2006

Reducing main memory access latency through SDRAM address mapping techniques and access reordering mechanisms

Jun Shao
Michigan Technological University

Reducing Main Memory Access Latency through SDRAM Address Mapping Techniques and Access Reordering Mechanisms

by

JUN SHAO

A DISSERTATION

Submitted in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

(Electrical Engineering)

MICHIGAN TECHNOLOGICAL UNIVERSITY

2006

This dissertation, "Reducing Main Memory Access Latency through SDRAM Address Mapping Techniques and Access Reordering Mechanisms", is hereby approved in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY in the field of Electrical Engineering.

DEPARTMENT or PROGRAM: Electrical and Computer Engineering

Dissertation Advisor

Dr. Brian T. Davis

Committee

Dr. Jindong Tan

Dr. Chunxiao Chigan

Dr. Soner Önder

Department Chair

Dr. Timothy J. Schulz

Date

To my parents and friends

Acknowledgments

First and foremost, I wish to express my gratitude to my advisor Dr. Brian T. Davis for his constant support, guidance and encouragement throughout the Ph.D. program. I was deeply impressed by his passion for teaching and his inspiring way to guide me to a deeper understanding of knowledge. He showed me the way to approach research objectives and encouraged me to be an independent researcher. His advice and comments in this dissertation are highly appreciated.

I would also like to thank the advisory committee: Dr. Jindong Tan, Dr. Chunxiao Chigan and Dr. Soner Önder. I am extremely grateful to their helpful advice and insightful comments to my thesis work. Thanks also go to my colleagues and friends, who gave me help and had fun with me. I really enjoy the life with them in the beautiful Keweenaw Peninsula.

Last but not least, a special thanks to my parents. Their continued support and encouragement in my life has truly given me strength through difficult situations and the desire to do my best at everything I do.

Support for this research was provided by National Science Foundation CAREER Award CCR 0133777 at Michigan Technological University.

Abstract

As the performance gap between microprocessors and memory continues to increase, main memory accesses result in long latencies which become a factor limiting system performance. Previous studies show that main memory access streams contain significant localities and SDRAM devices provide parallelism through multiple banks and channels. These locality and parallelism have not been exploited thoroughly by conventional memory controllers. In this thesis, SDRAM address mapping techniques and memory access reordering mechanisms are studied and applied to memory controller design with the goal of reducing observed main memory access latency.

The proposed *bit-reversal address mapping* attempts to distribute main memory accesses evenly in the SDRAM address space to enable bank parallelism. As memory accesses to unique banks are interleaved, the access latencies are partially hidden and therefore reduced. With the consideration of cache conflict misses, bit-reversal address mapping is able to direct potential row conflicts to different banks, further improving the performance.

The proposed *burst scheduling* is a novel access reordering mechanism, which creates bursts by clustering accesses directed to the same rows of the same banks. Subjected to a threshold, reads are allowed to preempt writes and qualified writes are piggybacked at the end of the bursts. A sophisticated access scheduler selects accesses based on priorities and interleaves accesses to maximize the SDRAM data bus utilization. Consequentially burst scheduling reduces row conflict rate, increasing and exploiting the available row locality.

Using a revised SimpleScalar and M5 simulator, both techniques are evaluated and compared with existing academic and industrial solutions. With SPEC CPU2000 benchmarks, bit-reversal reduces the execution time by 14% on average over traditional page interleaving address mapping. Burst scheduling also achieves a 15% reduction in execution time over conventional bank in order scheduling. Working constructively together, bit-reversal and burst scheduling successfully achieve a 19% speedup across simulated benchmarks.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since its introduction in the late 1960s [20], Dynamic Random Access Memory (DRAM) technology has progressed at a rapid pace, quadrupling chip density every three years [38]. As optimized for low cost and high yield, DRAM is commonly used as the main memory for modern PCs as well as many embedded systems. International Technology Roadmap for Semiconductors (ITRS) forecasts that commodity DRAM will continue to double in capacity every three years [29].

DRAM performance can be measured in two ways: by *bandwidth*, which measures the amount of data it can transfer each second, and by *latency*, which measures the length of time between the time data is requested and the time it is returned. DRAM latency does not improve as quickly as that bandwidth. DRAM bandwidth increases by 25% each year [29], and DRAM latency improves by only 5% per year [51]. The research work presented in this thesis is to propose and improve memory optimization techniques to reduce access latency to DRAM. The techniques being studied include SDRAM address mapping and memory access reordering mechanisms.

1.1 Memory Lags Microprocessor

In 1965, Gordon Moore noted that the number of transistors that could be economically fabricated on a single processor die was doubling every year [45]. Moore projected that such an increase was likely to continue in the future, which was referred as "Moore's Law"[1]. Today, approximately 50% more components can be placed on a single die each year [51]. This rate of growth is expected to continue for at least the next decade.

As the number of transistors on integrated circuits increases, the size of each transistor is also decreased. Because of their smaller size, these transistors can operate faster. Historically, transistor speeds have increased by 15% per year [18]. Together, the increase in transistor count and clock speed combine to increase the computing power of microprocessors by 71% per year [51].

Because the DRAM technology has been driven by cost, while the logic technology has been driven by speed, DRAM performance does not increase as quickly as that of microprocessors, leading to a wide gap between slower memory and faster microprocessors. Figure 1.1 illustrates the trend of performance improvements made to microprocessors and DRAM. As performance gap between microprocessors and DRAM continues to increase, main memory becomes a bottleneck of system performance [73, 15].

Improvements of DRAM latency lag behind improvements in DRAM bandwidth [50]. This is mainly because that bandwidth is an issue that can be largely solved by increasing resources, i.e. increasing data bus width or bus clock frequency. However, mechanisms to reduce latency require a reduction in density. Table 1.1 lists typical performances of CPU and memory from a few years back to the near future. CAS latency is a commonly used metric to evaluate DRAM latency and will be discussed in Section 2.1.2. While the bandwidth of DRAM improves significantly by 200% from 2.13GB/s to 6.4GB/s, the absolute

[1] Although Moore's Law is commonly referred to the rapidly continuing advance in computing power per unit cost, Moore's actual prediction referred only to the number of devices that could fit on a single die.

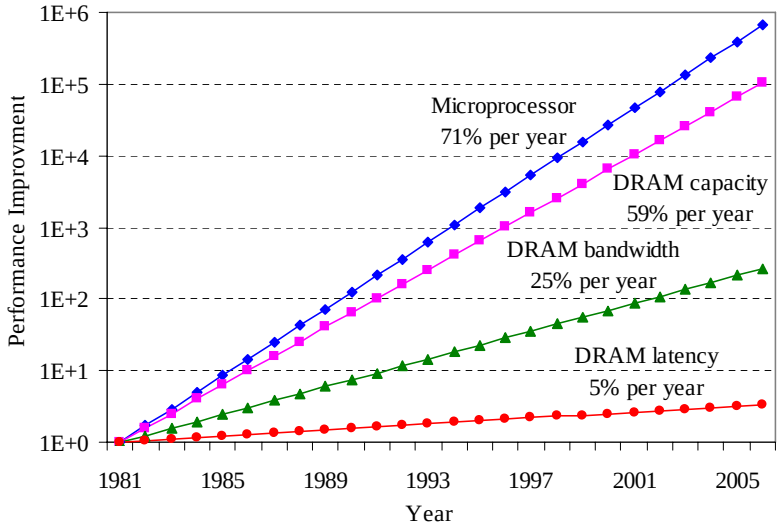

Figure 1.1: Performance improvement of microprocessors and SDRAM devices

DRAM latency only reduces by 17% from 15ns to 12.5ns. More importantly, as CPU improves faster than memory, the memory latency in terms of CPU cycles actually increases from 30 cycles to 50 cycles. Therefore main memory access latency will continue to be a factor limiting system performance.

Table 1.1: Typical performances of CPU and main memory

		A few years back	Today	Near future
CPU	Frequency	2GHz	3GHz	4GHz
Memory	Device	DDR PC-2100	DDR PC-3200	DDR2 PC2-6400
	Frequency	133MHz	200MHz	400MHz
	Bandwidth	2.13GB/s	3.2GB/s	6.4GB/s
	Timing	2-2-2	3-3-3	5-5-5
CAS latency	ns	15ns	15ns	12.5ns
	MEM cycle	2 cycles	3 cycles	5 cycles
	CPU cycle	30 cycles	45 cycles	50 cycles

1.2 Hitting the Memory Wall

Systems have been designed to tolerate the long main memory access time through various techniques, including caches and out-of-order execution. Caches comprised of Static Random Access Memory (SRAM), which is expensive (thus smaller) but much faster than DRAM, can store frequently accessed data for rapid access, therefore reducing the average memory access time. An out-of-order execution processor can continue to execute subsequent instructions when a memory access is outstanding to hide long memory access latency.

Average memory access time is given by Equation 1.1, where p_{cache_hit} is the overall cache hit rate, t_cache is the cache access time, and t_{main_mem} is the main memory access time.

$$t_{ave\_mem} = p_{cache\_hit} \times t_{cache} + (1 - p_{cache\_hit}) \times t_{main\_mem} \tag{1.1}$$

Pessimistically assume an out-of-order execution superscalar CPU can achieve an Instructions Per Cycle (IPC) of 1. And 20% of executed instructions are memory instructions. Also assume the cache has 1 cycle access time and has a 95% of overall hit rate. Main memory has 100 cycles access time. Using Equation 1.1, the average memory access time is 5.95 cycles, as given by Equation 1.2.

$$t_{ave\_mem} = 0.95 \times 1 + (1 - 0.95) \times 100 = 5.95 \ (cycles) \tag{1.2}$$

Although the CPU in this example can continue to execute subsequent non-memory instructions when a memory instruction is pending, it has to idle for $5.95 - 1/0.2 = 0.95$ cycle on average for every memory instruction. In this example, main memory is the bottleneck. Improving the IPC on non-memory instructions will not make the system run faster. The system performance is eventually determined by the memory speed, which is known as "hitting the memory wall" [73].

1.3 Goal of This Thesis

Synchronous DRAM (SDRAM) devices have a nonuniform access time due to the multi-dimensional structure [16, 19, 56]. The memory access latency largely depends upon the location of requested data and the state of the SDRAM device. For example, two temporally adjacent memory accesses directed to the same row of the same bank can complete faster than two accesses directed to different rows of the same bank. This is because an accessed row can be cached at the sense amplifiers allowing faster access for subsequent accesses to that row. In addition, two accesses directed to two unique banks may have shorter latency than two accesses directed to the same bank because accesses to unique banks can be pipelined.

These and other characteristics of SDRAM create a design space where locality and parallelism can be exploited to reduce memory access latency. Previously these locality and parallelism have not been fully exploited by conventional SDRAM controllers. The goal of this thesis is to reduce the observed main memory access latency through memory optimization techniques which do not require any modifications to memory devices, microprocessors, the operating systems or applications. Only the memory controller requires change. The techniques being studied herein are SDRAM address mapping and access reordering mechanisms.

SDRAM address mapping is a protocol in which a physical address is translated into an SDRAM address in order to locate the requested memory block in the SDRAM space. The major objective of SDRAM address mapping is to evenly distribute memory accesses in the entire SDRAM space to enable the parallelism available between SDRAM banks.

With an out-of-order execution processor and non-blocking caches, it is common that multiple memory requests are issued to main memory in the same time. The order in which these pending memory requests are served has impacts on system performance, because the actual memory access time is largely dependent upon the current access and the previ-

ous access to the same bank. Access reordering mechanisms execute outstanding memory accesses in a sequence that attempts to yield the shortest overall execution time.

SDRAM address mapping techniques and access reordering mechanisms can effectively reduce main memory access latency by exploiting both parallelism and locality. The techniques being proposed do not necessitate a large amount of chip area as required by other techniques such as caches, and only require modifications to the SDRAM controller.

1.4 Contributions

This thesis makes the following contributions:

- Makes modifications and improvements to existing computer architecture simulators which enable comprehensive studies of SDRAM optimization techniques.
- Predicts the upper bound improvement of a dynamic SDRAM controller policy over static controller policies.
- Studies and evaluates performance impacts contributed by SDRAM address mapping techniques and access reordering mechanisms.
- Proposes bit-reversal address mapping and describes the depth of reversal parameter. Compares bit-reversal with existing address mapping techniques including published and commercial solutions.
- Performs a study of the SDRAM address mapping in the presence of virtual paging.
- Proposes burst scheduling which creates bursts by clustering accesses directed to the same rows of the same banks to achieve a high SDRAM data bus utilization. Makes optimizations to burst scheduling by allowing reads to preempt writes and piggybacking writes at the end of bursts.

