + (tRTP - tBurst) + tRP} \tag{4.2}$$

Substituting values for GDDR5, the utilization numbers are given by the following equations.

If n is less than or equal to 6, utilization is given by

$$utilization = \frac{4 * n}{60} \tag{4.3}$$

and if n is more than 6, utilization is given by

$$utilization = \frac{4 * n}{4 * n + 34} \tag{4.4}$$

With reads from two different banks, reads to one bank can overlap the tRCD,tRTP and tRP stalls required in the other. If there were 9 bursts per row-bank, then the utilization can reach close to 100%. With 4 or 8 banks, only 4 or 2 bursts per row-bank are required to reach close to maximum bandwidth utilization. The row-to-row activation delay (tRRD) constraint prevents reaching maximum utilization with only a single row-burst to a bank.

The minimum number of row-bursts required per bank while rotating between different banks to achieve high bandwidth utilization can be used by the controller to estimate when it is safe to schedule a row-miss request. This enables the scheduler to service the row-miss request with low queueing delay and still allows the bus utilization to remain high.

The MERB scheduler primarily picks warps that are bank-friendly, similar to the BASJF scheduler. However, in contrast to the BASJF+AB scheduler, it opportunistically schedules row-miss requests using the MERB metric, often before the warp with the row-miss request has passed the age threshold. If the total number of row-hit requests across the banks is enough to hide the latency of a row-miss request, the scheduler schedules the oldest warp with a row-miss request. In essence, the MERB scheduler tries to alleviate the negative impact that the BASJF+AB scheduler has on bandwidth utilization.

4.3.6 Warp-Aware Write Draining (WAWD)

Writes to the DRAM are typically off the critical path. As a result, they are buffered in write queues. When the write-queue occupancy goes above a high-water-mark, the writes are drained from the write queue until the queue occupancy drops below a low-water-mark. At this time, reads are typically not scheduled because switching from a write to a read incurs idle cycles on the data bus due to the write-to-read turnaround delay (tWTR) [35]. Thus, reads can be stalled for a long time by the write-drain mechanism. All the warp-aware memory scheduling mechanisms described above use a modified write-drain mechanism that is warp-aware. Before starting the write-drain mechanism, the read queue is checked and the scheduler continues to issue reads if

- the reads are from warp-groups with a single request

- the reads are from warp-groups a subset of whose requests have been sent to the bank queues.

In effect, this ensures that the write-drain mechanism does not unduly stall some warps. Since this optimization is applied to every warp-aware scheduling scheme, it does little to improve the relative performance of these schemes. However, this helps improve performance over a naive, non-warp-aware draining scheme used in the baseline. With warp-group preemption, it is possible that up to 4 different warp-groups are in the transaction scheduler stage in the memory controller. The WAWD scheme first dequeues all requests from these warp-groups to the bank queues and then scans the batching unit for completed warps with single requests. Even if the warp-group has high scores (i.e., a large expected completion time), the WAWD scheme issues the warp ahead of the write-drain. The WAWD scheduler is triggered once the write queue occupancy gets close to the high water mark and we empirically observed that triggering the WAWD mechanism at (*high water mark* - 8) provides the best performance.

4.3.7 Hardware Overhead

To identify the source warp of a request, each request needs to be tagged with a warp-ID by the source SM. The 5 bits required to uniquely identify a warp will not be a significant increase in the size of a request packet, which already contains a SM identifier, a request ID (8 bits to allow each SM to have 256 outstanding request), the command type, and the address. Depending on the flit size, this might not require additional links in the interconnect. Previous studies have shown that the interconnect delay does not have a big impact on the round trip memory latency [77], [78], so a slight increase in packet size is not likely to impact performance.

The batch-formation stage needs a table to store the warp-IDs of the pending requests in the queue and pointers to the requests belonging to each queue. With each request potentially belonging to a different warp, a 64-entry CAM table indexed by the warp-ID is required in the batching unit of each controller. The small CAM table is sufficient because we track the requests only during the batching stage and thus the time window is relatively small. This means that we do not need to provision the tracking structure for every warp in the system. Once a batch is completely formed, it is not tracked in the CAM. Each entry in this table is a pointer to a request belonging to that warp in the Read Queue. Since each warp may have 32 different requests at the same time and each pointer will be 6 bits wide, this requires a total storage of $64 * 32 * 6$ bits, or 1.5KB. A timestamp when the last observed request for a warp arrived at the memory controller also needs to be recorded in this table. The warp-group is marked complete by the batch scheduler once sufficient time has passed since the arrival of the last request and is marked ready for the transaction scheduler to consider. We maintain a 64-bit value for the timestamp which is updated with the arrival timestamp of the last request for that warp-ID. In addition, each warp's score may be stored in the same table and we limit this to 8 bits per warp-group. Thus, the final storage overhead is 2.06KB. In addition, to estimate the score of a warp-group, each bank queue needs to maintain information about the total score of the requests in the queue and the row-address of the last entry in the queue. This again is little overhead to maintain and is similar to what a FR-FCFS scheduler would need in the baseline.

4.3.8 Summary of Proposed Schemes

The proposed schemes thus try to reduce the average effective memory latency for a warp (i.e., latency of the last request from the warp). A batching unit creates batches of requests for each warp (warp-groups) and the transaction scheduler picks different warps to schedule. With the exception of the WAFCFS scheme, the rest of the schemes progressively add features on top of the preceding one. First, the BASJF scheme tries to estimate the relative cost of servicing different warp-groups and prioritizes the shortest job. The BASJF+AB scheme introduces an age-bias to the BASJF scheme that also allows older warp-groups to be sometimes prioritized over shorter warp groups. The age bias effectively acts as an implicit communication mechanism between different memory controllers. The MERB scheme is an addition to the BASJF+AB scheduler. It carefully orchestrates the scheduling to ensure that older long running warp-groups (due to row-misses) are scheduled in conjunction with smaller, row-hit warp-groups in other banks to effectively use the bandwidth. This helps address the bandwidth inefficiencies introduced by the BASJF+AB scheme. Finally, the WAWD scheme adds warp-aware write draining to the MERB scheme and tries to eliminate situations where a subset of requests from a warp-group are orphaned by the write-drain mechanism. This reduces the write-induced stall time for warp-groups with only one or few requests.

4.4 Methodology

We use GPGPU-Sim version 3.1.2 [78], [79] from the University of British Columbia for our experiments. We model a GPU very similar to NVIDIA's GTX-480 GPU. The simulator has been verified against real hardware and reported to be within 3% accuracy [80]. The salient features of the GPU are listed in Table 4.1. We integrate the DRAM timing model from the USIMM [20] DRAM simulator after modifying USIMM to model GDDR5 timing and the memory controller model proposed in 4.3.1. We model a Hynix 1Gb GDDR5 DRAM part [76], and the timing constraints that were modeled are listed in Table 4.1.

To evaluate our proposals, we use benchmarks from Parboil [81], Rodinia [82], Mars [83], and Lonestar [37] suites. The complete list of benchmarks that we con-

Table 4.1. Simulation Parameters

GPU System Configuration	
No. of Compute Units	30
Warp Size	32
Max Threads/Core	1024
L1 cache/Core	32KB, 128B cache-line size 8-way assoc. LRU
Number of DRAM channels	8
L2 cache/Memory partition	128KB, 128B line-size 16-way assoc, LRU
DRAM device	Hynix GDDR5 H5GQ1H24AFR [76]
DRAM Configuration	6 64-bit Channels 2 x32 Chips/Channel 16 Banks/Chip 4 Banks/Bank Group
GDDR5 Pin Bandwidth	6.0 Gbps
GDDR5 Clk period (tCK)	.667ns
DRAM Read Queue	64 entries per controller
DRAM Write Queue High/Low Watermarks	64 entries per controller 32/16

GDDR5 Timing Parameters	tRC=40ns, tRCD=12ns, tRP=12ns
	tCAS=12ns, tRAS=28ns, tFAW=23ns
	tWTR=5ns, tFAW=23ns, tWL=4 tCK
	tRAS=28ns, tRTP=2ns, tRTRS=1 tCK
	tCCDL=3 tCK, tCCDS=2 tCK

sidered are shown in Table 4.2. We group these benchmarks into three groups as follows:

- **Type-1 Applications:** Type-1 applications are memory-sensitive and produce more than one uncoalesced memory accesses per load.
- **Type-2 Applications:** These applications are memory sensitive, but have perfectly coalesced memory accesses (i.e., a single request per warp on average).
- **Type-3 Applications:** These are not sensitive to memory performance showing less than 15% improvement with a perfect L1 cache and hence are not affected by our proposals.

We investigate the impact of our schemes on applications belonging to the Type-1 and Type-2 categories. We run each benchmark for 1 billion instructions or to completion, whichever is earlier.

Table 4.2. Workloads

Abbr.	Benchmark	Suite
	Type-1 Applications	
sad	Sum of Absolute Differences	Parboil
spmv	Sparse-Matrix Dense-Vector Multiplication	Parboil
bfs	Breadth-First Search	Rodinia
cfd	CFD Solver	Rodinia
kmeans	K-Means Clustering	Rodinia
nw	Needle-Man Wunsch	Rodinia
PVC	PageViewCount	MapReduce
SS	SimilarityScore	MapReduce
bh	Barnes-Hut	LonestarGPU
sp	Survey Propagation	LonestarGPU
sssp	Single-Source Shortest Paths	LoneStarGPU
	Type-2 Applications	
stream	Streamcluster	Rodinia
srad2	SRAD2	Rodinia
bp	Backpropagation	Rodinia
hotspot	HotSpot	Rodinia
invindex	InvertedIndex	MapReduce
PVR	PageViewRank	MapReduce
	Type-3 Applications	
pf	Particlefilter	Rodinia
lukcyt	Leukocyte	Rodinia
lud	LU Decomposition	Rodinia
mm	Matrix Multiplication	Rodinia
hw	Heartwall	Rodinia

4.5 Evaluation

We evaluate the impact of our proposed scheduling schemes on Type-1 and Type-2 applications. We compare the following five schedulers against the the baseline throughput optimized scheduler:

- Warp-Aware FCFS (**WF** in the figures)
- Bank-Aware Shortest-Job-First (**B** in the figures)
- Bank-Aware Shortest-Job-First with Age-Bias (**B+A**)
- Minimum Efficient Row Burst (**B+A+M**)
- Warp-Aware Write-Drain (**B+A+M+W**)

4.5.1 Impact on Type-1 Applications

In this section, we look at the performance impact of the four different scheduling schemes discussed in Section 4.3.

Fig. 4.4 and Fig. 4.5, respectively, show the IPC improvement and average effective DRAM latency experienced by warps with the different scheduling schemes. The results for the simulations have been normalized to a baseline FR-FCFS policy.

4.5.1.1 WAFCFS

We see that the WAFCFS scheme (the **WF** bar in the figures) is the worst performing across the board, with an average performance degradation of 11.2% over the baseline. This is still better than a naive FCFS scheme which does not form warp-groups by about 15%. By grouping requests from a warp together, the WAFCFS scheme sees higher row-hit rates than FCFS and also manages to reduce the effective memory latency of a warp over FCFS by eliminating interwarp interference. However, grouping warps alone does not recover all the row locality that an out-of-order

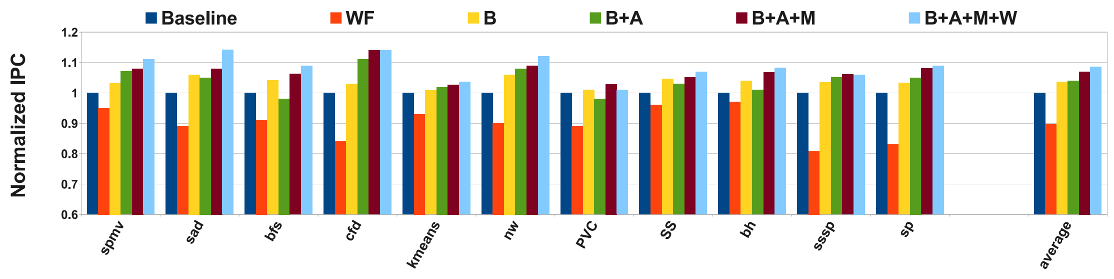


Figure 4.4. Performance normalized to FR-FCFS baseline

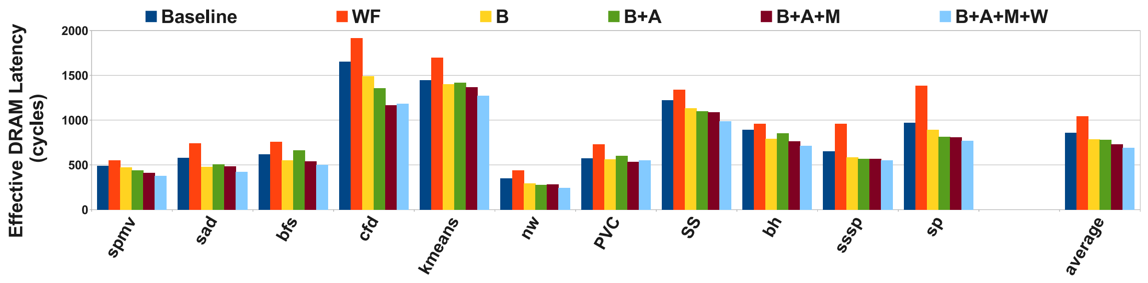


Figure 4.5. Effective main memory latency normalized to FR-FCFS baseline

FR-FCFS scheduler would be able to exploit, and this leads to the overall degradation in performance.