- Evaluates the performance of burst scheduling. Compares with conventional in order scheduling, published academic and industrial out of order scheduling.
- Explores the design space of burst scheduling by using a static threshold to control read preemption and write piggybacking. Determines the threshold that yields the shortest execution time by experiments and proposes a dynamic threshold algorithm.
- Combines bit-reversal address mapping and burst scheduling to achieve a maximal performance improvement.

1.5 Terminology and Assumptions

Throughout the rest of this thesis, the term *main memory access*, or *access*, denotes a memory request issued by the processor, such as a read or write to a memory location. Due to the existence of cache(s), a main memory access is actually a cache miss from the lowest level cache. In this thesis main memory accesses from other devices such as DMA controllers are not considered, although they can be treated in the exactly same way as accesses from the processor.

The term *SDRAM bus transaction*, or *transaction*, denotes a command or data transfer, such as a bank precharge, a row activate or a column access, appearing on the SDRAM address and/or data bus. A single main memory access may require one or more SDRAM bus transactions depending on the address of requested memory block and the current state of SDRAM device. It is the SDRAM controller that generates associated SDRAM bus transactions for each main memory access and sent them to the SDRAM device.

1.6 Organization

The rest of this thesis is organized as follows. Chapter 2 reviews the characteristics of modern SDRAM devices and related memory optimization techniques. Chapter 3 presents the

experimental environments and the SDRAM simulation modules added into the simulators. Chapter 4 introduces and studies the bit-reversal SDRAM address mapping, then compares it with existing address mapping techniques. Chapter 5 introduces the burst scheduling access reordering mechanism and evaluates the performance of burst scheduling by comparing it with existing access reordering mechanisms. Finally Chapter 6 draws conclusions based on simulation results and briefly discusses future work.

Chapter 2

Background and Related Work

This chapter reviews the characteristics of modern SDRAM devices, introduces main memory hierarchy, SDRAM controller policy and address bit representations. Two SDRAM optimization techniques, SDRAM address mapping techniques and access reordering mechanisms, are briefly discussed. Other related SDRAM access management and optimization techniques are also introduced.

2.1 Modern SDRAM Device

Many types of memory devices are used in computer systems. Static Random Access Memory (SRAM) and Dynamic Random Access Memory (DRAM) are two of the most important memory devices. SRAM retains its contents as long as the power is applied to the device. DRAM, however, needs to be periodically refreshed to retain the stored data. SRAM devices offer fast access times but are more expensive to produce than DRAM devices on a per bit basis. DRAM devices, on the other hand, provide large capacity but have relatively slow access times. Therefore SRAM devices are mainly used in caches where access speed is the major concern, while DRAM devices are commonly used as main memory where large address spaces are required.

Early DRAM devices were asynchronous and could not be pipeline, such as Fast Page Mode (FPM) and Extended Data Out (EDO) DRAM [17]. Modern DRAM devices are Synchronous DRAM (SDRAM), which uses a clock to synchronize its data input and output. Because of the synchronize interface, commands to the SDRAM are latched inside the device, which allows SDRAM accesses to be pipelined. SDRAM devices thus can run at faster clock speeds and deliver higher bandwidths than earlier asynchronous DRAM devices.

SDRAM dominated the DRAM market throughout the late 1990s and though 2001, then gave way to Double Data Rate SDRAM (DDR SDRAM) as well as Rambus DRAM (RDRAM) [53]. DDR SDRAM and RDRAM improve the memory bandwidth by transferring data on both the rising and falling edges of the clock signal. For example, with a 200MHz clock frequency and a 64-bit memory bus, DDR 400 (PC-3200) offers 3.2GB/s bandwidth.

DDR2 SDRAM, an successor of DDR SDRAM, offers higher clock frequency and reduced power consumption, i.e. a DDR2 800 (PC2-6400) running at 400MHz provides 6.4GB/s bandwidth. Today, as defined by Joint Electronic Device Engineering Council (JEDEC) [31], DDR and DDR2 SDRAM are most widely used as the computer main memory. This thesis focuses on memory optimization techniques which are applicable to JEDEC DDR and DDR2 SDRAM.

2.1.1 SDRAM Device Structure

SDRAM devices use memory cells to store data. Memory cells are built from storage capacitors and select transistors. Figure 2.1 shows a memory cell with a sense amplifier [67]. The sense amplifier is used to retrieve the data stored in the memory cell and shared across entire bit line. To read the data stored in a cell, first the bit line is charged to a middle value of V_{DD}, then the cell is activated by selecting the corresponding word line, which turns on the transistor and connects the capacitor to the bit line. As the capacitor is sharing

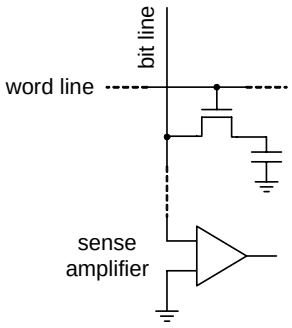

Figure 2.1: A single-transistor memory cell with a sense amplifier

the charge with the bit line, the sense amplifier detects voltage change of the bit line and distinguishes signals that represent a stored 0 or 1. Once the data is output through I/O logic, the sense amplifier restores the data to the cell as the capacitor was depleted [53].

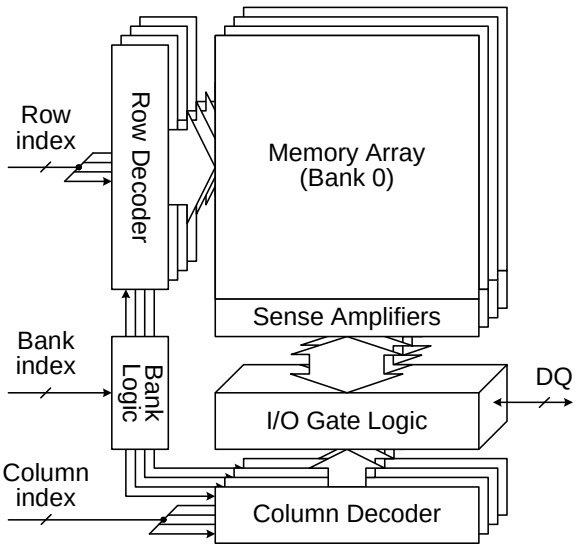

Figure 2.2: Modern SDRAM device structure

Memory cells are arranged in two-dimensional arrays, which are called *banks*. A typical JEDEC SDRAM device has 4 or 8 internal banks, as shown in Figure 2.2. Therefore, bank index, row index and column index are necessitated to locate a memory block in an SDRAM device. SDRAM devices usually have 4, 8 or 16 bits of data I/O, denoted as $\times 4$, $\times 8$ or $\times 16$.

For example, a 512Mb×8×4banks SDRAM device contains 4 banks, each of which is 8192 rows by 2048 columns. With 8-bit data I/O this device has a total capacity of 512Mbit.

Memory cells need to be periodically refreshed in order to keep the data due to the leakage currency of the capacity. This is known as refreshing, which can be simply done by a dummy read without output. According to the JEDEC standard, every bit of an SDRAM device has to be refreshed every 64ms or less to prevent data loss [42]. A refreshing logic is commonly used to refresh the DRAM device periodically. SDRAM rows can be refreshed all together every 64ms, or be refreshed row by row with a specified refresh rate.

2.1.2 Access SDRAM Device

Accessing an SDRAM device may require three SDRAM bus transactions besides the actual data transfer: bank precharge, row activate and column access [43]. A *bank precharge* prepares the selected bank by charging the bit lines. A *row activate* activates the word line selected by row index and copies the entire row of data from the array to the sense amplifiers, which function like a row cache. Then one or more *column accesses* can select the specified column data using column index and output it through I/O gate logic. To access a different row of the same bank, the bank needs to be precharged again, followed by the new row activate and column access. Figure 2.3 is a typical timing diagram of a DDR SDRAM device read [42].

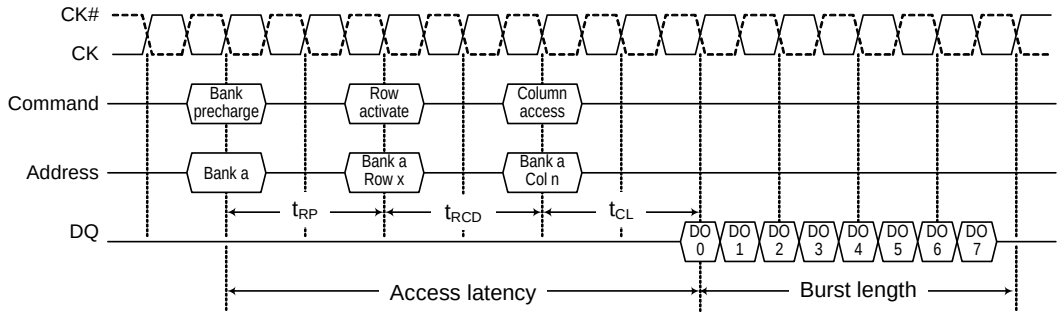

Figure 2.3: A typical DDR SDRAM read

Table 2.1: Some of the DDR/DDR2 SDRAM timing constraints

t_{CL}	Read column access to first data delay
t_{RCD}	Row activate to column access delay
t_{RP}	Bank precharge to row activate delay
t_{RAS}	Row activate to the next bank precharge delay
t_{RC}	Interval between successive row activates to the same bank ($= t_{RAS} + t_{RP}$)
t_{DQSS}	Write column access to first data delay
t_{WTR}	Internal write to read delay
t_{WR}	Write recovery time
t_{RRD}	Interval between successive row activates to different banks
t_{FAW}	No more than 4 banks may be activated in a rolling four activate window
t_{RRT}	Rank to rank turnaround cycle

SDRAM devices have a set of timing constraints that must be met for all transactions. As shown in Figure 2.3, after a bank precharge, no transaction can be performed on this bank until t_{RP} has lapsed. Similarly, t_{RCD} is the minimal time interval between a row activate and a column access. For a read, the data will appear on the data bus t_{CL} (CAS latency) cycles after the column access. Some of the DDR/DDR2 SDRAM timing constraints are listed in Table 2.1 [42].

SDRAM devices support burst mode, which allows multiple sequential data within the same row to be accessed without additional column access transactions[1]. The number sequential data in the burst mode is known as burst length, which can be set to 2, 4 or 8 for DDR; 4 or 8 for DDR2. Figure 2.3 shows a burst length of 8 and it takes 4 cycles to complete the burst due to the double data rate bus. Because main memory accesses are cache misses, memory accesses always require entire cache lines. The burst length is selected such that an entire cache line can be transferred in one burst. For an instance, a system that has a 64-bit memory bus and 64B L2 cache line size, the burst length should be 8.

[1]SDRAM's burst mode has nothing to do with the burst scheduling which will be introduced in Chapter 5.

2.1.3 SDRAM Access Latency

The *latency* of a memory access is the interval from the time when the access is issued to the time when the first word of data appears on the data bus. Given the memory bus and the SDRAM device are both idle, the access shown in Figure 2.3 has a latency of 6 cycles ($t_{RP} + t_{RCD} + t_{CL}$), which is often referred as a timing of 2-2-2 (t_{CL}-t_{RCD}-t_{RP}).

One of the most important characteristics of SDRAM devices is nonuniform access latency. Depending on the state of the SDRAM device, a memory access could be a row hit, a row empty or a row conflict and experiences different access latencies [57]. A *row hit* occurs when an access is directed to the same row as the last access to the same bank, and the bank has not been precharged since the last access. A *row empty* occurs when the bank has been precharged since the last access to this bank. Accessing a different row as the last access to the same bank results in a *row conflict*. Figure 2.4 shows examples of row hit, row empty and row conflict, assuming both the memory bus and the SDRAM device are idle when the access is issued.

As illustrated by Figure 2.4, a row hit only requires a column access because the row data accessed by the last access is still in the sense amplifiers (row cache). A row empty requires a row activate to copy the row data into the sense amplifiers followed by a column access to access the data. A row conflict, however, necessitates all three transactions. Obviously row hits have the shortest access latency while row conflicts have the longest latency. Therefore, SDRAM devices are not truly random access memory and have nonuniform access latencies. Table 2.2 summarizes the possible SDRAM access latencies given no contentions on the SDRAM buses.

Table 2.2: Possible SDRAM access latencies

Row hit	t_{CL}	Same row as the last access to the same bank
Row empty	t_{RCD}+t_{CL}	The bank has been precharged since the last access
Row conflict	t_{RP}+t_{RCD}+t_{CL}	Different row as the last access to the same bank

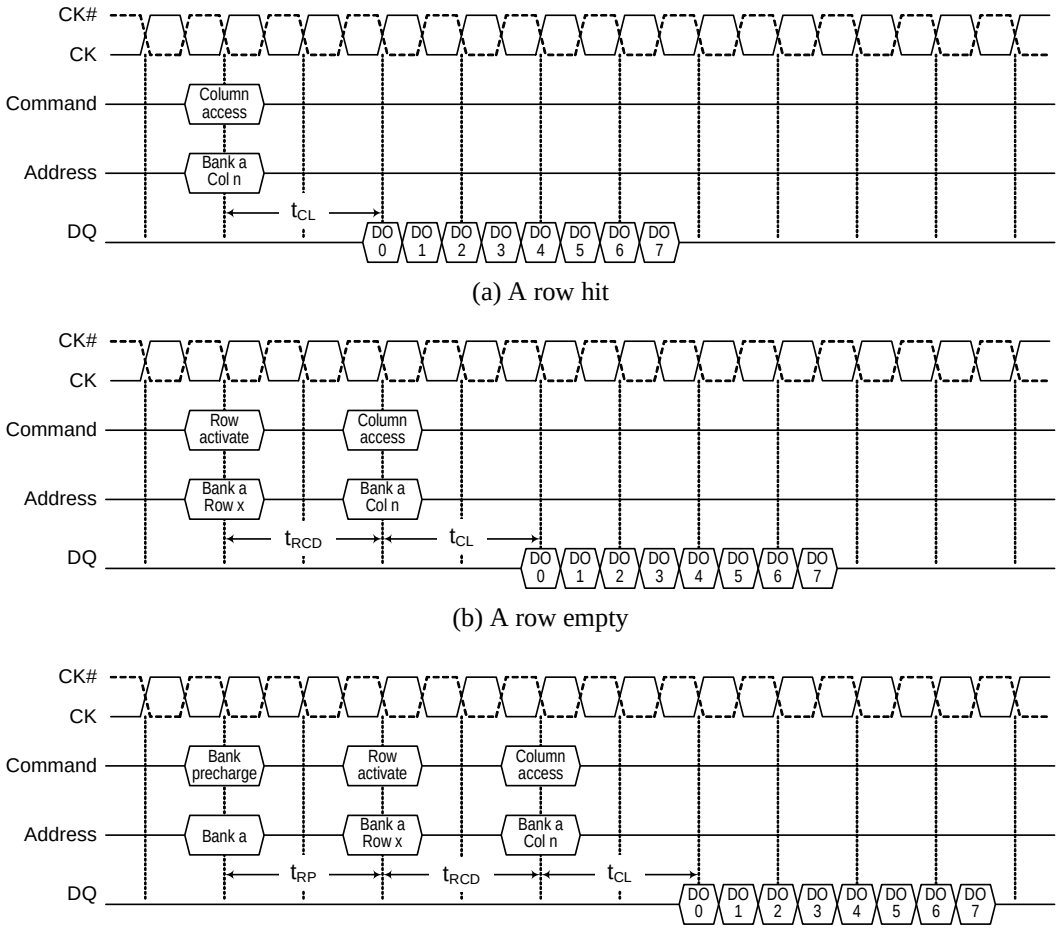

Figure 2.4: Examples of row hit, row empty and row conflict

There are enhanced DRAM devices, such as CDRAM and EDRAM [9, 48], which have an SRAM cache on the DRAM chip to reduce access latency, especially row conflict latency. Unlike the sense amplifiers of JEDEC DRAM, whose contents are lost during a bank precharge, the row data can still be accessed from the SRAM cache when the enhanced DRAM device is being precharged [72]. However DRAM caching techniques require modifications to DRAM devices. Enhanced DRAM devices are beyond the scope of this thesis.

2.2 Locality and Parallelism

As introduced in Section 2.1.3, row hits have the shortest access latency. If temporally successive accesses all directed to the same row of the same bank, they will result in successive row hits. This is referred to *SDRAM row locality*. An example is shown in Figure 2.5. Three accesses go to the same row. The first access is a row empty, while the next two accesses are row hits. The data transactions of all three accesses are transferred back to back on the data bus, resulting in the maximal data bus utilization.

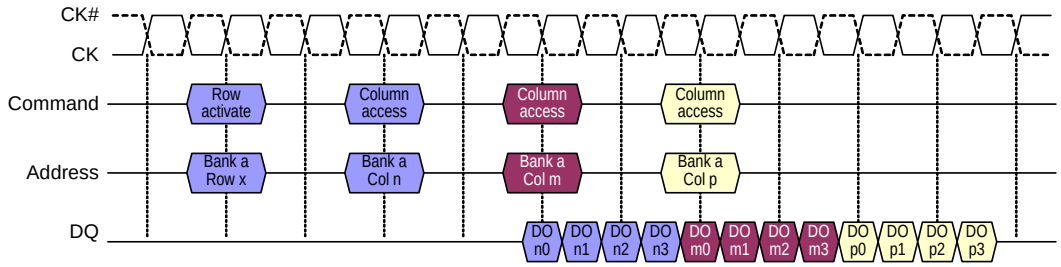

Figure 2.5: SDRAM row locality (DDR with 2-2-2 timing shown)

SDRAM row locality is a direct result of both temporal locality and spatial locality of memory access streams. When the same part of a row is reused (temporal locality), or when adjacent data of the same row is accesses (spatial locality), the sense amplifiers may be able to capture these localities as long as the requested row data is still in the sense amplifiers. Many memory optimization techniques attempt to exploit SDRAM row locality to reduce access latency and improve bus utilization, including SDRAM address mapping techniques and access reordering mechanisms which will be studies in this thesis.

SDRAM devices usually have 4 or 8 internal banks. Accesses to unique banks can be pipelined. This is known as *bank parallelism*. Figure 2.6 illustrates two row conflicts directed to different banks. Because of the available bank parallelism, it is not necessary to wait for the completion of the first access before starting the second one. While bank a is being precharged, bank b can start the bank precharge too as long as both bank precharges do not

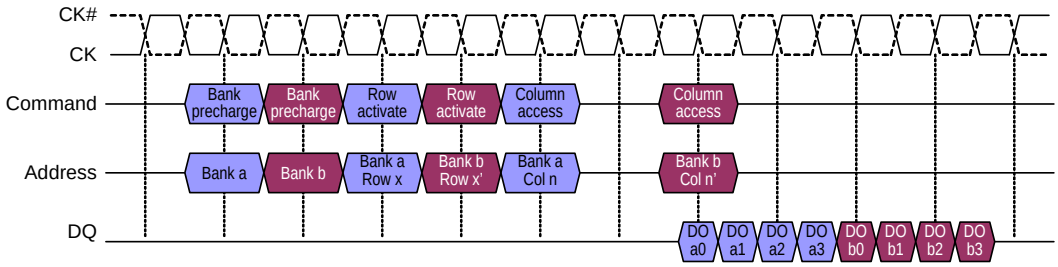

Figure 2.6: SDRAM bank parallelism (DDR with 2-2-2 timing shown)

conflict on the bus and all timing constrains are met. Similarly other transactions such as row activate can also be interleaved between banks. Although the absolute access latencies of both accesses do not reduce by interleaving, the overall execution time certainly reduces because a large part of the latency of the second row conflict is hidden by the first access.

This ability to interleave accesses between multiple unique banks is supported by many systems through BIOS [71]. Techniques such as SDRAM address mapping and intelligent data placement attempt to exploit bank parallelism. Research work presented in this thesis is to exploit both row locality and bank parallelism with the goal of reducing the observed access latency and/or overall execution time.

2.3 SDRAM Controller and Controller Policy

Main memory requests issued by the CPU typically contains the address of requested memory block, the size of the block and type of the request (read or write). SDRAM devices can not process these memory requests directly. Thus *SDRAM controllers*, also known as *memory controllers*, are used to manage the flow of data going to and from the memory. Traditionally SDRAM controllers are located on the motherboard's north bridge, while some modern microprocessors, such as AMD's Athlon 64 Processor [5], IBM's POWER5 [26, 62], and Sun Microsystems UltraSPARC T1 [44], have the memory controller integrated on the CPU die to reduce memory access latency.

SDRAM controllers contain the logic necessary to read and write SDRAM devices. To locate a memory block in the SDRAM space, an SDRAM controller translates the address of requested memory block into the SDRAM address, which is composed of channel, rank, bank, row and column index [58]. With the considerations of timing constrains and possible bus contentions, the controller generates transactions to access the SDRAM device. For reads, the SDRAM controller returns the requested data to the CPU; for writes, it updates the requested memory block in the SDRAM device with the new data. Also, the SDRAM controller periodically refreshes the SDRAM devices to prevent data loss due to capacitor leakage current.

After completing a memory access, the SDRAM controller selects whether to leave the accessed row open or close the row by performing a bank precharge. One of two static SDRAM controller policies, *Close Page Autoprecharge* (CPA) and *Open Page* (OP), typically makes this selection [71][2].

When OP policy is used, the accessed row data remains in the sense amplifiers after completion of a memory access. As discussed in Section 2.1.3, if the next access to this bank goes to the same row, then a row hit with a latency of t_{CL} occurs. Otherwise, a row conflict with a latency of $t_{RP} + t_{RCD} + t_{CL}$ occurs. With OP policy, row empties only happen after SDRAM refreshing, because banks are automatically precharged after each refreshing.

When CPA policy is used, the accessed row is closed immediately by an autoprecharge after an access completes. The next access to this bank, whether it goes to the same row or not, results in a row empty which has a fixed latency of $t_{RCD} + t_{CL}$. Thus all accesses are row empties with CPA policy. Table 2.3 summarizes the possible access latencies with OP or CPA policy, assuming there are no contentions on the buses and SDRAM devices are idle.

Controller policy has impacts on system performance. Obviously OP policy is more

[2]Historically the term page is referred to SDRAM row.

Table 2.3: Possible SDRAM access latencies with various controller policy

	Row hit	Row empty	Row conflict
Open Page	t_{CL}	$t_{RCD} + t_{CL}$	$t_{RP} + t_{RCD} + t_{CL}$
Close Page Autoprecharge	N/A	$t_{RCD} + t_{CL}$	N/A

useful for applications that have significant locality in main memory access stream, while CPA policy is more suitable for applications that do not have or have little locality. Please note, because main memory accesses are filtered by caches, CPA policy is also applicable to applications which have significant locality but most of which are captured by caches.

It is possible to apply different controller policies for each memory access. However, making the correct selection of maintaining an open row or precharging the bank requires knowledge of future access to this bank. Techniques such as SDRAM controller policy predictors can be used to predicate whether the next memory access would be a row hit or a row conflict based on history information [74]. Therefore the controller can apply the appropriate controller policy and reduce the average memory access latency at the cost of increased complexity.

2.4 Main Memory Hierarchy

SDRAM controllers and SDRAM devices compose main memory. SDRAM devices are organized to create an SDRAM address space. SDRAM controllers provide a memory interface through which the CPU or other devices (i.e. DMA controllers) can issue memory requests and receive responses.

SDRAM devices usually have 4, 8 or 16 I/O pins, while the system memory bus width is 64-bit for most PCs. A set of identical SDRAM devices are concatenated to create an SDRAM *rank* to fill the memory data bus. For example, eight 512Mbx8 SDRAM devices compose a rank that has a 64-bit data bus and a total of 512MB capacity. Used as main memory, commodity SDRAM devices are commonly packaged and sold in Dual In-line

Memory Module (DIMM). A DIMM has the same data bus width as the system memory bus. Depending on configurations, a single DIMM may comprise up to 4 ranks. Multiple ranks (either intra a DIMM or across differ DIMMs) share the same address bus as well as data bus, and are selected by chip select (CS) signals.