4.5.1.2 BASJF

The BASJF scheme (the **B** bar in the figures) provides an improvement of 3.4% over the baseline. This improvement is due to a 9.1% reduction in effective memory latency experienced by the warps. The reduction in latency comes from the carefully orchestrated scheduling of warp-groups by BASJF that allows requests from a warp to finish together. The BASJF scoring system prioritizes warp-groups with row-hits over row-miss requests and as a result, its row-hit rate is very similar to the baseline FR-FCFS case. Consequently, BASJF allows the GDDR5 system to maintain bandwidth utilization that is only 1.3% lower than the baseline. BASJF benefits are most pronounced for applications like *sad, nw*, and *SS* with these benchmarks showing up to 6.5% improvement. Warps from these applications generate several uncoalesced requests, but each warp accesses only one memory controller in the common case, which allows the BASJF scheduler to optimize the access stream. On the other hand, applications such as *spmv, sp*, and *sssp* show relatively lower benefits because each warp in these applications accesses multiple memory controllers in the common case and the BASJF scheduler has no form of coordination between the controllers to reduce the intrawarp, intercontroller latency variation.

4.5.1.3 BASJF+AB

The BASJF+AB scheme's (**B+A** in the figures) primary goal is to implicitly coordinate the servicing of warp-groups in different controllers. This is done with the age-bias which forces the controller to pick warp-groups that have been stalled in the transaction queue for a long time. With the age-bias set to the same value in all controllers, each memory controller picks warp-groups belonging to the same warp very often. As expected, BASJF+AB outperforms BASJF for some applications whose warps generate requests to multiple memory controllers. Applications cfd, spmv, sssp and sp whose warps touch 3.2 memory controllers on average show performance improvements up to 9% over the baseline. However, unlike BASJF, BASJF+AB also has the potential to degrade performance. Applications such as PVC and bfs, which are bandwidth sensitive and frequently require the involvement of only a single memory channel for their warp's data requests, suffer from the increased row-miss rate introduced by BASJF+AB. By enforcing the age-bias, BASJF+AB often prioritizes row-misses over row-hits. Applications with a limited memory controller spread do not enjoy any of the benefits of the BASJF+AB and instead suffer from the reduced bandwidth utilization of BASJF+AB. Fig. 4.6 shows that for PVC and bfs, the BASJF+AB scheme reduces the effective bandwidth utilization by 13%, and 8%, respectively, leading to performance degradation. In fact, BASJF+AB reduces the bandwidth utilization for all benchmarks compared to FR-FCFS. This is expected because FR-FCFS is designed to maximize bandwidth utilization by prioritizing row-hits over row-misses. However, for applications like *cfd, sp, sssp*, and *nw*, the effect of the reduced bandwidth utilization is outweighed by the effect of the reduced effective DRAM latency, which results from the elimination of intrawarp latency divergence. As a result of the interplay of these opposing factors, BASJF+AB shows improvements up to 10.8% and degradations up to 2.9%. This leads to an average improvement of only 4.0% with the BASJF+AB scheme.

To estimate the effectiveness of the age-bias in coordinating between different memory-controllers, we compare the BASJF+AB scheme against a hypothetical, multimemory-controller-aware scheme that we call *Ideal Coordination* (results shown in Fig. 4.7). In this scheme, whenever a controller picks a warp-group to service, it sends an *instantaneous message* to the other memory controllers with the currently selected warp-ID. The other memory-controllers prioritize the requests from the specified warp. This scheme thus abstracts away the complexities of explicit communication between controllers. We see that the BASJF+AB scheme closely follows

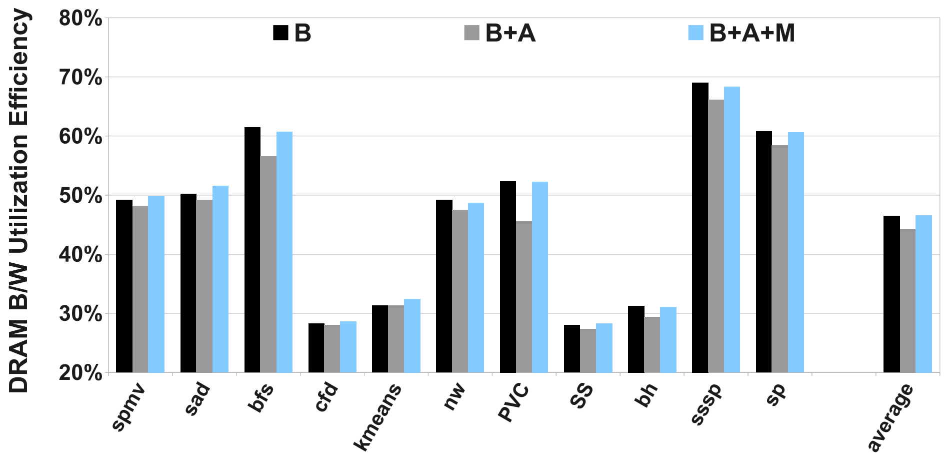


Figure 4.6. Bandwidth Utilization Compared to FR-FCFS

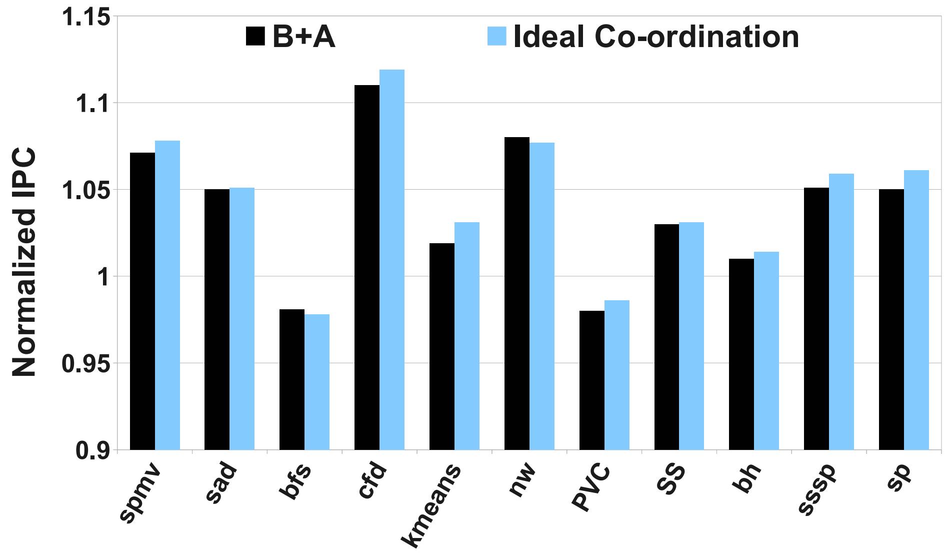


Figure 4.7. Effectiveness of Age-Bias

the Ideal Coordination scheme in terms of overall performance, which demonstrates that the age-bias can accomplish a degree of intermemory-controller coordination implicitly. The difference is more pronounced in the case of applications where many memory-controllers are involved in satisfying the request of a single warp (e.g., *cfd*, *kmeans*, and *sp*). Overall the Ideal Coordination scheme differs from the BASJF+AB scheme by 1.1%.

4.5.1.4 MERB

The MERB scheduler (**B+A+M**) tries to preserve the benefits of BASJF+AB, namely the implicit coordination between different memory controllers and, at the same time, intends to provide high bandwidth. This is achieved by allowing warps with row-misses to be scheduled as soon as enough requests are found to other DRAM banks with row-hits. This allows the row-hits in the other banks to hide the overheads involved with precharging the bank and opening a new row. The MERB scheduler thus improves the bandwidth utilization of the BASJF+AB scheme by hiding the row-miss overhead with overlapped row-hits. Fig. 4.6 shows the effect of MERB on the effective bandwidth utilization. By allowing row-misses to occur in a completely (or sufficiently) overlapped manner with the row-hits in other banks, MERB can not only recover the bus efficiency lost by BASJF+AB, it can improve the utilization over a baseline FR-FCFS in some cases. The effective main memory latency is lowered by 16.4% compared to the baseline (an improvement of 7% over the BASJF+AB scheme). Combined with the near-optimal bandwidth utilization, this leads to an increase of performance by 6.9% over the baseline. Applications like *PVC, bfs*, and *SS* which had suffered from the increased row-miss rates of BASJF+AB now show improvements of up to 6.3% over the baseline.

4.5.1.5 WAWD

The WAWD mechanism (**B+A+M+W** in the figures) represents the warp-aware write management scheme when it is applied to the final warp-aware memory scheduler (MERB). The WAWD mechanism ensures that long latency write-drains do not stall small warp-groups (containing one request) or warp-groups which have already received some service from the memory controller. Write drains have been shown to be detrimental to CPU performance [16], [35], and we see in Fig. 4.8 that the write traffic in GPUs constitutes a larger fraction of the memory traffic than what is conventionally found in CPUs. Fig. 4.4 shows that the WAWD scheduling increases the performance by an average of 8.9% over the baseline. Larger improvements are seen in cases where the write-to-read ratio is high, such as *sad*, and also in applications which generate few memory requests per warp, such as *nw*.

4.5.2 Impact on Type-2 Applications

Type-2 applications have structured and regular data accesses, exhibit high spatial locality, and are bandwidth-bound in many cases. We found that there is a modest

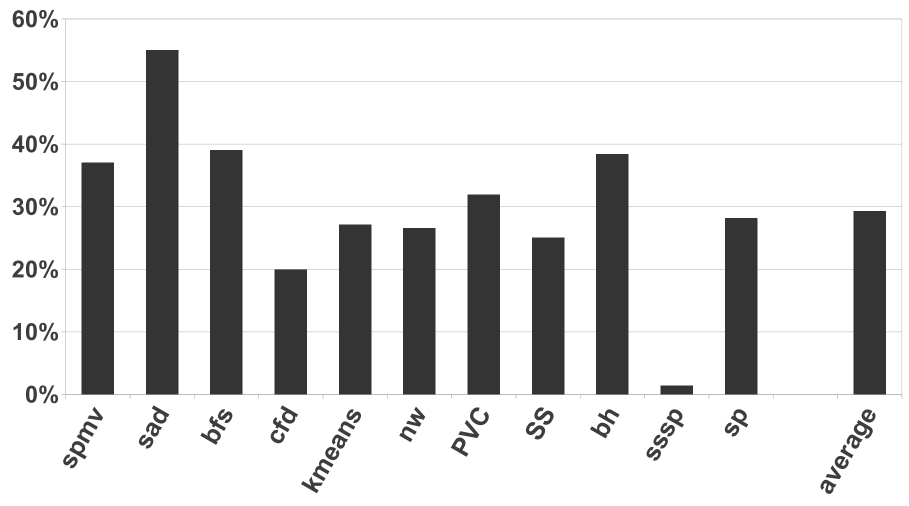


Figure 4.8. Write Intensity

1.4% performance improvement on average with the MERB+WAWD scheme over the baseline throughput-optimized scheduler (Fig. 4.9). None of the applications show any performance degradation. This is expected because all the warp-aware scheduling schemes, with the exception of WAFCFS, behave similar to the baseline throughput-optimized scheduler for regular applications. For example, in BASJF (and MERB), when there is only one memory request per warp, the scoring system will always allow older row-hits to be prioritized over row-misses and younger row-hits and thus will behave similar to the FR-FCFS policy. The performance gains with MERB+WAWD can be attributed to two factors. First, MERB handles row-miss requests more efficiently than the baseline; this is corroborated by the somewhat improved bandwidth utilization in a few benchmarks (*stream, PVR, hotspot*). Also with WAWD, the delaying of the write-drains helps a few pending reads and the waiting warps for the additional performance improvement. This demonstrates that when applications are regular and have high degree of coalescing, the MERB+WAWD scheduler is marginally better than the bandwidth-optimizing baseline scheduler. Since traditional graphics workloads are latency-insensitive and bandwidth-intensive, they would behave similar to the Type-2 workloads.