To meet the increasing memory demands, modern chipsets or microprocessors support multiple SDRAM channels, such as NVIDIA's nForce4 SLI (Intel Edition) [49], Intel's 965 Express Chipset [28], AMD's Athlon 64 Processor (939-pin package) [5]. An SDRAM channel is composed of one or more ranks. Different channels have unique address and data bus, therefore accesses to different channels can be performed simultaneously. SDRAM devices also allow multiple accesses to different internal banks to be interleaved, as introduced in Section 2.2. However, at each memory cycle only one transaction can be executed on the shared SDRAM buses. In contrast, multiple SDRAM transactions can happen on different channels in the exactly same clock due to separate address/data buses. Therefore multiple SDRAM channels provide great parallelism at the cost of duplicating address bus, data bus and related control logic.

2.5 Address Bits Representation

Due to the different structures and memory devices used, TLB, cache(s) and main memory have distinct interpretations of address bits. Figure 2.7 illustrates the interactions between the TLB, cache(s) and main memory. The addresses shown in the figure have high-order bits to the left. The TLB translates a virtual address to a physical address, which is then interpreted as a cache address and used to access the cache. The fields of cache address are determined by the cache organization. The cache address shown in Figure 2.7 represents a N-way set associative cache [23]. If a cache miss occurs, a memory request of the missing cache line is issued to the main memory.

The cache address or physical address is translated into the SDRAM address at main

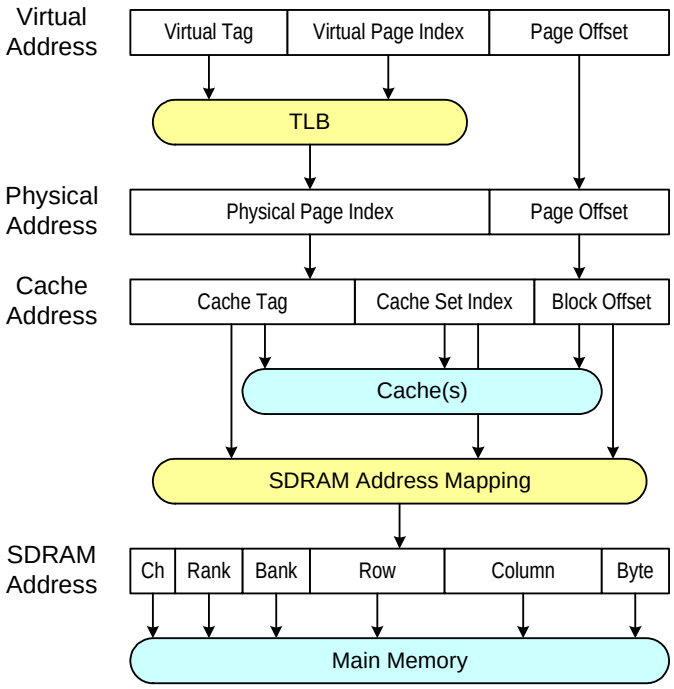

Figure 2.7: Address bit representations at different memory hierarchies

memory level. An SDRAM address is composed of channel, rank, bank, row, column and byte index. Channel index chooses the SDRAM channel and may not be available in systems that do not have multiple SDRAM channels. Rank index selects the rank within that channel. Bank, row and column index are then used to access the SDRAM devices of the selected rank as described in Section 2.1.2. The byte index allows a particular byte of the data that is transferred on the memory bus to be selected. However, due to the access granularity, which is equal to the lowest level cache line size, the byte index is usually fixed at zero.

The organization shown in Figure 2.7 serializes the TLB and cache. The TLB must be accessed before the cache can be accesses. A virtual indexed cache allows the accessing of the TLB and the cache to be performed in parallel to reduce the overall latency [59].

2.6 SDRAM Address Mapping

As discussed in Section 2.5, when a memory access is issued to the main memory, the memory address (physical address) needs to be interpreted into an SDRAM address in terms of channel, rank, bank, row, column and byte index. This process of translating physical addresses to SDRAM addresses is known as *SDRAM Address Mapping*.

Many possible ways exist to do SDRAM address mapping. The one shown in Figure 2.8 is called *flat* address mapping, which maps the channel, rank, bank index, row, column and byte index from the highest order address bits to the lowest order bits. Flat address mapping is instinctive, using higher order address bits for larger components in the memory system. Consequently, memory blocks are lineally allocated in SDRAM address space. For example, once $row0$ of $bank0/rank0/channel0$ is filled up, it goes to $row1$ of $bank0/rank0/channel0$, then $row2$ of $bank0/rank0/channel0$, so on and so forth.

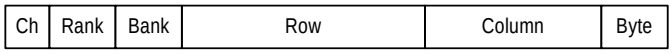

Figure 2.8: Flat SDRAM address mapping

Flat address mapping makes sense only for truly random access memory, where accessing any two locations in the memory space results in the exactly same latency. However, SDRAM devices are not truly random access memory and have non-uniform access latency as introduced in Section 2.1.3. Also, applications commonly exhibit hot spots where data are accessed frequently, such as the stack or a lookup table. Performance could be improved if those hot spots were allocated at locations where accesses to hot spots would experience shorter latencies than at other locations.

SDRAM address mapping techniques, besides the functionality of translating physical addresses into SDRAM addresses, could be used to change the allocation of memory blocks in SDRAM address space. By exploiting available locality and parallelism in main mem-

ory, SDRAM address mapping techniques can reduce average memory access latency and improve system performance.

2.6.1 Existing Address Mapping Techniques

There are existing address mapping techniques from previous studies [69, 58, 66, 21, 77, 72]. Industrial address mapping solutions can also be found in motherboard chipsets [27, 70]. This section briefly introduces some of them.

2.6.1.1 Page Interleaving Address Mapping

Page interleaving, also known as *page mode interleaving*, is one commonly used address mapping technique [69, 66]. Early SDRAM devices were slow, so it was necessary to interleave data across several SDRAM banks to obtain adequate bandwidth for the processor [14]. As shown in Figure 2.9, the bank index is mapped from the middle order address bits, while channel and rank index are still mapped from the higher order address bits.

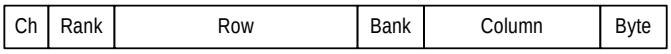

Figure 2.9: Page interleaving address mapping

With page interleaving, memory blocks adjacent in the physical address space are mapped into the first rows of internal banks within a rank, then the second rows of the banks within the same rank; once all banks of a rank are filled up, allocation continues to the next rank. By this way, memory blocks are actually stridden across different banks intra ranks. For example, memory blocks are allocated in the order of $row0/bank0/rank0$, $row0/bank1/rank0$, $row0/bank2/rank0$, ..., $row1/bank0/rank0$, $row1/bank1/rank0$, so on and so forth.

2.6.1.2 Rank Interleaving Address Mapping

Rank interleaving is similar to page interleaving except that memory blocks are stridden across banks not only intra ranks but also inter ranks [58]. Figure 2.10 illustrates rank interleaving address mapping. The rank index is mapped from address bits right above bank index. Rank interleaving increases the possibility of pipelining accesses to different banks crossing different ranks. One disadvantage of rank interleaving is that it may increase the number rank-to-rank turnaround cycles required by DDR2 devices [30].

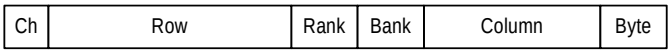

Figure 2.10: Rank interleaving address mapping

2.6.1.3 Address Mapping for Direct Rambus DRAM

Wei-fen Lin et al. pointed out that address mappings affect performance significantly and proposed an address mapping scheme for Direct Rambus DRAM (DRDRAM) [21].

DRDRAM provides a 1.6GB/s bandwidth on a single 16-bit device running at 400MHz clock frequency with double data rate. Each DRDRAM device has multiple banks, allowing pipelining and interleaving of accesses to different banks. DRDRAM device's split control buses (a 3-bit row bus and a 5-bit column bus) allows the memory controller to send commands to independent banks concurrently [13].

Similar to JEDEC SDRAM, DRDRAM memory space has coordinates of channel, device (equivalent to rank in JEDEC SDRAM), bank, row and column. Channel index is mapped from lower order bits to create a single wider logical channel from several physical channels. Then column, bank and device index are mapped from the next lowest order bits to higher order bits, allocating memory blocks contiguously into rows, then stripping across devices and banks. The highest bits are used as row index. This address mapping is further improved by XORing the device and bank index with the lower bits of row index, as well

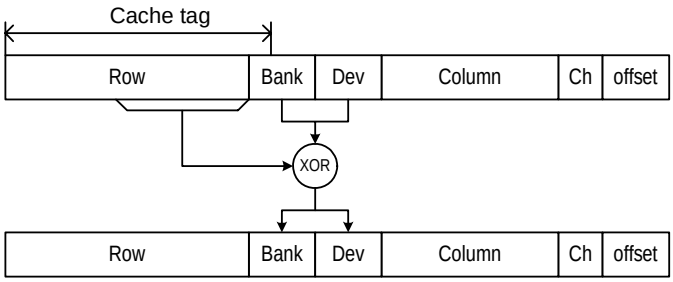

Figure 2.11: Direct Rambus DRAM address mapping

as changing the bit order of bank index to create a more randomized bank interleaving.

2.6.1.4 Permutation-based Page Interleaving Address Mapping

As an improvement to page interleaving address mapping, Zhao Zhang et al. proposed the permutation-based page interleaving address mapping for JEDEC SDRAM [77]. As shown in Figure 2.12, the permutation-based page interleaving XORs bank index with a range of row index bits which correspond to the lowest bits of the L2 cache tag in a cache address. This permutation-based address mapping attempts to convert row conflicts mainly due to L2 conflict misses into accesses to different banks, therefore reduce row conflict rate and preserve the spatial locality in SDRAM rows.

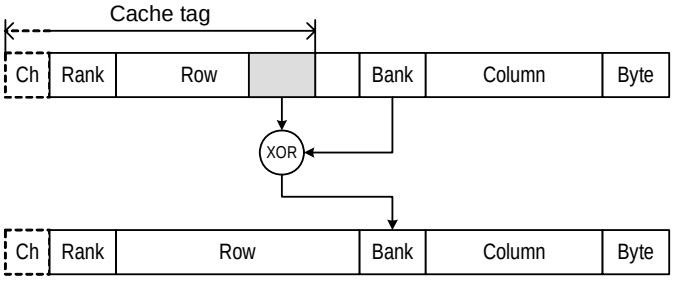

Figure 2.12: Permutation-based page interleaving address mapping

2.6.1.5 Intel 925X Chipset Address Mapping

Intel's 925X chipset uses an SDRAM address mapping method similar to page interleaving. Figure 2.13 shows one address mapping of 925X chipset as the width of row and column index may vary with different SDRAM devices. In addition to mapping bank index from the middle order address bits, a subset of bank and row index bits are reordered [27]. Intel's 925X chipset supports two dual channel modes, dual asymmetric mode and dual symmetric mode. With dual asymmetric mode, channel index is mapped from the highest address bit, leaving two independent channels. In dual symmetric mode, bit{6} is used as the channel index. Given a 128B cache line size, each cache line strides on two channels. Two 64-bit channels are equivalent to a single 128-bit channel in dual symmetric mode.

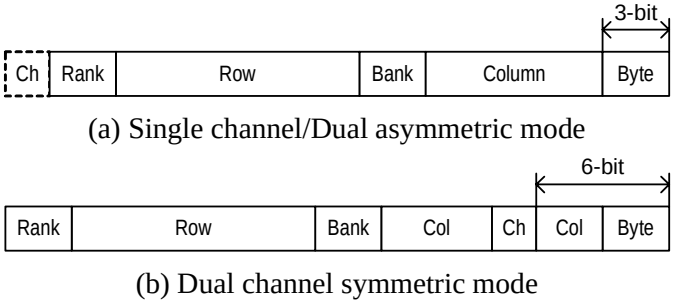

Figure 2.13: Intel 925X chipset address mapping

2.6.1.6 VIA KT880 North Bridge Address Mapping

Figure 2.14 shows the address mapping method used in VIA's KT880 north bridge [70]. The bank index is permanently mapped from address bit{14:13}. As the width of column and row index may vary with different SDRAM devices, row index may be mapped from two segments of address bits right above or below bank index. KT880 also supports dual memory channel. In dual channel mode, the channel index is mapped from address bit{3}. Therefore, each 16-byte memory block strides on two 64-bit channels, also resulting in a 128-bit logical channel.