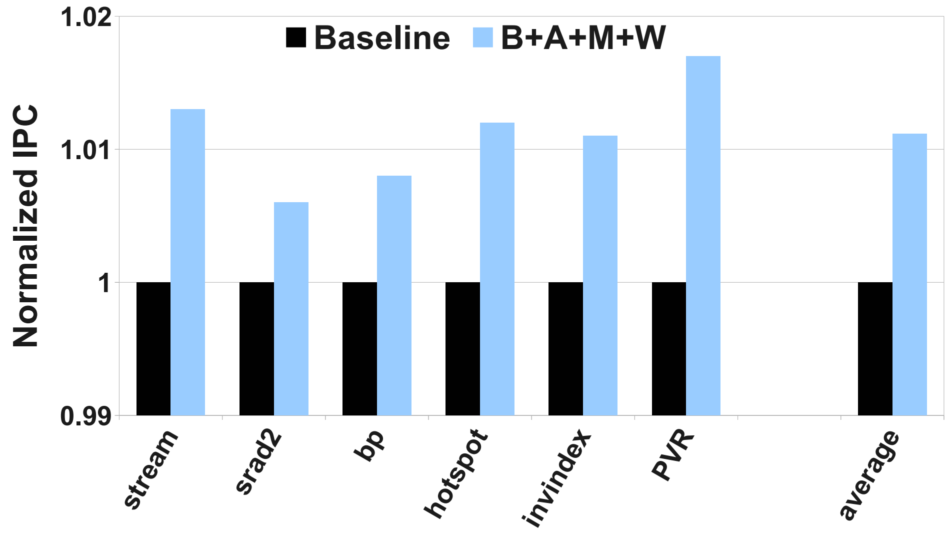


Figure 4.9. Performance Impact on Type-2 Applications

4.5.3 Comparison with Single-Bank Warp-Aware Scheduling

Lakshminarayana *et al.* were the first to propose a warp-aware memory scheduling policy in [84]. Their strategy comprises the formation of queues of requests from each warp at the memory controller and on every cycle, using an evaluation function [85] to decide between issuing a row-hit request and a request belonging to the warp-group that was recently serviced and is also the shortest queue (in terms of requests remaining in the warp-group).

The algorithm in [84] uses a potential function that, by default, aims to maximize the row-hit rate. A parameter α is used to bias the potential function towards preferring a request from the smallest remaining warp-group. This parameter has to be determined empirically and set statically for each program. In contrast, our batch-formation and scheduling schemes (BASJF and BASJF+AB) determine the completion time of a warp-group and schedules the one with the *shortest completion time* and not from the warp-group that has the fewest requests. When a selected warp-group has row-miss requests, the MERB scheduler finds the scheduling slot when the row-miss can be overlapped efficiently with row-hits in other banks. Thus, our method is more general and does not require profiling of applications to figure out the right balance between warp-aware scheduling and FR-FCFS. Also, the algorithm in [84] applies to within a bank. As we have shown in Section 4.5, it is important to coordinate the scheduling of requests across banks and even multiple channels. Our schemes determine the completion time of warp-groups with information from every bank-queue. In [84], writes are interleaved with reads in the baseline as well as in the proposed schemes. We assume a more conventional, and higher performance baseline write scheduling mechanism and we incorporate warp-awareness in the write-drain mechanism to avoid starving critical, orphaned reads. Finally, the potential function in [84] requires a combination of complex calculations. The BASJF scheme and its derivatives require simple addition and comparison operations to select a warp-group.

We compare the MERB scheme with a variant of the α-SJF scheme [84], which we call the Single-Bank Warp-Aware Scheduler (SBWAS). In SBWAS, at each cycle, the transaction scheduler schedules either a row-hit request to a bank or a request from the warp-group with the fewest requests for that specific bank (there is no global bank information) and rotates over banks in round-robin fashion. The row-hit request is picked with a probability of α. For each benchmark, we determined the value of α by profiling (possible values being 0.25, 0.5, and 0.75). We found that on average, SBWAS provides an improvement of 2.51 % compared to the baseline warp-unaware scheduler. The applications which generate requests to multiple banks and controllers (e.g., *spmv, sp, ssp, cfd*) show little improvement with SBWAS. *bfs* shows the most improvement with SBWAS as it touches fewer banks than other benchmarks (3.8%). On the other hand, although the application *sad* generally touches one or two banks per warp, the high number of writes erode the benefits of SBWAS when interleaved with reads.

4.6 Related Work

4.6.1 Memory Scheduling

A large body of work has looked at memory scheduling techniques for multicore systems [10], [12]–[14], [34], [35].

4.6.2 GPU Memory Scheduling

The only paper that explores the benefits of warp-aware scheduling is by Lakshminarayana *et al.* [84], which we discuss in detail in Section 4.5.3. Staged-Memory-Scheduling [15] aims to improve the bandwidth utilization of the DRAM channel in a heterogeneous CPU+GPU system by forming batches of row-hit requests from each source and then arbitrating between these requests. Jeong *et al.* propose a QoS aware policy that allows the GPU to consume only the bandwidth that is absolutely necessary to maintain a certain QoS [86] and prioritize CPU requests to provide the latency sensitive CPUs with low latency. Yuan *et al.* [77] order requests from a SM at the interconnect to harvest intrawarp locality with a simple FCFS scheduler by eliminating interwarp request interleaving. However, none of the proposed techniques have looked at the importance of incorporating warp-level ideas to reduce memory divergence.

4.6.3 PAR-BS

The PAR-BS scheme [34] forms batches of requests in a CPU's memory controller and issues requests from a batch to the memory system. The express motivation behind the batch-formation is fairness and as a result, a batch in PAR-BS will include requests from many threads and have different batches for different banks. Our batching scheme does exactly the opposite and groups requests from a warp together to reduce the latency divergence of a warp. In addition, we arbitrate between batches based on a bank-aware shortest job first policy to reduce wait time for warps, which is different from PAR-BS, which uses a MLP-based SJF policy for thread priorities.

4.6.4 ATLAS

The ATLAS scheduler [13] was proposed to promote fair-scheduling across channels in a multimemory-controller CPU-chip. ATLAS uses the Least-Attained-Service metric to rank thread priorities and a long time quanta after which different memory controllers exchange information to update thread ranks for the next execution epoch. Threads which received lower service in the previous quanta are prioritized over others. The main impediment to implementing an ATLAS-like scheduler in SIMT context is the need for intermemory-controller information exchange at a fine time granularity. The ATLAS scheme uses long time quantas for scalability, whereas we need memory-controllers to coordinate at the granularity of warps. We also show that a strong age-bias can effectively achieve this coordination without explicit communication. In addition, none of the schemes described above mitigate the impact of the write-drain policy. Our warp-aware write-drain scheme provides significant benefits in many benchmarks.

4.6.5 Memory Divergence Mitigation in GPUs

Instead of utilizing warp-level multithreading to hide the memory divergence latency, Meng *et al.* [18] advocate intrawarp latency hiding. This is accomplished through dynamic warp subdivision, a technique that allows some threads in a warp to make progress while the others are stalled on memory accesses [87]. This requires a single warp to be able to occupy multiple slots in the warp-scheduler and thus incurs at least double the cost and complexity in the scheduling hardware in each core. Several software optimizations have been proposed to tackle memory divergence. These include data herding [19] to force all threads in a warp to load from the same memory block through a compiler framework, a runtime system that tries to optimize the memory layout to reduce memory divergence [88] as well as techniques to improve memory coalescing [89]. Recently other techniques have been proposed to reduce effective memory latency [72], [90], [91].

4.7 Conclusions

Memory divergence is a complex problem which is affected by the nature of data parallel applications and by the implementations of the compilers, libraries, and the runtime system. We show that existing GPUs can be made more efficient by incorporating warp-awareness in the DRAM scheduling policy. We demonstrate novel techniques that can be implemented in state-of-the-art schedulers to reduce the interwarp interference leading to lower effective DRAM stall times for warps. The proposed scheduler (MERB) can reduce the latency of stalled warps while maintaining good bandwidth utilization. Our best-performing scheduler boosts average performance by 8.6% over a throughput-optimized baseline.

CHAPTER 5

USIMM: A SIMULATION FRAMEWORK FOR MAIN MEMORY

The USIMM DRAM simulator was released in 2012 as the simulation infrastructure for the Memory Scheduling Championship held with ISCA-2012. USIMM is a detailed main memory timing simulator. In its native form, USIMM uses a trace-based input and a simple out-of-order processor model. However, the modular design has allowed it to be integrated with other execution driven full-system simulators such as SIMICS [48] and GPGPU-Sim [78]. In this chapter, we describe the simulation infrastructure in detail: the software architecture of the simulator, the verification methodology, and the simulator accuracy.

5.1 Simulator Design

5.1.1 High-Level Overview

This section provides a detailed description of the USIMM code. USIMM has the following high-level flow. A front-end consumes traces of workloads and models a reorder buffer (ROB) for each core on the processor. Memory accesses within each ROB window are placed in read and write queues at the memory controller at each channel. Every cycle, the simulator examines each entry in the read and write queues to determine the list of operations that can issue in the next cycle. A scheduler function is then invoked to pick a command for each channel from among this list of candidate commands. This scheduler function is the heart of the USIMM simulator and can be customized easily to implement different schedulers. The underlying USIMM code is responsible for modeling all the correctness features: DRAM states, DRAM timing parameters, and models for performance and power. The scheduler must only worry about performance/power features: heuristics to select commands every cycle such that performance, power, and fairness metrics are optimized. Once the scheduler selects these commands, USIMM updates DRAM state and marks instruction completion times so they can be eventually retired from the ROB.

5.1.2 Code Files

The code is organized into the following files:

- **main.c :** Handles the main program loop that retires instructions, fetches new instructions from the input traces, and calls update_memory( ). Also calls functions to print various statistics.
- **memory_controller.c :** Implements update_memory( ), a function that checks DRAM timing parameters to determine which commands can issue in this cycle. Also has functions to calculate power.
- **scheduler.c :** Function provided by the user to select a command for each channel in every memory cycle.
- **configfile.h memory_controller.h params.h processor.h scheduler.h utils.h utlist.h :** various header files.

5.1.3 Inputs

The main( ) function in file main.c interprets the input arguments and initializes various data structures. The memory system and processor parameters are derived from a configuration file, specified as the first argument to the program. Each subsequent argument represents an input trace file. Each such trace is assumed to run on its own processor core.

5.1.4 Simulation Cycle.

The simulator then begins a long while loop that executes until all the input traces have been processed. Each iteration of the while loop represents a new *processor cycle*, possibly advancing the ROB. The default configuration files assume 3.2 GHz processor cores and 800 MHz DRAM channels, so four processor cycles are equivalent to a single memory bus cycle. Memory functions are invoked in processor cycles that are multiples of four.

5.1.5 Commit

The first operation in the while loop is the commit of oldest instructions in the pipelines of each core. Each core maintains a reorder buffer (ROB) of fixed size that houses every in-flight instruction in that core. For each core, the commit operation in a cycle attempts to sequentially retire all completed instructions. Commit is halted in a cycle when the commit width is reached or when an incomplete instruction is encountered. A commit width of 2 per processor cycle corresponds to a core IPC of 2 if the trace was devoid of memory operations. The simulated IPC for most traces will be much less than 2.

5.1.6 Checking for Readiness

The next operation in the while loop is a scan of every memory instruction in the read and write queues of the memory controller to determine what operation can issue in this cycle. A single memory instruction translates into multiple memory system commands (e.g., PRE, ACT, Column-Read). Our scan first computes what the next command should be. Note that this changes from cycle to cycle based on the current row buffer contents, the low-power state, and whether a refresh is being performed. We also examine a number of DRAM timing parameters to determine if the command can issue in this cycle. In addition to examining the read and write queues, we also consider the list of general commands (refresh, power down/up, precharge) and determine if they can be issued.

5.1.7 Scheduling

Once a list of candidate memory commands for this cycle is determined by our scan, a schedule( ) function (in file schedule.c) is invoked. This is the heart of the simulator and the function that must be provided by contestants in the JWAC MSC. In each memory cycle, each memory channel is capable of issuing one command. Out of the candidate memory commands, the schedule function must pick at most one command for each channel. Once a command has been issued, other commands that were deemed "ready for issue in this cycle" to the same channel will be rejected in case the scheduler tries to issue them. While each channel is independently scheduled, some coordination among schedulers may be beneficial [13].

5.1.8 Instruction Completion Times

Once the scheduler selects and issues commands, the simulator updates the state of the banks and appropriately sets the completion time for the selected memory instructions. This eventually influences when the instruction can be retired from the ROB, possibly allowing new instructions to enter the processor pipeline.

5.1.9 Advancing the Trace and Trace Format

Next, new instructions are fetched from the trace file and placed in the ROB. Memory instructions are also placed in the read and write queues. This process continues until either the ROB or write queues are full or the fetch width for the core is exhausted. The trace simply specifies if the next instruction is a memory read (R), memory write (W), or a nonmemory instruction (N). In case of memory reads and writes, a hexadecimal address is also provided in the trace. For the MSC, we assume that a trace can only address a 4 GB physical address space, so the trace is limited to 32-bit addresses. Memory writes do not usually correspond to actual program instructions; they refer to evictions of dirty data from cache. As a simplification, we assume that each line in the trace corresponds to a different program instruction. Note that this is an approximation not just because of cache evictions, but because some x86 instructions correspond to multiple memory operations and the traces will occasionally include memory accesses to fetch instructions (and not just data).

5.1.10 Fetch Constraints and Write Drains

We assume that nonmemory (N) and memory write (W) instructions finish instantaneously, i.e., they are never bottlenecks in the commit process. Memory-writes will hold up the trace only when the write queue is full. To prevent this, it is the responsibility of the scheduler to periodically drain writes. Memory-reads are initially set to complete in the very distant future. The schedule function will later determine the exact completion time and update it in the ROB data structure. We do not model an explicit read queue size. The typical length of the read queue is determined by the number of cores, the size of the ROB, and the percentage of memory reads in a ROB. In other words, we assume that the read queue is not underprovisioned, relative to other processor parameters. The write queue on the other hand does need a capacity limit in our simulator since a write queue entry need not correspond to a ROB entry.

5.1.11 Refresh Handling

The simulator ensures that in every $8 \times tREFI$ window, all DRAM chips on a channel are unavailable for time $8 \times tRFC$, corresponding to eight refresh operations. If the user neglects to issue eight refreshes during the $8 \times tREFI$ time window, USIMM will forcibly issue any remaining refreshes at the end of the time window. During this refresh period, the memory channel is unavailable to issue other commands. Each cycle, the simulator calculates a refresh deadline based on how many refreshes are pending for that window and eventually issues the required number of refreshes at the deadline. In order to ensure that the refresh deadline is not missed, the simulator marks a command ready only if issuing it does not interfere with the refresh deadline. So, when the refresh deadline arrives, the DRAM chip will be inactive (i.e., the banks will be precharged and in steady state or some rows will be open but with no on-going data transfer). The rank may also be in any of the power-down modes, in which case, it will be powered up by the auto refresh mechanism; the user does not need to issue the power-up command explicitly. At the end of the refresh period, all banks are in a precharged, powered-up state.

5.1.12 Implicit Scheduling Constraints

It is worth noting that the simulator design steers the user towards a greedy scheduling algorithm, i.e., the user is informed about what can be done in any given cycle and the user is prompted to pick one of these options. However, as we show in the example below, the user must occasionally not be tempted by the options presented by the simulator. Assume that we are currently servicing writes. A read can only be issued if time $tWTR$ has elapsed since the last write. Hence, following a write, only writes are presented as options to the memory scheduler. If the user schedules one of these writes, the read processing is delayed further. Hence, at some point, the scheduler must refrain from issuing writes so that time $tWTR$ elapses and reads show up in the list of candidate commands in a cycle.

5.1.13 Address Mapping

A cache line is placed entirely in one bank. The next cache line could be placed in the same row, or the next row in the same bank, or the next bank in the same rank, or in the next rank in the same channel, or in the next channel. The data mapping policy determines the extent of parallelism that can be leveraged within the memory system. The MSC focuses on two different processor-memory configurations; each uses a different data mapping policy. The first configuration (*1channel*, with ADDRESS_MAPPING set to 1) tries to maximize row buffer hits and places consecutive cache lines in the same row, i.e., the lower-order bits pick different columns in a given row. The address bits are interpreted as follows, from left (MSB) to right (LSB):

$$1channel\ mapping\ policy :: row : rank : bank : channel : column : blockoffset$$

The second configuration (*4channel*, with ADDRESS_MAPPING set to 0) tries to maximize memory access parallelism by scattering consecutive blocks across channels, ranks, and banks. The address bits are interpreted as follows:

$$4channel\ mapping\ policy :: row : column : rank : bank : channel : blockoffset$$

5.1.14 Example Schedulers

As part of the USIMM distribution, the following set of sample baseline scheduler functions were released. These functions were meant to demonstrate the general programming approaches that could be adapted by the MSC contestants.

5.1.14.1 FCFS, scheduler-fcfs.c

True FCFS, i.e., servicing reads in the exact order that they arrive and stalling all later reads until the first is done, leads to very poor bank-level parallelism and poor bandwidth utilization. We therefore implement the following variant of FCFS. Assuming that the read queue is ordered by request arrival time, our FCFS algorithm simply scans the read queue sequentially until it finds an instruction that can issue in the current cycle. A separate write-queue is maintained. When the write queue size exceeds a high water mark, writes are drained similarly until a low water mark is reached. The scheduler switches back to handling reads at that time. Writes are also drained if there are no pending reads.

5.1.14.2 Credit-Fair, scheduler-creditfair.c

For every channel, this algorithm maintains a set of counters for credits for each thread, which represent that thread's priority for issuing a read on that channel. When scheduling reads, the thread with the most credits is chosen. Reads that will be open row hits get a 50% bonus to their number of credits for that round of arbitration. When a column read command is issued, that thread's total number of credits for using that channel is cut in half. Each cycle all threads gain one credit. Write queue draining happens in an FR-FCFS manner (prioritizing row hits over row misses). The effect of this scheduler is that threads with infrequent DRAM reads will store up their credits for many cycles so they will have priority when they need to use them, even having priority for infrequent bursts of reads. Threads with many, frequent DRAM reads will fairly share the data bus, giving some priority to open-row hits. Thus, this algorithms tries to capture some of the considerations in the TCM scheduling algorithm [14].

5.1.14.3 Power-Down, scheduler-pwrdn.c

This algorithm issues PWR-DN-FAST commands in every idle cycle. Explicit power-up commands are not required as power-up happens implicitly when another command is issued. No attempt is made to first precharge all banks to enable a deep power-down.

5.1.14.4 Close-Page, scheduler-close.c

This policy is an approximation of a true close-page policy. In every idle cycle, the scheduler issues precharge operations to banks that last serviced a column read/write. Unlike a true close-page policy, the precharge is not issued immediately after the column read/write, and we do not look for potential row buffer hits before closing the row.

5.1.14.5 First-Ready-Round-Robin, scheduler-frrr.c

This scheduler tries to combine the benefits of open row hits with the fairness of a round-robin scheduler. It first tries to issue any open row hits with the "correct" thread-ID (as defined by the current round robin flag), then other row hits, then row misses with the "correct" thread-ID, and then finally, a random request.

5.1.14.6 MLP-aware, scheduler-mlp.c

The scheduler assumes that threads with many outstanding misses (high memory level parallelism, MLP) are not as limited by memory access time. The scheduler therefore prioritizes requests from low-MLP threads over those from high-MLP threads. To support fairness, a request's wait time in the queue is also considered. Writes are handled as in FCFS, with appropriate high and low water marks.

5.2 DRAM Timing Model

In this section, we take a detailed look at the heart of the USIMM DRAM simulator, the part that maintains the DRAM state and decides the "readiness" of the different DRAM commands. First, we discuss the different memory commands that can be issued by the command scheduler.

5.2.1 Memory Commands.

In every cycle, the memory controller can either issue a command that advances the execution of a pending read or write, or a command that manages the general DRAM state. The four commands corresponding to a pending read or write are:

- **PRE:** Precharge the bitlines of a bank so a new row can be read out.
- **ACT:** Activate a new row into the bank's row buffer.
- **COL-RD:** Bring a cache line from the row buffer back to the processor.
- **COL-WR:** Bring a cache line from the processor to the row buffer.

The six general "at-large" commands used to manage general DRAM state and not corresponding to an entry in the read or write queues are:

- **PWR-DN-FAST:** Power-Down-Fast puts a rank in a low-power mode with quick exit times. This command can put the rank into one of two states: active power down or precharge power down (fast). If all the banks in the DRAM chip are precharged when the PWR-DN-FAST command is applied, the chip goes into the precharge power down mode. However, if even a single bank has a row open, the chip transitions into the active power down mode. The power consumption of the active power down mode is higher than that of the precharge power down mode. In both these states, the on-chip DLL is active. This allows the chip to power-up with minimum latency. To ensure transition into the lower power state, it may be necessary to first precharge all banks in the rank (more on this below).

- **PWR-DN-SLOW:** Power-Down-Slow puts a rank in the precharge power down (slow) mode and can only be applied if all the banks are precharged. The DLL is turned off when the slow precharge power-down mode is entered, which leads to higher power savings, but also requires more time to transition into the active state.
- **PWR-UP:** Power-Up brings a rank out of low-power mode. The latency of this command (i.e., the time it takes to transition into the active state) is dependent on the DRAM state when the command is applied (fast or slow exit modes). If the chip is in the active power down mode, it retains the contents of the open row-buffer when the chip is powered up. When the rank is powered down, all pending requests to that rank in the read and write queue note that their next command must be a PWR-UP. Thus, picking an instruction from the read or write queues will automatically take care of the power-up, and an at-large power-up command (similar to a PWR-DN-FAST or PWR-DN-SLOW) is not required. Similarly, refresh operations will automatically handle the exit from the power-down mode.
- **Refresh:** Forces a refresh to multiple rows in all banks on the rank. If a chip is in a power-down mode before the refresh interval, the rank is woken up by refresh.
- **PRE:** Forces a precharge to a bank. This makes the bank ready for future accesses to new rows).
- **PRE-ALL-BANKS:** Forces a precharge to all banks in a rank. This is most useful when preparing a chip for a power-down transition.

5.2.2 Timing Parameters

Table 5.1 shows the timing parameters that are honored by USIMM and Table 5.2 shows the minimum delays that are enforced between successive commands. In response to the above commands, the next state of the DRAM is decided based on the previous state.A subset of commands can be issued to the bank when it is in one of the stable states.

Table 5.1. DRAM timing parameters for default memory system configuration.

Timing parameter	Default value (cycles at 800MHz)	Description
tRCD	11	Row to Column command Delay. Interval between row access and data ready at sense amplifiers.
tRP	11	Row Precharge. The time interval that it takes for a DRAM array to be precharged for another row access.
tCAS	11	Column Access Strobe latency. The time interval between column access command and the start of data return by the DRAM device(s). Also known as tCL.
tRC	39	Row Cycle. The time interval between accesses to different rows in a bank. tRC =tRAS +tRP.
tRAS	28	Row Access Strobe. The time interval between row access command and data restoration in a DRAM array. A DRAM bank cannot be precharged until at least tRAS time after the previous bank activation.
tRRD	5	Row activation to Row activation Delay. The minimum time interval between two row activation commands to the same DRAM device. Limits peak current profile.
tFAW	32	Four (row) bank Activation Window. A rolling time-frame in which a maximum of four-bank activations can be engaged. Limits peak current profile in DDR2 and DDR3 devices with more than 4 banks.
tWR	12	Write Recovery time. The minimum time interval between the end of a write data burst and the start of a precharge command. Allows sense amplifiers to restore data to cells.
tWTR	6	Write To Read delay time. The minimum time interval between the end of a write data burst and the start of a column-read command. Allows I/O gating to overdrive sense amplifiers before read command starts.
tRTP	6	Read to Precharge. The time interval between a read and a precharge command.
tCCD	4	Column-to-Column Delay. The minimum column command timing, determined by internal burst (prefetch) length. Multiple internal bursts are used to form longer burst for column reads. tCCD is 2 beats (1 cycle) for DDR SDRAM, and 4 beats (2 cycles) for DDR2 SDRAM.
tRFC	128	Refresh Cycle time. The time interval between Refresh and Activation commands.
tREFI	6240	Refresh interval period.
tCWD	5	Column Write Delay. The time interval between issuance of the column-write command and placement of data on the data bus by the DRAM controller.
tRTRS	2	Rank-to-rank switching time. Used in DDR and DDR2 SDRAM memory systems; not used in SDRAM or Direct RDRAM memory systems. One full cycle in DDR SDRAM.
tPDMIN	4	Minimum power down duration.
tXP	5	Time to exit fast power down
tXPDLL	20	Time to exit slow power down
tDATATRANS	4	Data transfer time from CPU to memory or vice versa.