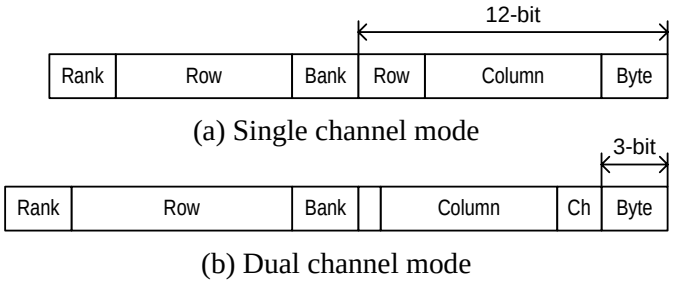

Figure 2.14: VIA KT880 north bridge address mapping

2.6.2 Dual Channel Configurations

Page interleaving, permutation-based page interleaving as well as rank interleaving were originally designed for signal channel systems. They can be used in dual channel systems by simply mapping the channel index from the highest address bit. However, using the highest address bit as channel index may result in an unbalanced load to each channel, leaving parallelism between channels unexploited.

Intel's 925X chipset and VIA's KT880 north bridge both support symmetric dual channel configuration, with which a lower order address bit is used as the channel index to interleave memory blocks between two channel at a small granularity. Symmetric dual channel configuration is equivalent to a logical channel which has the data bus width doubled. The advantage of a wider data bus is the bandwidth. Data are transferred faster on a wider bus. However the access latency may not benefit directly from a wider data bus. Two independent channels, such as dual asymmetric mode of Intel's 925X chip set, are able to start a new access immediately while one channel is serving the previous access. Two independent channels enable parallelism therefore can reduce access latency. Appendix A presents a comparison between single channel, dual asymmetric channel and symmetric channel.

2.7 Access Reordering Mechanism

Similar to an out-of-order execution processor, which executes subsequently independent instructions when the current instruction is pending due to a cache miss, main memory accesses can also be scheduled out of order to hide long latency or avoid row conflicts. *Access Reordering Mechanisms* select accesses from available outstanding memory accesses and schedule them in an order that yields the minimal overall execution time.

2.7.1 Memory Access Scheduling

As introduced in Section 2.1, SDRAM devices use a synchronized interface to synchronize the input and output signals with the clock. Most address bus transactions, such as bank precharge or row activate, are done in a single clock cycle, while data transactions may take more than one cycles depending upon the access granularity (burst length) and the bus data rate. SDRAM buses are split transaction buses, which mean transactions belonging to different accesses can be interleaved. *Memory access scheduling* is how the SDRAM controller schedules bus transactions on the SDRAM buses for each access and handles possible bus contentions.

Figure 2.15 gives an example of scheduling four outstanding memory accesses. In this example, *access*0 and *access*1 are row empties; *access*2 and *access*3 are row conflicts. In Figure 2.15(a), a naive SDRAM controller performs these accesses in the same order as they arrive and does not interleave any transactions. Assuming the SDRAM device in this example has a timing of 2-2-2 (t_{CL}-t_{RCD}-t_{RP}) and a burst length of 4 (2 cycles with double data rate), it takes 28 SDRAM cycles to complete four accesses.

In Figure 2.15(b), the same four accesses are executed by an out of order scheduling SDRAM controller. *Access*3 is scheduled prior to *access*1, which turns *access*3 from a row conflict to a row hit because *access*0 and *access*3 are for the same row of the same bank. The controller also attempts to interleave and overlap transactions of different accesses to

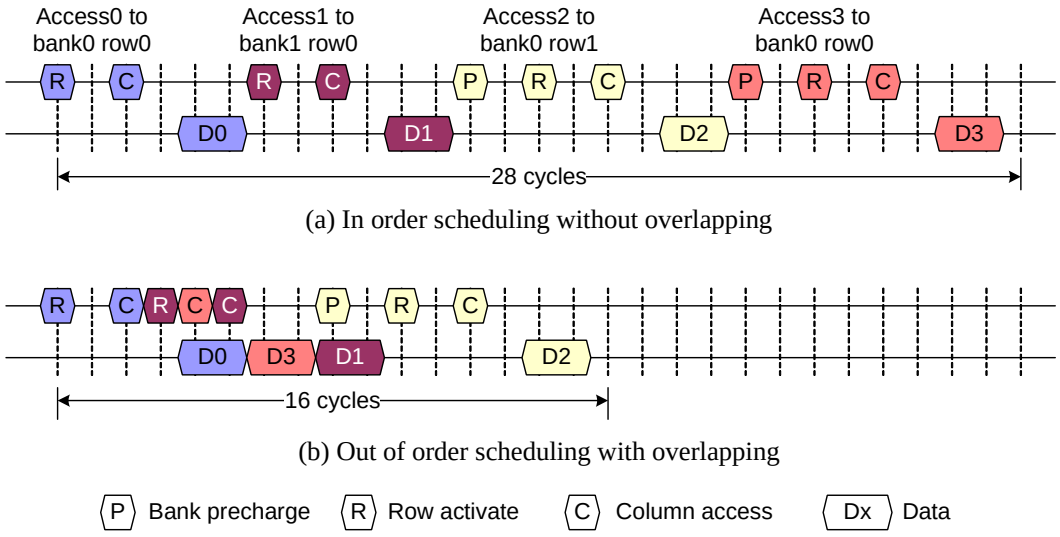

Figure 2.15: Memory access scheduling

maximize the utilization of the SDRAM buses. With the out of order scheduling SDRAM controller, it only takes 16 SDRAM cycles to complete the same four accesses. Therefore, memory access scheduling has significant impacts on system performance, especially for memory intensive applications.

2.7.2 Design Issues

As introduced in Section 1.1, SDRAM bandwidth improves by 25% each year, mainly due to the increase clock frequency, while SDRAM latency only reduces by 5% per year. Table 2.4 compares the specifications of a DDR PC-2100 SDRAM with that of a DDR2 PC2-6400 SDRAM. The actual latency for a row conflict reduces from 45ns to 37.5ns, however, because the clock frequency improves faster, the access latency in terms of memory clock cycles increases from 6 cycles to 15 cycles. Compared to bandwidth, memory latency is a bigger issue that needs to be concerned. Fortunately, the increased access latency (in cycles) leaves more performance improvement space to memory optimization techniques, such as access reordering mechanisms.

Table 2.4: PC-2100 DDR SDRAM vs. PC2-6400 DDR2 SDRAM

	DDR PC-2100 SDRAM	DDR2 PC2-6400 SDRAM
Bus clock	133MHz	400MHz
Bandwidth (64-bit bus)	2.13GB/s	6.4GB/s
Timing (t_{CL}-t_{RCD}-t_{RP})	2-2-2 (15ns-15ns-15ns)	5-5-5 (12.5ns-12.5ns-12.5ns)
Latency (row conflict)	6 cycles (45ns)	15 cycles (37.5ns)

As access latency increases, so does the temporal interval between transactions belonging to the same accesses. Consequentially the address bus becomes sparse and there are fewer contentions on the address bus. The SDRAM data bus becomes more critical. Therefore, how to improve the data bus utilization is one of the major design goals of access reordering mechanisms.

Traditionally access reordering mechanisms are implemented with SDRAM controllers located in north bridges. Some modern microprocessors have integrated memory controllers, from which access reordering mechanisms could benefit. Due to a tighter connection between the on-die SDRAM controller and the CPU, more instruction level information, such as the number of dependent instructions, can be obtainable to the SDRAM controller. This information may help the controller to make intelligent scheduling decisions. In addition, an integrated SDRAM controller can run at the same speed as the CPU. By taking advantage of the clock multiplier between CPU clock and memory clock, an integrated controller can take multiple CPU cycles to make a scheduling decision which is due in one memory clock, making complex scheduling algorithms feasible.

2.7.3 Existing Access Reordering Mechanisms

While a naive memory controller using in order scheduling is easy to implemented, it is clearly inefficient as illustrated in Figure 2.15. Previous studies have proposed access reordering mechanisms for stream-oriented systems [56, 25, 41, 55], web servers [54], network processors [22], embedded systems [36, 33] and other applications [79, 39, 46, 34].

2.7.3.1 Bank In Order Scheduling

As introduced in Section 2.1, modern SDRAM devices provide multiple internal banks. Accesses to different banks are allowed to be executed in parallel as long as all timing constrains are met. *Bank in order scheduling* takes advantage of this bank parallelism [56]. It has a structure as shown in Figure 2.16. Composed by unique memory access queues for each bank and a global arbiter, bank in order scheduling can start another access from a different bank when the current access is being scheduled. Therefore access between banks may be scheduled out of order.

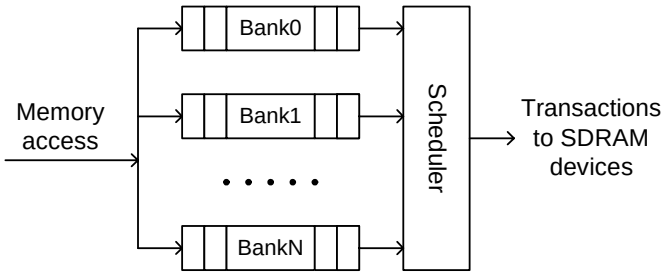

Figure 2.16: Bank in order memory scheduling

Outstanding memory accesses are stored at memory access queues depending on the bank they are directed. Bank queues are based on first come first served, which means accesses within the same bank are served in the same order as they arrive. The global arbiter then selects one access from all bank queues using a bank selection policy. A round robin or an oldest first bank selection policy usually works well. While bank in order scheduling exploits bank parallelism by interleaving accesses between different banks, accesses within the same banks are still served in order.

2.7.3.2 The Row Hit Access Reordering

One improvement to bank in order scheduling is to use priority queues instead of FIFO queues. With priority queues, accesses within the same bank can be scheduled out of order

as well. Figure 2.17 shows a *row hit access reordering*, which uses priority bank queues and a row hit first policy [56]. At each bank, the oldest access that goes to the same row as the last access is assigned the highest priority and scheduled first. Similar to bank in order scheduling, bank selection policies, such as round robin, are still applicable when the arbiter needs to select accesses from different banks.

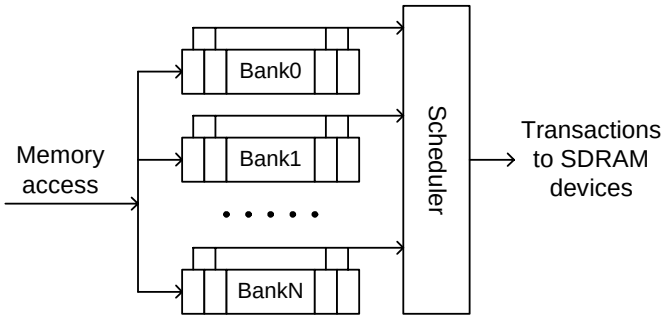

Figure 2.17: The row hit access reordering

The row hit access reordering attempts to create as many row hits as possible out of available accesses. Because row hits have the shortest access latencies, the row hit access reordering will have a better performance than bank in order scheduling.

2.7.3.3 Memory Access Scheduler for Virtual Channel SDRAM

Virtual Channel SDRAM provides a set of channel buffers within the SDRAM to hold segments of rows for faster access and greater concurrency than conventional SDRAM [47]. Scott Rixner et al. proposed a memory access scheduler, which has a similar structure to the row hit access reordering as shown in Figure 2.17, for Virtual Channel SDRAM [54].

While most bank queue reordering policies such as row hit first, and bank selection policies such as round robin can still apply to Virtual Channel SDRAM, the scheduler must be augmented with a channel selection policy to exploit the features of Virtual Channel SDRAM devices.

Access latency is reduced both by increasing parallelism within conventional SDRAM and by increasing the segment hit rate exclusive for Virtual Channel SDRAM. With proposed access reordering policies, their memory access scheduler achieves high memory bandwidth and low memory latency for modern web servers.

2.7.3.4 Adaptive History-based Memory scheduler

Proposed by Ibrahim Hur et al., the adaptive history-based memory scheduler tracks the access pattern of recently scheduled accesses and selects memory accesses matching the program's mixture of reads and writes [25].