Table 5.2. Command timing restrictions

Previous Command	Next Command	Rank	Bank	Minimum Gap
ACT	ACT	same	same	tRC (also tFAW to be considered)
ACT	ACT	same	diff	tRRD (also tFAW to be considered)
ACT	PRE	same	same	tRAS
ACT	COL-RD	same	same	tRCD
ACT	COL-WR	same	same	tRCD
PRE	ACT	same	same	tRP
PRE	Refresh	same	same	tRP
COL-RD	COL-RD	same	any	max(tBURST, tCCD)
COL-RD	COL-RD	diff	any	tBURST + tCCD
COL-RD	COL-WR	any	any	tCAS + tBURST + tRTRS - tCWD
COL-RD	PRE	same	same	tBURST + tRTP - tCCD
COL-WR	COL-RD	same	any	tCWD + tBURST + tWTR
COL-WR	COL-RD	diff	any	tCWD + tBURST + tRTRS - tCAS
COL-WR	COL-WR	same	any	max(tBURST, tCCD)
COL-WR	COL-WR	diff	any	tBURST + tODT
COL-WR	PRE	same	same	tCWD + tBURST + tWR
Refresh	ACT	same	any	tRFC
Refresh	PRE	same	any	tRFC

When a command is issued to a bank, it changes the state of the target bank (or the target rank when the command is Refresh, PWR-UP, PWR-DN-SLOW, PWR-DN-FAST, or PRE-ALL-BANKS). A subset of commands can be issued to the bank when it is in one of the stable states. When a command is issued to a bank, it changes the state of the target bank (or the target rank when the command is Refresh, PWR-UP, PWR-DN-SLOW, PWR-DN-FAST, or PRE-ALL-BANKS). In addition, it determines what is the earliest possible time when another command can be issued to that bank (and also possibly other banks and ranks on that channel). The values in Table 5.1 are typical of many Micron DDR3 chips, with only the tRFC parameter varying as a function of chip capacity. Consider tWTR as an example timing parameter. The direction of the memory channel data bus must be reversed every time the memory system toggles between reads and writes. This introduces timing delays, most notably the delay between a write and read to the same rank (tWTR). To reduce the effect of this delay, multiple writes are typically handled in succession before handling multiple reads in succession. Note that commands are not required to turn the bus direction; if sufficient time has elapsed after a write, a read becomes a candidate for issue.

5.3 DRAM Power Model

The simulator also supports a power model. Relevant memory system statistics are tracked during the simulation, and these are fed to equations based on those in the Micron power calculator [92].

5.3.1 Memory Organizations

The power model first requires us to define the type of memory chip and rank organization being used. The input configuration file specifies the number of channels, ranks, and banks. This organization is used to support a 4 GB address space per core. As more input traces are provided, the number of cores and the total memory capacity grows. Accordingly, we must figure out the memory organization that provides the required capacity with the specified channels and ranks. For example, for the 1channel.cfg configuration and 1 input trace file, we must support a 4 GB address space with 1 channel and 2 ranks. Each rank must support 2 GB, and we choose to do this with 16 x4 1 Gb DRAM chips. If 1channel.cfg is used with 2 input trace files, we support an 8 GB address space with the same configuration by instead using 16 x4 2 Gb DRAM chips. For the MSC, we restrict ourselves to the configurations in Table 5.3. USIMM figures out this configuration based on the input system configuration file and the number of input traces. It then reads the corresponding power and timing parameters for that DRAM chip from the appropriate file in the input/ directory (for example, 1Gb_x4.vi). The only timing parameter that shows variation across DRAM chips is tRFC.

Table 5.3. Different memory configurations in our power model.

System config file	Channels and Ranks per Channel	Number of cores	Memory Capacity	Organization of a rank
1channel.cfg	1 ch, 2 ranks/ch	1	4 GB	16 x4 1 Gb chips
1channel.cfg	1 ch, 2 ranks/ch	2	8 GB	16 x4 2 Gb chips
1channel.cfg	1 ch, 2 ranks/ch	4	16 GB	16 x4 4 Gb chips
4channel.cfg	4 ch, 2 ranks/ch	1	4 GB	4 x16 1 Gb chips
4channel.cfg	4 ch, 2 ranks/ch	2	8 GB	8 x8 1 Gb chips
4channel.cfg	4 ch, 2 ranks/ch	4	16 GB	16 x4 1 Gb chips
4channel.cfg	4 ch, 2 ranks/ch	8	32 GB	16 x4 2 Gb chips
4channel.cfg	4 ch, 2 ranks/ch	16	64 GB	16 x4 4 Gb chips

While the simulator can support more than 16 traces with 4channel.cfg and more than 4 traces with 1channel.cfg, the power model does not currently support models other than those in Table 5.3. The different allowed DRAM chips and the power parameters for each are summarized in Table 5.4. The current and voltage values in Table 5.4 were derived from Micron datasheets [31], [32], [47].

5.3.2 Power Equations

The power equations are as follows and are based on equations in the Micron power calculator [92] and the Micron Memory System Power Technical Note [93]:

$$ReadPower = (I_{DD4R} - I_{DD3n}) * V_{DD} * \%Cycles\, when\, data\, is\, being\, Read \tag{5.1}$$

$$WritePower = (I_{DD4W} - I_{DD3n}) * V_{DD} * \%Cycles\, when\, data\, is\, being\, Written \tag{5.2}$$

$$RefreshPower = (I_{DD5} - I_{DD3n}) * V_{DD} * T_{RFC}/T_{REFI} \tag{5.3}$$

$$ActivatePower = Max.\, Activate\, Power * T_{RC}/(Average\, gap\, between\, ACTs) \tag{5.4}$$

$$Max.\, Activate\, Power = ((I_{DD0} - (I_{DD3N} * T_{RAS} + I_{DD2N} * (T_{RC} - T_{RAS}))/T_{RC}) * V_{DD}) \tag{5.5}$$

Background Power is the combination of many components. These components are listed below.

$$act\_pdn = I_{DD3P} * V_{DD} * \%(Time\, Spent\, in\, PowerDown\, with\, atleast\, one\, Bank\, Active) \tag{5.6}$$

Table 5.4. Voltage and current parameters of modeled chips.

Parameter	1Gb x4	1Gb x8	1Gb x16	2Gb x4	2Gb x8	4Gb x4	4Gb x8
VDD	1.5	1.5	1.5	1.5	1.5	1.5	1.5
IDD0	70	70	85	42	42	55	55
IDD2P0	12	12	12	12	12	16	16
IDD2P1	30	30	30	15	15	32	32
IDD2N	45	45	45	23	23	28	28
IDD3P	35	35	35	22	22	38	38
IDD3N	45	45	50	35	35	38	38
IDD4R	140	140	190	96	100	147	157
IDD4W	145	145	205	99	103	118	128
IDD5	170	170	170	112	112	155	155

$$act\_stby = I_{DD3N} * V_{DD} * \%(Time\,Spent\,in\,Active\,Standby) \tag{5.7}$$

$$pre\_pdn\_slow = I_{DD2P0} * V_{DD} * \%(Time\,Spent\,in\,PreCharge\,Powerdown\,Slow\,Mode) \tag{5.8}$$

$$pre\_pdn\_fast = I_{DD2P1} * V_{DD} * \%(Time\,Spent\,in\,PreCharge\,Powerdown\,Fast\,Mode) \tag{5.9}$$

$$pre\_stby = IDD2N * V_{DD} * \%(Time\,Spent\,in\,Standby\,with\,all\,Banks\,PreCharged) \tag{5.10}$$

Finally,

$$Background\,Power = act\_pdn + act\_stby + pre\_pdn\_slow + pre\_pdn\_fast + pre\_stby \tag{5.11}$$

Power dissipated in the ODT resistors is called the Termination Power. Termination Power not only depends on the activity in the rank in question but also depends on the activity in other ranks on the same channel. Power dissipated due to reads and writes terminating in the rank in question is given by

$$ReadTerminate = pds\_rd * \%Cycles\,Reads\,from\,this\,Rank \tag{5.12}$$

$$WriteTerminate = pds\_wr * \%Cycles\,Writes\,to\,this\,Rank \tag{5.13}$$

$$ReadTerminateOther = pds\_termRoth * \%Cycles\,Reads\,from\,other\,Ranks \tag{5.14}$$

$$WriteTerminateOther = pds\_termWoth * \%Cycles\,Writes\,to\,other\,Ranks \tag{5.15}$$

We use the same rank configuration as assumed in the Micron Technical Note [93]; hence, we assume the same ODT power dissipation. The values of *pds_rd*, *pds_wr*, *pds_termRoth*, *pds_termWoth* are taken from the Micron Technical Note [93].

The total chip power is the sum of the individual components above. The above DRAM chip power must be multiplied by the number of DRAM chips to obtain total memory system power.

5.3.3 System Power Model

When computing energy-delay-product (EDP), we must multiply system power with the square of system execution time. For our 4-channel configuration, we assume that our system incurs 40 W of constant power overheads for processor uncore components, I/O, disk, cooling, etc. Each core (including its private LLC) incurs a power overhead of 10 W while a thread is running and 0 W (perfect power gating) when a thread has finished. The rest comes from the memory system, with the detailed estimation described above. Our 1-channel configuration is supposed to represent a quarter of a future many-core processor where each channel on the processor chip will be shared by many simple cores. Consequently, our system power estimate with this configuration assumes only 10 W for miscellaneous system power (a total 40 W that is divided by 4) and 5 W peak power per core (since the core is simpler). Similar to the 4-channel configuration, a core is power-gated once a thread has finished executing. In either system power model, the memory system typically accounts for 15–35% of total system power, consistent with many reported server power breakdowns [1], [94]–[97].

5.4 Using USIMM

The USIMM simulator can take multiple workload traces as input. Each workload trace represents a different program running on a different core, with memory accesses filtered through a 512 KB private LLC. The JWAC MSC will later construct and release a specific set of workload traces, including commercial workload traces, which will be used for the competition. The initial USIMM distribution has a few short traces from single-thread executions of the PARSEC suite that can be used for testing.

The simulator is executed with multiple arguments. The first argument specifies the configuration file for the processor and memory system. The remaining arguments each specify an input trace file. The number of cores is the same as the number of input trace files. The traces only contain the instruction types and memory addresses accessed by a program, but no timing information (the timing is estimated during the simulation). Based on the address being touched by a memory instruction, the request is routed to the appropriate memory channel and memory controller.

Some of the traces are derived with publicly available benchmarks. These benchmarks are executed with Windriver Simics [48] and its g-cache module to produce the trace. Some of the traces are derived from commercial workloads. To keep simulation times manageable, the traces released for the JWAC MSC simulated a few million instructions, but these shortened traces were representative of the behavior for a longer execution.

Each thread's trace is restricted to a 4 GB space. When multiple traces are fed to USIMM, the address space grows and each trace is mapped to its own 4 GB space within this address space. This is implemented by adding bits to the trace address (corresponding to the core ID). These additional bits are interpreted as part of the row address bits. Thus, as more cores are added, the DRAM chips are assumed to have larger capacities.

Modeling a shared cache would require us to predetermine the threads that will share the cache. We therefore assume that each thread's trace is filtered through a private LLC. Since each core and trace is now independent, we can construct arbitrary multicore workloads by feeding multiple traces to USIMM. When generating a trace for a multithreaded application, we must confirm that a memory access is included in a thread's trace only after checking the private LLCs of other threads.