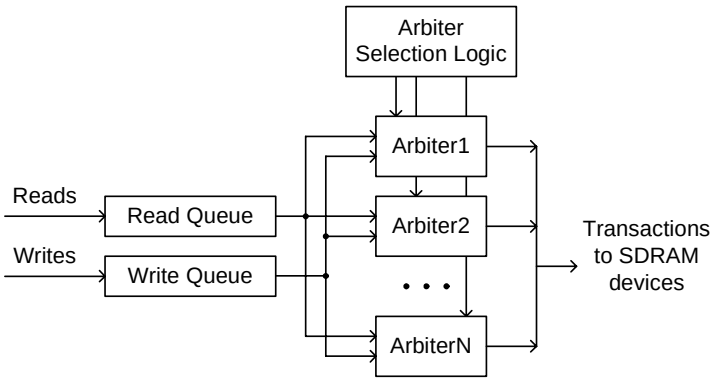

Figure 2.18: Adaptive history-based memory scheduler

As illustrated in Figure 2.18, outstanding accesses are stored at a global read queue and write queue. A set of arbiters select accesses from the read queue or the write queue with the goal of matching some expected mixtures of reads and writes based on history information. For example, if the arbiter has a history of $write|read|read$ and the expected read write ratio is 1:1, then the arbiter will select a write from the write queue. When multiple qualified accesses are available, the arbiter uses the expected latency as the second criterion to make final decisions.

Each history-based arbiter is optimized for one particular read write ratio. Therefore

only one arbiter is working at any time. An arbiter selection logic observes the recent program access pattern and periodically chooses the most appropriate history-based arbiter.

2.7.3.5 Stream Memory Controller

Sally McKee et al. described a Stream Memory Controller (SMC) system for streaming computations. The SMC system combines compile time detection of streams with execution time access reordering to exploit the existing memory bandwidth [41, 24].

The SMC system detects streams at compile time, then selects access and issues access at execution time. The SMC effectively prefetches read streams, buffers write streams, and reorders accesses to exploit the existing memory bandwidth as much as possible. Unlike most other hardware prefetching or stream buffer designs, the SMC system does not increase bandwidth requirements. The SMC can be implemented using existing compiler technology and requires a modest amount of special purpose hardware. The SMC system is designed for applications performing streaming memory accesses and Fast Page Mode DRAM and Rambus DRAM.

2.7.3.6 Fine-grain Priority Scheduling

Zhichun Zhu et al. proposed fine-grain priority scheduling for Direct Rambus DRAM systems [79]. Memory accesses are split and mapped into different channels and critical data are returned first, to fully utilize the available bandwidth and concurrency provided by DRDRAM.

Requested memory blocks are split into sub-blocks with minimal granularity, which are then mapped into different channels. Sub-blocks that contain the desired data are marked as critical ones with higher priorities therefore are returned earlier than non-critical sub-blocks. This approach also allows critical sub-blocks of one access to bypass non-critical sub-blocks from other accesses.

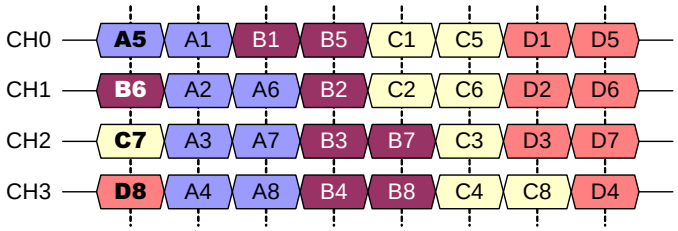

Figure 2.19: Fine-grain priority scheduling

An example of a 4-channel memory systems processing four accesses is shown in Figure 2.19. Each memory block is split into eight sub-blocks, and the four critical sub-blocks (the boxes with bold letters) are mapped to different channels. With fine-grain priority scheduling, all critical sub-blocks finish earlier than non-critical sub-blocks, reducing latencies to critical data.

2.7.3.7 Access Reordering for Network Processors

SDRAM access reordering and prefetching were also proposed by Jahangir Hasan et al. to increase row locality and reduce row conflict penalty for network processors [22]. Network processors (NP) are programmable microprocessors optimized for packet switches. NPs implement a variety of packet processing functions, such as IP forwarding, in software. Thus memory bandwidth is a key consideration in the design of packet switching platforms.

They propose a piece-wise linear allocation that attempts to allocate memory space for contemporaneously arriving packets in the same row. To enhance row locality, SDRAM accesses that are likely to go to the same row are made consecutively in small groups (e.g., 4 accesses), without being intervened by other accesses. For the row misses that occur despite the allocation and reordering, prefetching is used to overlap a row conflict access with the preceding access to hide the latency, given the two accesses go to different banks.

2.7.3.8 Intel's Out of Order Memory Access Scheduling

Intel has a patented memory access scheduling algorithm that performs read and write memory accesses out of order to improve SDRAM buses utilization and gain performance over a range of workloads [57]. Figure 2.20 shows the structure of Intel's out of order memory access scheduling.

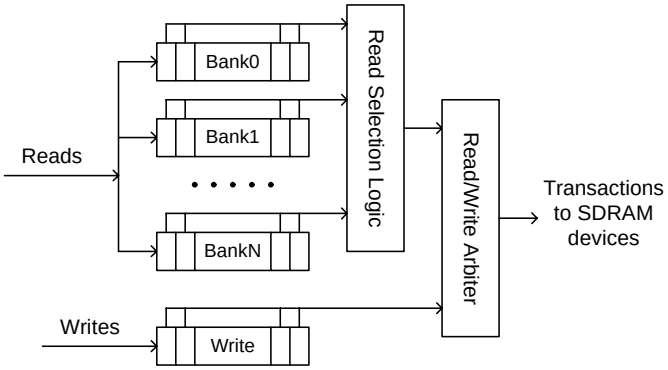

Figure 2.20: Intel's out of order memory access scheduling

In their design, read queues have a similar structure to the row hit scheduling as shown in Figure 2.17. But writes are stored at a separate write queue to allow reads to bypass writes. Read selection logic uses a complex algorithm to make decisions. Basically those already started but unfinished accesses, such as a row conflict whose bank precharge and row activate have been performed, have the highest priority. Any row hit accesses have the next highest priority.

Write scheduling is out of order as well. When possible, writes from other banks are inserted between the back to back write row conflicts. Writes to the same row are clustered. A read/write arbiter then prioritizes between reads and writes. Reads generally have higher priorities over writes to optimize read latency. Preempting an already started but unfinished write, such as a write that has performed bank precharge and/or row activate but has not started column access yet, with a later arrived read can result a better performance.

2.8 Other SDRAM Optimization Techniques

This section briefly introduces some other SDRAM related techniques, which also attempt to improve SDRAM performance by exploiting locality and parallelism.

2.8.1 The Impulse Memory System

Proposed by John Carter et al., the *Impulse Memory System* adds two features to a traditional memory controller. First, an extra stage of address translation is added into the memory controller to allow applications to control how their data is accessed and cached. Second, the Impulse controller supports prefetching at the memory controller, which reduces the effective latency to memory [10, 76].

Real systems usually do not use all physical address space, i.e. in a system with 4GB of physical address space having only 1GB memory populated. The Impulse system uses these unused addresses constitute a *shadow address space* which is mapped to physical memory by the Impulse controller. By giving applications and the operating system control over the use of shadow addresses, Impulse supports application-specific optimizations that restructure data.

As an example, consider a program that accesses five elements of an array. Given the physical memory layout of the requested elements shown in Figure 2.21(a), a conventional memory system loads the five elements in five separate memory accesses, each of which contains a full cache line of contiguous physical memory. The five elements then occupy five cache lines, although only a subset of each line is requested.

In an Impulse memory system, an application can configure the memory controller to export a dense shadow space alias that contains just the requested elements, and have the OS map a new set of virtual addresses, which fall into the same cache line. The application can then access the elements via the virtual alias in a single memory access as shown in Figure 2.21(b). In this way the five requested elements only occupy one cache line and

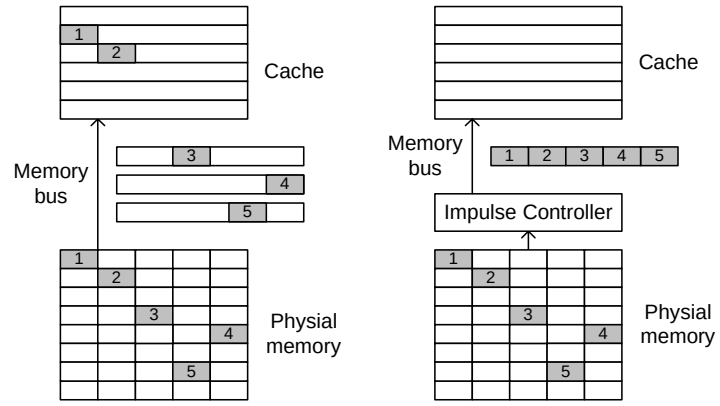

Figure 2.21: The Impulse Memory System

require only one bus transfer.

The Impulse memory controller also supports prefetching by adding a small amount of SRAM on the controller. For non-remapped data, prefetching is useful for reducing the latency of sequentially accessed data. The Impulse memory system is designed for applications that do not exhibit sufficient locality, e.g. sparse matrix, database and multimedia applications.

2.8.2 SDRAM Controller Policy Predicator

SDRAM controller policy is introduced in Section 2.3. Two static controller policies, CPA and OP, are commonly used and can be selected through many BIOS [71]. However, which policy yields a better performance largely depends upon an applications' access pattern. A dynamic controller policy which applies different policies to each access can reduce access latency.

An ideal dynamic controller policy, referred to as Dynamic Upper Bound, will be presented in Section 3.2.1.3. The dynamic upper bound policy uses future information only available in simulations to give an upper bound of performance improvement that a real dynamic controller policy can achieve without reordering accesses [58].

While dynamic upper bound policy requires future access information, a two-level dynamic SDRAM controller policy predictor proposed by Ying Xu uses a history based predictor to make the decision of leaving the accessed row open or precharging the bank after completing each access [74]. This controller policy predictor is similar to branch predictors [40, 75], making predictions using history information.

2.8.3 Adaptive Data Placement

When virtual paging systems are in use, the performance of a virtual paging system is often evaluated by how fast a virtual page can be allocated or freed. However, how fast a page can be accessed during the runtime also impacts performance, especially when the main memory has nonuniform access latencies.

SDRAM address mapping techniques, as will be presented in Chapter 4, can change access distribution in the SDRAM address space to exploit parallelism. However, SDRAM address mapping must be static and does not reflect any dynamic changes in program behavior.

The operating system, provided with the knowledge of the memory hierarchy, can intelligently place data in the SDRAM space to exploit parallelism available in the main memory. Theoretically an intelligent virtual paging system can achieve at least the same performance improvement as that of SDRAM address mapping techniques at the cost of operating system page allocation complexity. In addition, the virtual paging system may have the ability to change the data placement as program access patterns change [37].

Chapter 3

Methodology

This chapter describes the simulation environments and methodologies used in the thesis. Two modified simulators, SimpleScalar v3.0d and M5 v1.1, are used for SDRAM address mapping and access reordering mechanisms studies respectively. Decisions and considerations in selecting simulators are discussed. Modifications made to the simulators in order to support the studies are presented.

3.1 Methodologies Used in the Thesis

Simulators are widely used in computer architecture and microprocessor studies [35, 2, 1]. Using these architecture simulators is an efficient way to study memory organizations and optimization techniques. However, many simulators focus on microprocessor studies, such as pipeline or cache organizations, and use simplified main memory modules, which may not be accurate for memory studies.

The selected simulators are revised to replace the original main memory modules with more detailed SDRAM modules. The techniques being studied, including the proposed and existing techniques, are implemented in simulation modules and added into the simulators. Using standard benchmarks, these techniques are simulated and examined. Trace files gen-

erated by the simulators are used to validate the implementations. Further improvements are made based on analysis of simulation results.

3.2 SimpleScalar Simulation Environment

SimpleScalar, selected for use in the SDRAM address mapping technique studies, is a well established and widely used computer architecture simulator in academic research [35].