The JWAC MSC focused on two main system configurations. The first uses a smaller scale processor core and a single memory channel, while the second uses a more aggressive processor core and four memory channels. The two configurations are summarized in Table 5.5, with the differences in bold. While a single channel appears underprovisioned by today's standards, it is more representative of the small channel-to-core ratio that is likely in future systems. Table 5.6 lists all the different workloads that were distributed with the simulator.

5.5 Validation Against Micron DDR3 Verilog Models

Validating a DRAM timing simulator is a nontrivial task. The command scheduler needs to take into account the state of the DRAM banks, needs to be cognizant of the consequence of the interaction of different commands, and most importantly, makes sure to not violate the minimum timing delays between different commands. We design a validation methodology for the USIMM simulator that aims to make sure that the command scheduler never issues commands at a rate faster than what is allowed by the physical characteristics of the DRAM device (as represented by the timing parameters). For this, we make use of the Verilog device models provided by Micron for DDR3 devices. These Verilog timing models have easily customizable

Table 5.5. System configurations used for the JWAC MSC.

Parameter	**1channel.cfg**	**4channel.cfg**
Processor clock speed	3.2 GHz	3.2 GHz
Processor ROB size	128	**160**
Processor retire width	2	**4**
Processor fetch width	4	4
Processor pipeline depth	10	10
Memory bus speed	800 MHz	800 MHz
DDR3 Memory channels	1	**4**
Ranks per channel	2	2
Banks per rank	8	8
Rows per bank	32768 × NUMCORES	32768 × NUMCORES
Columns (cache lines) per row	128	128
Cache line size	64 B	64 B
Address bits	32+log(NUMCORES)	34+log(NUMCORES)
Write queue capacity	64	96
Address mapping	rw:rnk:bnk:ch:col:blk	rw:col:rnk:bnk:ch:blk
Write queue bypass latency	10 cpu cycles	10 cpu cycles

Table 5.6. USIMM workloads

Workload No.	**Benchmarks**
1	4-threaded canneal
2	blackscholes-blackscholes-freqmine-freqmine
3	commercial1-commercial1
4	commercial1-commercial1-commercial2-commercial2
5	commercial2
6	facesim-facesim-ferret-ferret
7	fluidanimate-fluidanimate-ferret-ferret-swaptions -swaptions-commercial2-commercial2
8	fluidanimate-fluidanimate-ferret-ferret-swaptions -swaptions-commercial2-commercial2 -blackscholes-blackscholes-freqmine-freqmine -commercial1-commercial1-stream-stream
9	fluidanimate-swaptions-commercial2-commercial2
10	stream-stream-stream-stream

DRAM timing parameters (to model different kinds of DRAM chips) and define a set of Verilog tasks, each of which corresponds to one DRAM command. These Verilog modules, in conjunction with a Verilog input file that contains a list of the different DRAM commands in the form of tasks, can be simulated in ModelSim. The simulation will report violations if consecutive DRAM commands do not have the necessary gap between them.

To verify USIMM, we first log all commands issued, their target address, and also the time that elapses between consecutive commands. This log is then converted to a Verilog file which serves as the test input for subsequent ModelSim simulations. When this file is run with ModelSim, it basically amounts to replaying the commands scheduled by the USIMM memory controller on the Verilog model. Any USIMM simulation can be used to generate the log file and the consequent Verilog test input and can be verified against the Micron models.

With our verification methodology, we ran several programs and scheduling algorithms. Of particular importance were the tests with the following three schedulers.

- **FCFS** This algorithm (see Section 5.1.14) issues a command as soon as it is ready to be issued. In other words, this is the most aggressive in terms of issuing DRAM commands and problems in the timing model are likely to be exposed relatively easily with this scheduler.
- **Power-Down** This algorithm tries to aggressively put banks to sleep whenever there are idle cycles, i.e., there are no pending requests to a bank. This test ensures that the power-down modes are entered with the right delay. In addition, we also modify this scheduler and create a different version for the purposes of this test to also check if all banks can be precharged in a cycle and be put in the PRE-PDN-SLOW (or deep power down) mode to test correctness.
- **Refresh-enforce** This algorithm issues a refresh command to a rank every tREFI cycles. This models the baseline refresh mechanism adopted by modern memory controllers.

In all cases, we found that USIMM strictly follows the timing restrictions, i.e., it never issues commands earlier than they should be. It can thus be said with reasonable confidence that the DRAM modules in USIMM do not provide overestimates for performance numbers.

There are however drawbacks to this verification scheme. Each validation run of USIMM only shows that the scheduler in question is conservative enough to not violate any timing constraints. It does not conclusively prove that a scheduler can not be developed which causes USIMM to violate the timing constraints. To address this, we wrote a scheduler which achieves no particular scheduling tasks, but tries to, in a sense, stress-test USIMM. Every cycle, this stress-testing scheduler picks a command at random from the list of all possible commands, and issues it to the memory system. If the command is valid, then it is accepted by USIMM and logged for validation. If USIMM flags the command as nonissuable in the current cycle, the scheduler tries to schedule another command that it picks at random from the list of commands. Thus, a cycle is advanced only when the scheduler runs out of commands to schedule or a command is accepted. We believe that this is a sufficiently aggressive scheduler that can expose problems with USIMM's timing model. We tried three runs of this scheduler (each spanning 1 million DRAM commands) to collect logs for the Verilog models and did not observe any violations reported by ModelSim.

5.6 Alternative Software Architecture

The USIMM model has a unified transaction and command scheduler. The read and write queues contain detailed information about the command timing which can be exploited by the scheduler to take "informed" scheduling decisions. In comparison, the memory controllers employed in modern processors have a different organization, similar to the organization described in Section 4.2.3. In this organization, a transaction scheduler selects a read or write based on high-level scheduling decisions such as thread-priority, memory-level parallelism, write-queue occupancy, etc. The transaction scheduler enqueues the commands required for each transaction to complete into the per-bank command queues. The command scheduler then goes over the bank queues in a round-robin fashion and issues commands as they become ready to issue based on timing constraints and DRAM bank states. The main motivation behind the unified queue and schedulers in USIMM is to allow the scheduler to be aware of microarchitectural details of the DRAM device and at the same time be able to take decisions based on system-level considerations. This enables the user to control scheduling decisions at a single point in the simulator's program flow.

We implemented another version of USIMM with the same organization as described in Section 4.2.3 (that we call *alternate-USIMM* hereafter) and compared the performance of the baseline scheduler distributed with USIMM for MSC on the two systems. The performance difference between the two is quite small. Fig. 5.1 shows that the maximum difference in performance reported by Alternate-USIMM is 3% over USIMM. However, it requires significant programmer effort in native USIMM

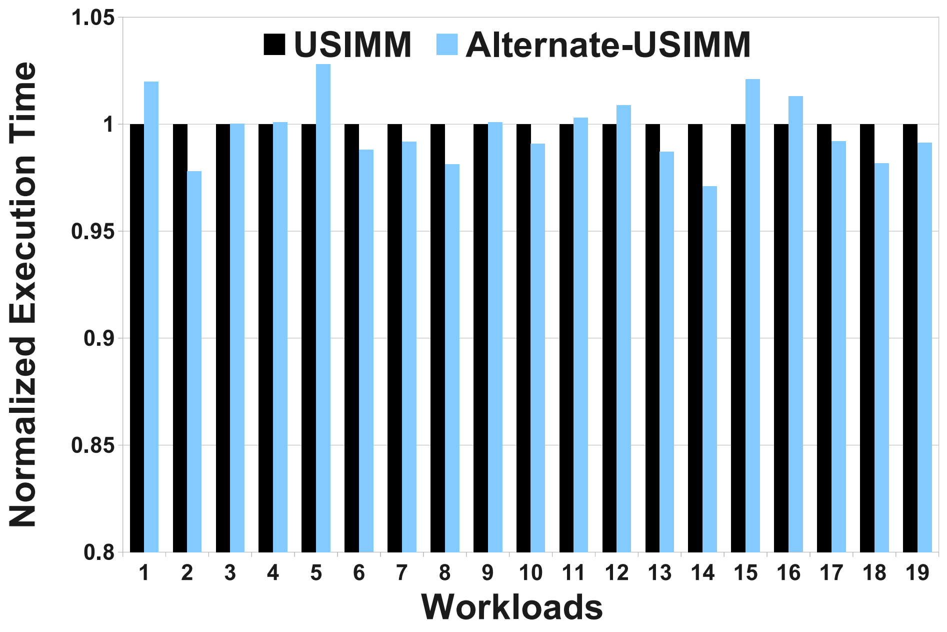


Figure 5.1. Comparison of Execution Times Reported by USIMM vs Alternate-USIMM (workloads from Table 5.6).

to correctly mimic the new controller organization. For example, it is possible that in native USIMM, the scheduling decision taken for a single request (by issuing an ACT) is rendered useless in a subsequent cycle because some other request issues a PRE to the same bank before the COL-RD for the first request has been issued. This will not happen in alternate-USIMM because the decisions taken by the transaction scheduler are never reversed by the *in-order* command scheduler.

In summary, the IPCs reported by USIMM and alternate-USIMM do not diverge significantly. Alternate-USIMM represents real-world hardware designs and hence is attractive for fidelity. However, by dissociating the task of command and transaction scheduling, it prevents researchers from exploring schedulers which have fine-grained, and customized control over command scheduling. On the other hand, USIMM's unified queues and scheduler model requires significant programmer effort for modification, but also allows exploration of timing-aware transaction schedulers.

5.7 Integrating USIMM with a Full-System Simulator

The importance of memory in determining the performance of the overall system is well known. In spite of this, many full-system simulators have relatively simple memory models—often simulating fixed access-latency for cache-misses. The full-system simulator, SIMICS [48], is one such example. It has a detailed out-of-order processor timing model and can model multiple levels of caches with different organizations and the MESI cache-coherence protocol. However, the main memory timing is fixed at 200 processor cycles.

We integrated the USIMM DRAM timing and power models with the SIMICS simulator. Since both these simulators are cycle-accurate and operate on a cycle-by-cycle basis, it is fairly straightforward to integrate SIMICS and USIMM. Hereafter in the text, SIMICS refers to this combined simulator. The main advantage of using trace-based simulators such as USIMM is the speed of simulation. With USIMM, we observed 7X, 154X, and 289X speedup over SIMICS when simulating single-core, 4-core, and 8-core versions of the NAS Parallel Benchmarks (each core running 100 million instrusctions). We see that USIMM is significantly faster than the full-system simulator SIMICS. However, the main differences between the USIMM simulator and SIMICS are the former's lack of a detailed processor model and the inability to model cache-contention for multiprogrammed/multithreaded workloads. To assess the accuracy of USIMM compared to SIMICS, we compared the performance of the MSC schedulers on the two simulators. We observed that the DRAM latencies and IPCs reported by USIMM are both higher compared to SIMICS. This happens because in SIMICS, the nonmemory instructions are deemed independent of each other and also the memory instructions in the reorder-buffer are considered completely independent of each other and not dependant on the result of the other nonmemory instructions for their memory addresses. As a result, all nonmemory instructions have a CPI of 1 which is not true in SIMICS. Secondly, this also causes the memory system to encounter much higher traffic compared to SIMICS, leading to higher queueing delays and hence higher overall DRAM latencies in USIMM. The relative performance of the different MSC schedulers remained unchanged.

To increase the accuracy of the USIMM processor model, while still retaining the simple trace-based features and speed of USIMM, we model dependences between instructions in USIMM by stalling some instructions for longer than a single cycle based on a probability. Thus two values, extra stall cycles (ESC) and a dependecy factor (DF), were determined empirically for each benchmark to equate the performance numbers reported by USIMM and SIMICS. Nonmemory instructions have their completion times extended by ESC cycles with a probability of DF. The ESC and DF values are different for the different benchmarks and, in addition, also show sensitivity to other IPC improvement techniques. We determined the EF and DSC values by comparing the SIMICS reported IPC with USIMM simulations consisting of a single-core, single-channel, and FR-FCFS scheduler. In Fig. 5.2, we show the IPCs reported by USIMM, SIMICS and augmented-USIMM (i.e., the one using DF and ESC) for a set of single-core PARSEC benchmarks. The DF and ESC values are shown in Table 5.7. We see that augmented-USIMM is able to bridge the performance gap between USIMM and SIMICS for all the benchmarks. To investigate the effect of the scheduling policy on the ESC and DF values, we compare the error between augmented-USIMM and SIMICS for the best performing MSC scheduler by Ishii *et al* [98]. We find that for most benchmarks, the errors range between 1% to 2.3%. Finally, we ran all the MSC schedulers with multicore PARSEC benchmarks. We found that the relative performance of the different schedulers is not affected. Fig. 5.3 shows the average IPC of a set of 4-core PARSEC benchmarks reported by

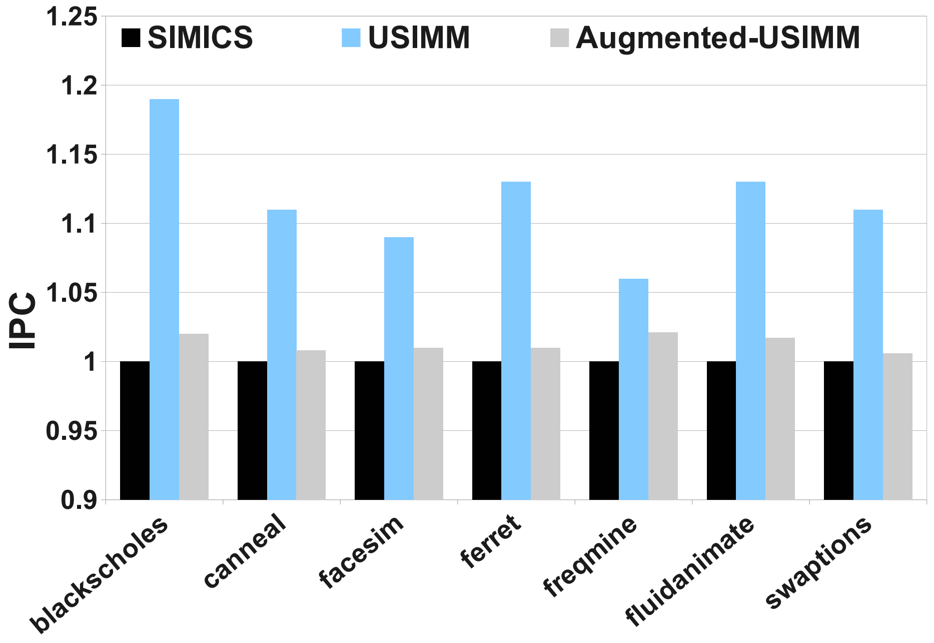


Figure 5.2. Comparison of the IPCs reported by SIMICS, USIMM and Augmented-USIMM for the FR-FCFS scheduler on single-core PARSEC Benchmarks.

Table 5.7. Fudge values

Benchmark	ESC	DF
blackscholes	60	.66
canneal	41	.5
facesim	39	.5
ferret	28	.5
freqmine	43	.66
fluidanimate	46	.66
swaptions	32	.50

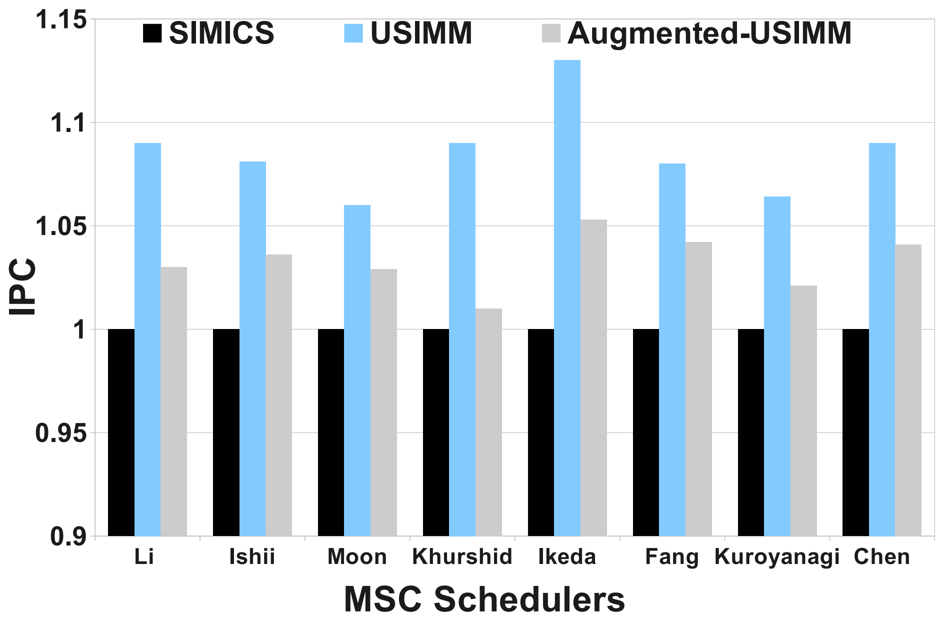


Figure 5.3. Comparison of the IPCs reported by SIMICS, USIMM and Augmented-USIMM for all MSC schedulers on 4-core PARSEC Benchmarks.

SIMICS, USIMM and Augmented-USIMM for all the MSC schedulers. We see that the difference between SIMICS and augmented-USIMM is within 5% in terms of IPC. This indicates that the results reported by stand-alone, augmented-USIMM can be used to draw reasonable first-impressions on memory system optimizations through simulations that are significantly faster than SIMICS.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This chapter summarizes the topics explored in this dissertation and describes future scheduling techniques for emerging memory systems.

6.1 Contributions

In this dissertation, we discuss novel memory scheduling techniques that extend the state-of-art. Memory scheduling has received significant attention over the years, more so in the recent past. Through the techniques we have proposed in this dissertation, we draw attention to memory schedulers that are aware of the microarchitectural intricacies of the DRAM devices and the execution idiosyncrasies of the client compute units. We also describe a new main memory system simulator that can assist the community in analyzing the memory system behavior for future systems.

To summarize, we now list the major highlights of each of the described memory scheduling techniques and the USIMM tool.

- In Chapter 3, we quantified the impact of write scheduling on the overall system performance. We also identified that simple additions to the DRAM chip can promote write and read parallelism and significantly boost performance. For this, we instituted a small set of buffers inside the DRAM chip's least cost-sensitive area. While writes are being sent to some banks inside a DRAM chip, reads can fetch data from some other banks and store them inside the staged-read registers. When the write queue has been drained, the data from the staged-read registers can be drained over the data bus after the bus-turnaround time has elapsed. Aided by the modifications to the DRAM chip, the memory scheduler is able to provide higher performance by selectively issuing writes to banks that have fewer reads and issuing simultaneous reads to banks that are not servicing writes. The impact of this scheme is even higher when future technology trends, such as increased reliability (which increases write-traffic) and nonvolatile memories with long write times are considered.

- In Chapter 4, we present several memory scheduling techniques for increasing the efficiency of GPUs when they execute irregular compute workloads. Due to the SIMT execution model of the GPU, each load instruction in a GPU issues many memory accesses and the warp issuing the load instruction is blocked until all the memory requests are serviced by the memory system. We observed that modern memory scheduling policies often take scheduling decisions that allow the interleaved servicing of requests from different warps, which leads to higher average wait time for the warps. We propose servicing requests from a warp in a grouped manner, to reduce this interwarp interference. We propose mechanisms to group requests from a warp together through a batching mechanism. We then seek to arbitrate between these different batches using shortest-job-first policy, which takes into account the state of the DRAM banks and the approximate service time of each warp-group (BASJF). To ensure that different memory controllers finish servicing requests from a warp at approximately the same time, we introduced a strong age-bias in the BASJF policy (BASJF+AB). This allowed implicit coordination between the schedulers in different memory controllers when servicing a single warp. However, the age-bias also forces the memory controller to schedule row-miss requests before row-hit requests, thereby causing some performance degradation. To overcome this issue, we ensure that the bandwidth utilization is not impacted while scheduling row-miss requests. This is done by carefully orchestrating the scheduling of row-miss requests in one bank such that they overlap with row-hits in other banks. We finally devise a warp-aware write-drain policy that scans the read queues to service requests from warps with few (and orphaned) requests.
- In Chapter 5, we discuss the design of USIMM, a detailed memory system simulator which was distributed publicly as the infrastructure for the Memory Scheduling Championship held with ISCA 2012. We discuss in detail the timing and power models instituted in USIMM and also discuss the manner in which the scheduler can be used to rapidly develop scheduling algorithms. We then

described a validation methodology for the simulator, which allows a log of the simulation run to be verified against Micron's DDR3 Verilog models for correctness. We also compare the relatively simple software model of USIMM to a more accurate representation of the hardware controller and show that the simple USIMM model is sufficiently faithful to actual memory controllers. We finally integrate USIMM with SIMICS, a full-system simulator, and compare the accuracy of the trace-driven USIMM model against the combined simulator. We find that the lack of dependence modeling in USIMM's simple CPU model is the source of inaccuracy. We use a simple probability-based stalling of nonmemory instructions to equalize the IPCs reported by USIMM and SIMICS and find that the new simulator is within acceptable accuracy levels of simulators with complex processor models.

We conclude that there are several settings where the memory controller has a nontrivial impact on system performance. The staged read proposal can yield 7% improvement, but may face opposition to commercial adoption because of DRAM chip changes. Interestingly though, upcoming DDR4 devices have adopted a chip microarchitecture that has some similarity with the Staged Read optimization. In DDR4, the 16 on-chip banks are split into 4 bank-groups. Each such bank-group has its own dedicated read/write bus that goes to the I/O pads from the sense amps. Compared to this, in DDR3, all banks share the bus to the I/O pads, while in staged-reads, every bank has its own path to the I/O pads. This innovation allows DDR4 to have one-third the bus turnaround penalty when interleaving writes and reads to *different* bank-groups compared to interleaved writes and reads to the same bank-group.

Scheduler innovations, on the other hand, can yield a maximum of 11 % improvement, as shown in the MSC. We improve our tool and show that the improvement continues to be similar even in a detailed simulator. However, it is possible that the MSC submissions may have picked the low-hanging fruit, and not too many additional improvements are possible for DDR3. We examine an emerging platform, the GPGPU, and show that scheduler innovations can continue to yield significant (8.9%) improvements as architectures and workloads evolve beyond DDR3.

6.2 Future Work

In this section, we look at how emerging trends can lead to interesting applications of novel memory scheduling strategies.

6.2.1 Scheduling for Heterogeneous Platforms

Advances in process technology have enabled architects to integrate functional blocks of different types on the same logic die. For example, AMD's Fusion series of Accelerated Processing Units (APUs) feature a modest GPU alongside multiple CPU cores. Integrating CPUs and GPUs on the same die has interesting implications on the memory architecture of such systems. In the Fusion series, for example, the DRAM memory controller is shared by the CPUs and the GPU(s), and these different clients have different expectations from the memory system. GPUs, by design, are more tolerant of memory latency but require high effective memory bandwidth. CPUs, on the other hand, are generally more sensitive to memory latency. A memory scheduling strategy should thus be able to satisfy the diverse requirements of these different clients. We tried a scheduler that implements the minimum-efficient row-burst (MERB, Chapter 4) for this purpose. This scheme allows the controller to figure out the earliest time when a row-miss request from a CPU can be serviced given the total number of row-hit requests to other banks. We observed that when MERB is used as a scheduling policy in a system that runs a bandwidth-intensive CPU application with other latency-sensitive CPU programs, it can improve the system performance by 4% over the PAR-BS scheme. The main issue with the current evaluation technique is the lack of real-world benchmarks which use the GPU and CPU concurrently. Most applications execute some sequential series of operations on a CPU and then delegate the task to the GPU which spawns a large number of threads that perform the number-crunching while the CPU sits unutilized. We plan to explore the applicability of our scheme in the future when new benchmarks that simultaneously execute on the CPU and GPU become available.

6.2.2 Scheduling for HMCs

Hybrid Memory Cube [63] is a new memory technology that stacks multiple layers of DRAM on a logic layer. The HMC device can be connected to a processor(s) through a fast, serial link. One interesting problem to study in the context of the HMC is the division of scheduling responsibilities between the on-chip memory controller and the controller that sits on the logic layer of the HMC. One obvious design point is the implementation of the high level transaction scheduler in the on-chip memory scheduler and the command scheduler in the HMC's logic layer. This allows the on-chip memory controller to deal with thread priorities and other memory access patterns, while allowing the command scheduler to implement policies that are best suited for the HMC DRAM layers. Specifically, this division allows the HMC's logic layer to present a layer of abstraction to the on-chip memory controller and implement DRAM device specific protocols.