SimpleScalar performs fast, flexible, and accurate simulation of modern processors. It is an execution-driven simulator, featuring a detailed, out-of-order issue, superscalar processor simulator that supports nonblocking caches and speculative execution. It executes the PISA among other ISAs, which is a close derivative of the MIPS architecture [52]. Precompiled binaries for the SimpleScalar architecture exist, including SPEC CPU95 and SPEC CPU2000 benchmark suites [8, 11]. Therefore, SimpleScalar is selected for SDRAM address mapping techniques studies.

As distributed, SimpleScalar v3.0d uses a simplified main memory module which has a fixed access latency for all main memory accesses. Due to the implementation, access latencies have to be determined at the time when memory accesses are issued, which is not an issue so long as main memory accesses are scheduled in order such that access latencies are deterministic. However, the fixed-latency main memory does not represent any real memory systems and requires modifications to support the studies of SDRAM address mapping techniques. The following section presents a detailed SDRAM simulation module designed for SimpleScalar v3.0d, which not only allows the studies of SDRAM address mapping but also other memory related techniques.

3.2.1 SDRAM Simulation Module v1.0

The SDRAM module v1.0 is developed to replace the original main memory module provided with SimpleScalar v3.0d. As a timing simulation module, the major functionality of

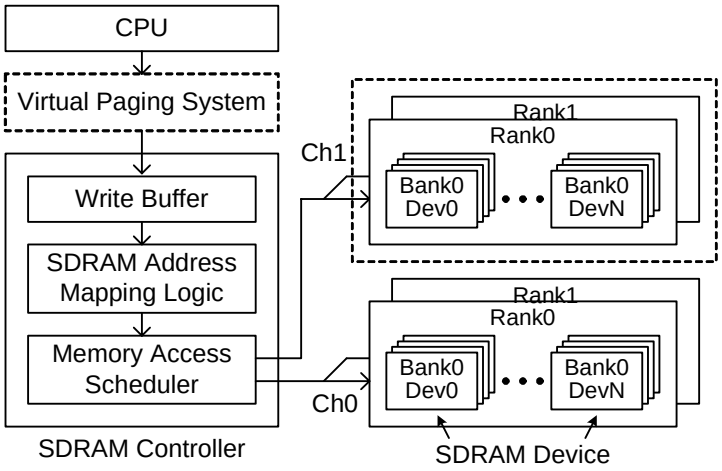

Figure 3.1: SDRAM Module v1.0 for SimpleScalar

the SDRAM module v1.0 is to determine the access latency at the time when a memory access is issued. The access latency is calculated based on the address of the requested memory block, the current state of SDRAM devices and bus availability. These considerations are absent from the original main memory module used by SimpleScalar v3.0d. The SDRAM module v1.0 is a purely timing module, which does not provide any of the data storage and retrieval functionalities of main memory. Functional memory continues to be handled by SimpleScalar. The SDRAM module v1.0 is composed of SDRAM device modules, an SDRAM controller module and an optional virtual paging system, as illustrated in Figure 3.1.

3.2.1.1 SDRAM Device Module

The SDRAM device module represents a typical JEDEC SDR/DDR SDRAM device. It is fully parameterized, including many of the parameters present in an SDRAM device, including capacity, clock frequency, data rate, I/O width, row size, column size and timing constraints (t_{RP}, t_{RCD}, t_{CL}, etc). Some of these timing constraints are listed in Table 2.1.

A state machine is used to keep track of the current state of each bank. According to

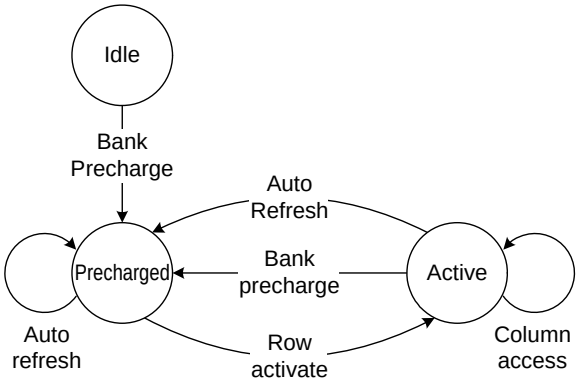

Figure 3.2: Bank state transition diagram

the bank state transition diagram given in Figure 3.2, all banks are reset to `idle` state after initialization. Each bank precharge or auto refresh will send the bank to the `precharged` state. The state is transited to `active` by a row activate, where one or more column accesses can perform reads or writes.

3.2.1.2 SDRAM Controller Module

The SDRAM controller module provides a main memory interface to the CPU, accepts and schedules main memory requests issued by the CPU in a manner similar to that of a hardware memory controller. As introduced in Section 2.4, multiple SDRAM devices are concatenated to form a rank to fill the data bus of the memory interface. Ranks share address/data buses and are selected by chip select (CS) signals. Composed by one or more ranks, channels have unique address/data buses as well as control signals. As shown in Figure 3.1, the SDRAM controller module further contains three submodules. They are SDRAM address mapping logic, memory access scheduler and write buffer.

- *SDRAM address mapping logic* translates the physical address of each memory access into the SDRAM address in terms of channel, rank, bank, row, column and byte index using the selected SDRAM address mapping technique, as introduced in Section 2.6.

- *Memory access scheduler* generates required SDRAM bus transactions for each memory access according to the current state of the target SDRAM device. Transactions are then scheduled and sent to the device. Unlike the naive in order scheduler as shown in Figure 2.15(a) which schedules transactions strictly in order, the scheduler used by the SDRAM module v1.0 attempts to interleave and overlap transactions between adjacent accesses whenever possible, while it maintains the actual data transactions in the same order as accesses arrive. This scheduler is still considered as an in order scheduler, however it is more efficient than a naive in order scheduler and is used in SDRAM address mapping studies.

- A limited-size *write buffer* replaces the infinite write buffer originally used by SimpleScalar v3.0d [63]. The write buffer offers three features. First, it hides the write latency so that writes can complete immediately. Second, the write buffer allows reads to bypass writes to reduce read latency. Third, if a read requests the same memory block as a previous write, the write data can be forwarded from the latest write to the read, therefore reducing the traffic to the main memory. In the presence of a write buffer, reads must search the write buffer for possible read hits before accessing the main memory to avoid data hazard. The write buffer performs writebacks when the write buffer reaches its capacity or when the SDRAM bus has been idle for a certain number of memory cycles.

3.2.1.3 Dynamic Upper Bound Controller Policy

The SDRAM controller makes the decision, logically at the end of each access, whether or not leave the accessed row open. This decision making process is known as applying controller policies. The SDRAM controller module supports two static controller policies, OP and CPA, as introduced in Section 2.3.

In order to support future studies of dynamic controller policies, a dynamic controller

policy called *Dynamic Upper Bound* (DYN-UPB) is added into the SDRAM module v1.0. With DYN-UPB policy, the controller keeps the accessed row open and postpones the decision until the next access to this bank is available. If the subsequent access to the same bank is known to be a row conflict, then the controller goes back in time and precharges the bank by modifying the state of that bank. Obviously DYN-UPB policy requires the knowledge of future access information which is only available in simulations. The purpose of introducing DYN-UPB policy is to predict an upper bound of performance improvement that a dynamic controller policy could achieve under the constraints of the simulation environment, specifically access sequence.

3.2.1.4 Virtual Paging System

Virtual paging system determines the placement of data in the physical address space and allows programs that require a larger amount of memory than the size of physical memory to run by swapping pages [60]. There is no virtual paging system implemented in SimpleScalar v3.0d as it does not simulate the entire operating system. Given single thread benchmarks, enough main memory (no page swapping) and fixed main memory access latency, virtual paging system does not have impacts on the performance. However, with a detailed nonuniform main memory, such as what the SDRAM module v1.0 represents, page placements will result in different access latencies. A preliminary virtual paging system is therefore implemented and added into SimpleScalar v3.0d.

The virtual paging system translates virtual addresses into physical addresses using an inverted page table [60]. As a purely functional module, this simulated virtual paging system does not generate any timing information. It does not have a Translation Look-aside Buffer (TLB) either. In addition, assuming simulated machines have enough physical memory for each benchmark, there will be no page swapping or page faults expect for initial page allocations.

Two paging allocation algorithms are implemented, Sequential Allocation and Random Allocation. With *sequential allocation*, pages are allocated sequentially from the beginning of the physical address space. *Random allocation* on the other hand allocates pages completely randomly. While sequential allocation is more like a typical machine shortly after booting up, random allocation models a machine running for a long time with high memory utilization, in which pages are randomized located in the physical address space.

3.2.2 Revised SimpleScalar Baseline Machine Configuration

Table 3.1 lists the configuration of the baseline machine used for SDRAM address mapping studies. A revised SimpleScalar v3.0d and the SDRAM module v1.0 are used. The baseline machine is representative of a typical desktop PC with the memory controller located on the north bridge. The simulated CPU is a 2.4GHz 4-way out-of-order execution processor with 128KB split L1 cache and 512KB unified L2 cache. The simulated SDRAM controller has a memory access buffer big enough to hold the maximal number of outstanding memory accesses supported by the LSQ and RUU. Main memory uses 512Mbit technology (64M×8) DDR 400 (PC-3200) SDRAM device with timing parameters of 3-4-4-8 (t_{CL}-t_{RCD}-t_{RP}-t_{RAS}), which mimics the Micron MT46V64M8 DDR SDRAM [42]. Burst length is set to 8 such that an entire L2 cache line (64-Byte) can be loaded in each main memory access. The baseline machine has a single SDRAM channel with 4 ranks, providing a 3.2GB/s memory bandwidth. A total of 2GB of main memory is simulated.

The baseline machine uses OP controller policy, leaving the accessed row open after each access. There is no virtual paging system for the baseline machine and accesses are scheduled in order (not naive in order), as described in Section 3.2.1.2, to isolate the contribution of SDRAM address mapping from other techniques. The impacts of controller policies and virtual paging systems will be studied in Section 4.4.1 and Section 4.4.2 respectively. Chapter 5 will be presenting access reordering mechanisms.

Table 3.1: Revised SimpleScalar baseline machine configuration

SimpleScalar v3.0d configuration	
CPU	2.4GHz, 4-way, out-of-order execution, 16 RUU, 8 LSQ
L1 cache	64KB I-cache and 64KB D-cache, 2-way, 32B cache line
L2 cache	512KB, 16-way, 64B cache line
SDRAM module v1.0 configuration	
Front Side Bus	64-bit, 200MHz (400MHz data rate)
Main memory	2GB DDR 400 (PC-3200) SDRAM, 3-4-4-8, 64-bit data bus, burst length 8
Channel/Rank/Bank	1/4/4
Virtual paging	Not available
Controller policy	Open Page
Address mapping	Page interleaving
Write buffer size	16 entries

3.3 M5 Simulation Environment

First introduced in 2003, the M5 simulator is a modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture [7, 6]. Version 1.0 was released in 2005 and the current version is 2.0.

Written in C++ and Python, M5 is object oriented. Major simulation structures (CPUs, buses, caches, etc.) are represented as objects and interchangeable. M5 features a detailed model of an out-of-order SMT-capable CPU (Alpha ISA) and an event-driven memory system including non-blocking caches and split-transaction buses. M5 is capable of full-system simulation. It models a DEC Tsunami system in sufficient detail to boot unmodified Linux/FreeBSD. M5 is also capable of multiprocessor or multi-system simulation [2].

3.3.1 Switching from SimpleScalar to M5 Simulator

A revised SimpleScalar v3.0d is used for SDRAM address mapping studies as shown in Section 3.2. When memory accesses are scheduled in program order, the latency of each access is determinate at the time when the access is issued. SimpleScalar uses an event-

driven mechanism to invoke the main access process function provided by the SDRAM module when a main memory access is requested. The SDRAM module schedules the access immediately, calculates and return the access latency.

Consider a memory controller that can schedule memory access out of order, an access received later may be scheduled earlier than its predecessors. The access latency (of the first access) is therefore not determined at the time when the access is issued by the CPU. In order to perform access reordering, following requisites must be satisfied. First, the main memory has to be non-blocking so that new memory requests can continue to be issued to the main memory while previously issued requests are still pending. Second, memory requests need to be separated from responses. For example, after issuing a memory request, the CPU will be notified by a separate response, which contains the requested data as well as other information such as the access latency of that access.