6.2.3 Scheduling for Mobile Devices

A lot of attention has been paid to memory controller designs for large servers. However, handheld devices are growing rapidly and require equal attention to their memory systems. A study of scheduling techniques for mobile devices is made interesting by the following characteristics of the mobile environment. First, the memory scheduling logic itself has to be simple, to save expensive chip area. Second, the main emphasis for the scheduler would be energy-efficiency instead of performance. As a result, techniques that can efficiently leverage the numerous power-down modes of Low-Power DRAM (LPDDR3) without significant performance penalty need to be investigated.

REFERENCES

[1] L. A. Barroso and U. Holzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan and Claypool, 2009. http://www.morganclaypool.com/doi/pdf/10.2200/s00193ed1v01y200905cac006

[2] U.S. Environmental Protection Agency, "Report to congress on server and data center energy efficiency," tech. rep., Public Law 109-431, Washington D.C., 2007.

[3] Seamicro Inc., "SeaMicro servers." [Online], Available: http://www.seamicro.com

[4] Calxeda Inc., "Calxeda servers." [Online], Available: http://www.calxeda.com

[5] ITRS, "International technology roadmap for semiconductors, 2007 edition." [Online], Available: http://www.itrs.net/Links/2007ITRS/Home2007.htm

[6] NVIDIA Corporation, "NVIDIA cuda C programming guide v4.2." [Online], Available: http://developer.nvidia.com/nvidia-gpu-computing-documentation

[7] Khronos Group, "OpenCL." [Online], Available: http://www.khronos.org/opencl

[8] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems," in *Proc. ISPASS*, 2012.

[9] A. N. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi, "LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems," in *Proc. ISCA*, 2012.

[10] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proc. MICRO*, 2007.

[11] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *Proc. ISCA*, 2009.

[12] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory access scheduling," in *Proc. ISCA*, 2000.

[13] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *Proc. HPCA*, 2010.

[14] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proc. MICRO*, 2010.

[15] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. Loh, and O. Mutlu, "Staged memory scheduling: Achieving high performance and scalability in hetergenous systems," in *Proc. ISCA*, 2012.

[16] J. Stuecheli, D. Kaseridis, D. Daly, H. Hunter, and L. John, "The virtual write queue: Coordinating DRAM and last-level cache policies," in *Proc. ISCA*, 2010.

[17] H. Lee and G. Tyson, "Eager writeback – a technique for improving bandwidth utilization," in *In Proc. MICRO*, 2000.

[18] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proc. ISCA*, 2010.

[19] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications," in *IEEE Transactions on Multimedia*, 2012.

[20] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: The Utah Simulated Memory Module," Tech. Rep. UUCS-12-002, University of Utah, Salt Lake City, UT, 2012.

[21] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. Burlington, MA: Morgan Kauffman, 2008.

[22] JEDEC Solid State Technology Association, "DDR3 SDRAM specification," Tech. Rep. JESD79-3E, Arlington, VA, 2010.

[23] JEDEC Solid State Technology Association, "DDR4 SDRAM specification," Tech. Rep. JESD79-4, Arlington, VA, 2012.

[24] R. Swinburne, "Intel Core i7 – nehalem architecture dive." [Online], Available: http://www.bit-tech.net/hardware/2008/11/03/intel-core-i7-nehalem-architecture-dive/

[25] V. Romanchenko, "Quad-Core opteron: Architecture and roadmaps." [Online], Available: http://www.digital-daily.com/cpu/quad_core_opteron

[26] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (SALP) in DRAM," in *Proc. ISCA*, 2012.

[27] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. Jouppi, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *Proc. ISCA*, 2010.

[28] 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. ASPLOS-XV*, 2010.

[29] Z. Zhang, Z. Zhu, and X. Zhand, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proc. MICRO*, 2000.

[30] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *Proc. HPCA*, 2012.

[31] Micron Technology Inc., "Micron DDR3 SDRAM part MT41J256M8," 2006. [Online], Available: http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM\\.pdf

[32] Micron Technology Inc., "Micron DDR3 SDRAM part MT41J1G4," 2009. [Online], Available: http://download.micron.com/pdf/datasheets/dram/ddr3/4Gb_DDR3_SDRAM\\.pdf

[33] M. Bojnordi and E. Ipek, "PARDIS: A programmable memory controller for the DDRx interfacing standards," in *Proc. ISCA*, 2012.

[34] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling – enhancing both performance and fairness of shared DRAM systems," in *Proc. ISCA*, 2008.

[35] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi, "Staged reads: mitigating the impact of DRAM writes on DRAM reads," in *Proc. HPCA*, 2012.

[36] M. Burtscher and K. Pingali, *GPU Computing Gems Emerald Edition*, ch. An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-body Algorithm. San Francisco, CA: Morgan Kaufmann, 1st ed., 2011.

[37] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. IISWC*, 2012.

[38] S. Ghose, H. Lee, and J. F. Martinez, "Improving memory scheduling via processor-Side load criticality information," in *Proc. ISCA*, 2013.

[39] P. Prieto, V. Puente, and J. A. Gregorio, "CMP off-chip bandwidth scheduling guided by instruction criticality," in *Proc. ICS*, 2013.

[40] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *Proc. PACT*, 2010.

[41] E. Ipek, O. Mutlu, J. Martinez, and R. Caruana, "Self optimizing memory controllers: A reinforcement learning approach," in *Proc. ISCA*, 2008.

[42] J. Borkenhagen and B. Vanderpool, "Read prediction algorithm to provide low latency reads with SDRAM cache," Oct 5, 2004. US Patent 6801982 A1.

[43] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *Proc. MICRO*, 2010.

[44] D. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," in *Proc. ASPLOS*, 2010.

[45] M. Qureshi, V. Srinivasan, and J. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. ISCA*, 2009.

[46] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proc. ISCA*, 2009.

[47] Micron Technology Inc., "Micron DDR3 SDRAM part MT41J128M8," 2006. [Online], Available: http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf,

[48] WindRiver Corp., "WindRiver Simics full system simulator." [Online], Available: http://www.windriver.com/products/simics/

[49] D. Wang *et al.*, "DRAMsim: A memory-system simulator," in *SIGARCH Computer Architecture News*, September 2005.

[50] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Proc. IISWC*, 2008.

[51] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, and V. V. andd S. Weeretunga, "The NAS parallel benchmarks," *International Journal of Supercomputer Applications*, vol. 5, pp. 63–73, Fall 1994.

[52] SPEC, "SPEC Java virtual machine benchmark," 2008. [Online], Available: http://www.spec.org/jvm2008

[53] J. D. McCalpin, "STREAM – sustainable memory bandwidth in high performance computers." [Online], Available: http://www.cs.virginia.edu/stream/

[54] S. Rixner, "Memory controller optimizations for web servers," in *Proc. MICRO*, 2004.

[55] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima, "The cache DRAM architecture: A DRAM with an on-chip cache memory," *IEEE Micro*, vol. 10, no. 2, 1990.

[56] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. MICRO*, 2007.

[57] NEC Corporation, "64M-bit virtual channel SDRAM data sheet," tech. rep., Japan, 2003.

[58] Micron Technologies Inc., "Mobile DRAM power-saving features and power calculations," Tech. Rep. TN-46-12, Boise, ID, 2009.

[59] Micron Technologies Inc., "Low-power versus standard DDR SDRAM," Tech. Rep. TN-46-15, Boise, ID, 2007.

[60] Micron Technologies Inc., "Exploring the RLDRAM II feature set," Tech. Rep. TN-49-02, Boise, ID, 2004.

[61] A. N. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi, "Combining memory and a controller with photonics through 3D-stacking to enable scalable and energy-efficient systems," in *Proc. ISCA*, 2011.

[62] D. H. Woo *et al.*, "An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth," in *Proc. HPCA*, 2010.

[63] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube – new DRAM architecture increases density and performance," in *Symposium on VLSI Technology*, 2012.

[64] B. Schroeder, E. Pinheiro, and W. D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," in *Proc. SIGMETRICS*, 2009.

[65] Hewlett-Packard, "HP ProLiant server memory." [Online], Available: http://h18000.www1.hp.com/products/servers/technology/memoryprotection.html

[66] M. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. MICRO*, 2009.

[67] M. Qureshi, M. Franceschini, L. Lastras-Montano, and J. Karidis, "Morphable memory system: A robust architecture for exploiting multi-level phase change memory," in *Proc. ISCA*, 2010.

[68] F. Tabrizi, "Non-volatile STT-RAM: A true universal memory." [Online], Available: http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2009/20090813_ThursPlenary_Tabrizi.pdf

[69] ITRS, "International technology roadmap for semiconductors, 2009 edition." [Online], Available: http://www.itrs.net/Links/2009ITRS/Home2009.htm

[70] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU implementation of inclusion-based points-to analysis," in *Proc. PPoPP*, 2012.

[71] D. G. Merrill, M. Garland, and A. S. Grimshaw, "Scalable GPU graph traversal," in *Proc. PPoPP*, 2012.

[72] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. Das, "OWL: Cooperative thread array scheduling techniques for improving GPGPU performance," in *Proc. ASPLOS*, 2013.

[73] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke, "PEPSC: A power-efficient processor for scientific computing," in *Proc. PACT*, 2011.

[74] "NVIDIA Kepler GK110 whitepaper." [Online], Available: http://http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[75] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *Proc. ISCA*, 2009.

[76] Hynix, "Hynix GDDR5 SGRAM part H5GQ1H24AFR revision 1.0." [Online], Available: http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf

[77] G. L. Yuan, A. Bakhoda, and T. M. Aamodt, "Complexity effective memory access scheduling for many-core accelerator architectures," in *Proc. MICRO*, 2008.

[78] A. Bakhoda, G. Yuan, W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. ISPASS*, 2009.

[79] T. M. Aamodt and W. L. Fung, "GPGPU-Sim 3.x manual." [Online], Available: http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual

[80] T. M. Aamodt and W. L. Fung, "GPGPU-Sim accuracy." [Online], Available: http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual\#Accuracy

[81] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W.-. M. W. Hwu, "The parboil technical report," Tech. Rep. IMPACT-12-01, University of Illinois, Urbana-Champaigne, IL, 2012.

[82] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IISWC*, 2009.

[83] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang, "Mars: A MapReduce framework on graphics processors," in *Proc. PACT*, 2008.

[84] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin, "DRAM scheduling policy for GPGPU architectures based on a potential function," *IEEE CAL*, vol. 11, pp. 33–36, July-Dec 2012.

[85] D. Shah and D. Wischik, "Switched networks with maximum weight policies: Fluid approximation and multiplicative state space collapse," *Ann Appl Probab*, vol. 22, no. 1, pp. 70–127, 2012.

[86] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-wware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC," in *Proc. DAC*, 2012.

[87] D. Tarjan, J. Meng, and K. Skadron, "Increasing memory miss tolerance for SIMD cores," in *Proc. SC*, 2009.

[88] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for GPU computing," in *Proc. ASPLOS*, 2012.

[89] S. Che, J. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *Proc. SC*, 2011.

[90] T. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proc. MICRO*, 2012.

[91] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. Das, "Orchestrated scheduling and prefetching for GPUs," in *Proc. ISCA*, 2013.

[92] "Micron system power calculator." [Online], Available: http://www.micron.com/support/dram/power_calc.html

[93] Micron Technologies Inc., "Calculating memory system power for DDR3," Tech. Rep. TN-41-01, Boise, ID, 2007.

[94] J. Laudon, "UltraSPARC T1: A 32-Threaded CMP for servers," 2006. [Online], Available: http://www.cs.duke.edu/courses/fall06/cps220/lectures/UltraSparc_T1_Niagra.pdf

[95] C. Lefurgy *et al.*, "Energy management for commercial servers.," *IEEE Computer*, vol. 36, no. 2, pp. 39–48, 2003.

[96] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proc. ISCA*, 2009.

[97] P. Bose, "The risk of underestimating the power of communication," in *NSF Workshop on Emerging Technologies for Interconnects (WETI)*, 2012.

[98] Y. Ishii, K. Hosokawa, M. Inaba, and K. Hiraki, "High performance memory access scheduling using compute-phase prediction and writeback-refresh overlap," in *Proc. JWAC Memory Scheduling Championship*, 2012.