Although it is not impossible to modify SimpleScalar v3.0d to support access reordering through introducing callback functions, it certainly requires significant amount of effort to make callback functions work with SimpleScalar's event-driven mechanism. M5 simulator v1.1 features split-transaction buses. Memory requests can be easily separated from responses, thus the CPU does not need to wait for the response before it can issue another memory request. Meanwhile M5 simulator executes Alpha binaries so that it can share benchmarks with SimpleScalar. These features of M5 simulator make it an ideal simulator for studies of access reordering mechanisms. The M5 simulator's Syscall Emulation (SE) mode (non-full-system) is used for the studies of access reordering mechanisms to be presented in Chapter 5.

Unfortunately M5 simulator v1.1 still uses a fixed-latency main memory module. An all new SDRAM module v2.0 is therefore developed for M5 v1.1, featuring out-of-order memory access scheduling and supporting DDR2 SDRAM devices. The following sections present the details of the SDRAM module v2.0.

3.3.2 SDRAM Simulation Module v2.0

The SDRAM module v2.0 is an update from the SDRAM module v1.0 and designed for M5 v1.1. Unlike the SDRAM module v1.0 which is driven by memory access events, the SDRAM module v2.0 is driven by its own clock events. The SDRAM clock is divided from the CPU clock by a clock multiplier. The SDRAM module v2.0 is still a purely timing simulation module, which means it only provides timing information. Main memory functionality is provided by M5's functional memory.

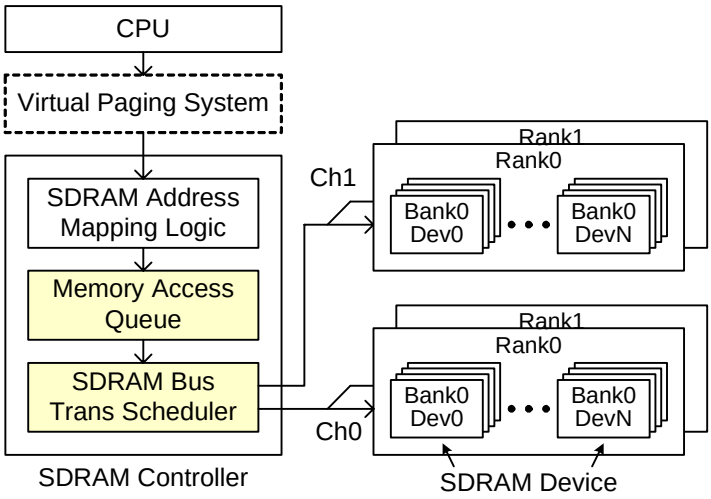

Figure 3.3: SDRAM Module v2.0 for M5 Simulator

Figure 3.3 illustrates the structure of the SDRAM module v2.0, which is composed of an optional virtual paging system, an SDRAM controller module and SDRAM device modules. Inside the SDRAM controller module, there are the SDRAM address mapping logic, the memory access queue and the SDRAM bus scheduler.

Virtual paging system and SDRAM address mapping logic are identical to that of the SDRAM module v1.0. Major changes to the SRAM device module include the support for DDR2 SDRAM devices, an SDRAM power consumption module which estimates the power consumption of main memory. The memory access queue is new in the SDRAM module

v2.0 and will be discussed below. The write buffer used in the SDRAM module v1.0 is now integrated into the memory access queue, which can achieve the same functionality as the write buffer. The in order memory access scheduler, presented in Section 3.2.1.2, is replaced by a sophisticated SDRAM bus scheduler which works closely together with the memory access queue to perform various access reordering mechanisms.

3.3.2.1 Memory Access Queue

The *memory access queue* is the place where outstanding memory accesses are stored when they are waiting to be scheduled. The memory access queue has a buffer structure and associated control logic. Depending on its structure, the memory access queue can model various access reordering mechanism as introduced in Section 2.7.3.

For example, to model a bank in order scheduling as shown in Figure 2.16, the memory access queue uses one FIFO queue for each bank. Replacing the FIFO queues with priority queues and applying certain policies, such as row hit fist policy, the memory access queue can model the row hit access reordering as introduced in Section 2.7.3.2.

Figure 3.4 shows a more detailed memory access queue structure for the row hit access reordering. Each bank has an unique queue and an associated control logic, known as the *bank arbiter*. Bank arbiters use a row hit first policy, which selects the oldest access in the queue directed to the same row as the previous access of the same bank if such an access exists. The selected access is marked as the *ongoing access* of that bank.

Ongoing accesses from all banks are then forwarded to the SDRAM bus scheduler, which will be introduced in Section 3.3.2.2, to perform transaction scheduling. Once all transactions belonging to an ongoing access have been scheduled (notified by the access complete signal), a response is sent to the CPU containing the requested information as well as access latency. The complete access is removed from that bank and the bank arbiter selects another access from the bank queue to be the next ongoing access.

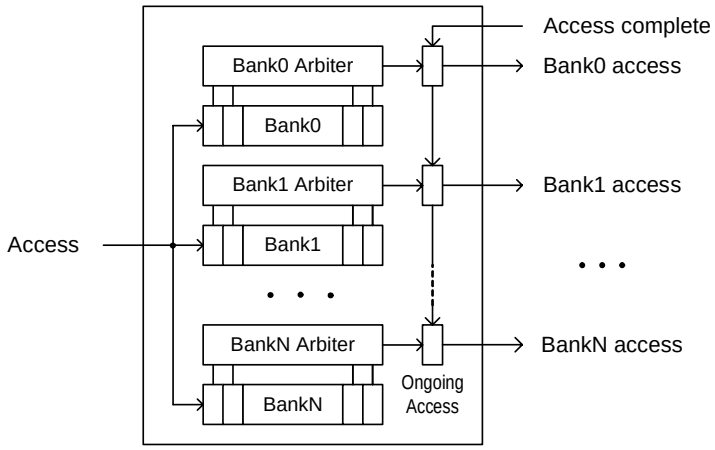

Figure 3.4: Memory access queue for the row hit access reordering

Depending on the implementation, the memory access queue could be considered as full when all bank queues are full, or any one bank queue is full, given a multiple bank queues structure as shown in Figure 3.4. In case the memory access queue is full, the SDRAM controller blocks the memory interface to the CPU so that no new accesses can be issued to the main memory.

The write buffer, used in the SDRAM module v1.0 and discussed in Section 3.2.1.2, can be integrated into the memory access queue by adding a dedicated write queue and an extra read/write arbiter. Because the memory access queue has the ability to delivery the functionality of a write buffer, the SDRAM module v2.0 does not contain a stand alone write buffer module.

Thanks to M5's object orientated structure, different implementations of memory access queues can share the same or compatible interfaces so that different memory access queue modules, representing various access reordering mechanisms, can be easily exchanged for performance comparisons.

3.3.2.2 SDRAM Bus Scheduler

While the memory access queue determines the order in which accesses within the same banks are served, the *SDRAM bus scheduler* schedules the transactions for the accesses sent by the memory access queue. The SDRAM bus scheduler performs transaction level scheduling with the considerations of bank states as well as bus utilization. Figure 3.5 illustrates an SDRAM bus scheduler that will work with the memory access queue as shown in Figure 3.4.

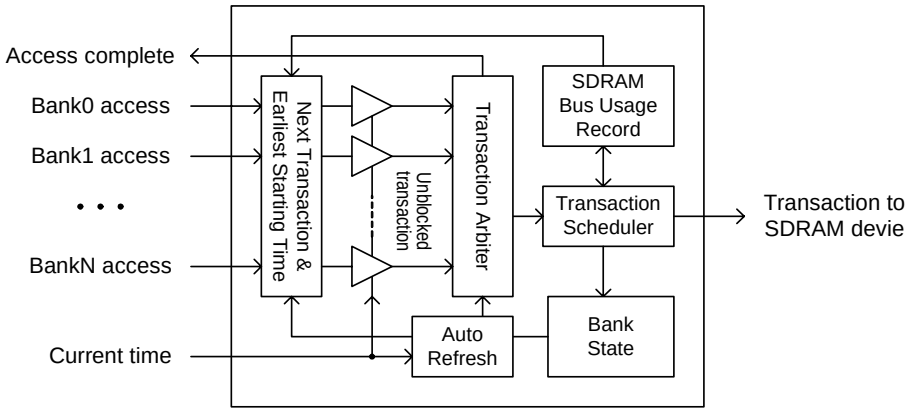

Figure 3.5: SDRAM bus scheduler

First, the SDRAM bus scheduler determinates the next transaction of each access it receives based on the current state of the target bank. For example, the next transaction of an access directed to a precharged bank would be a row activate. Then the earliest starting time of the next transaction is calculated with the considerations of all applicable timing constraints and bus usages. Any transactions whose starting times have not come are marked as *blocked*. Only unblocked transactions enter the transaction arbiter, which selects one transaction at each memory cycle based on arbiter policies. Various arbiter policies can be applied here to interleave transactions between different accesses/banks and maximize the bus utilization. Finally the selected transaction is scheduled. The target bank's state will be updated according to the bank state transition diagram as shown in Figure 3.2. The

next transaction of the access will also be updated. If the scheduled transaction is the last transaction of an access, the access complete signal is used to notify the memory access queue to select and send a new access.

An SDRAM auto refresh logic periodically generates refreshing transactions. Refreshing is done via row by row for all banks. Upon the completion of a refreshing, the bank will be precharged. Therefore refreshing may interrupt partially scheduled accesses. For example, an access whose row activate has done but the column access has not scheduled yet will be interrupted by a refreshing, because the bank precharge closes the activate row. Those interrupted accesses need to be restarted after the refreshing.

3.3.3 Revised M5 Baseline Machine Configuration

Table 3.2 lists the baseline machine configuration for the revised M5 v1.1 simulator and the SDRAM module v2.0. Representing a high-end desktop PC, the baseline machine is to be used for address reordering studies.

The CPU of the baseline machine is a 4GHz, 8-way out-of-order execution superscalar processor with a 128KB instruction cache and a 128KB data cache, as well as an unified 2MB L2 cache. The simulated machine has 4GB DDR2 800 (PC2-6400) SDRAM main memory, which is based on Micron MT47H64M8 512Mb DDR2 SDRAM device with 5-5-5-17 (t_{CL}-t_{RCD}-t_{RP}-t_{RAS}) timing [43]. Burst length 8 is selected to transfer an entire cache line in each memory access. There are two 64-bit channels, each of which consist of 4 ranks and delivers a 6.4GB/s bandwidth. The 64-bit Front Side Bus (FSB) runs at 800MHz with double data rate, resulting in a 12.8GB/s bandwidth matching for the total bandwidth provided by two SDRAM channels.

The baseline machine uses a traditional bank in order scheduling, as discussed in Section 2.7.3.1. Accesses within the same banks are scheduled in the same order as they are issued, while access between different banks may be scheduled out of order. The static OP

Table 3.2: Revised M5 baseline machine configuration

M5 v1.1 configuration	
CPU	4GHz, 8-way out-of-order execution, 32 LSQ, 196 ROB
L1 cache	128KB I-cache and 128KB D-cache, 2-way, 64B cache line
L2 cache	2MB, 16-way, 64B cache line
SDRAM module v2.0 configuration	
Front Side Bus	64-bit, 800MHz (1.6GHz data rate)
Main memory	4GB DDR2 800 (PC2-6400) SDRAM, 5-5-5-17, 64-bit data bus, burst length 8
Channel/Rank/Bank	2/4/4
Virtual paging	Not available
Controller policy	Open Page
Address mapping	Page interleaving
Access reordering	Bank in order scheduling
Memory access pool	256
Maximal writes	64

policy and conventional page interleaving address mapping are used to isolate the contributions of controller policy and address mapping techniques. Read accesses and write accesses share a memory access pool. The size of the memory access pool (256) is chosen not to create a bottleneck. Because writes need extra spaces to store the write data, a maximal of 64 writes can be hold by the memory access pool.

3.4 Benchmarks

SPEC CPU2000 is the fourth major version of the Standard Performance Evaluation Corporation (SPEC) CPU benchmark suites, which in 1989 became the first widely accepted standard for comparing compute-intensive performance across various architectures [65].

SPEC CPU2000 comprises two sets of benchmarks: CINT2000 and CFP2000. The CINT2000 comprises 12 application-based benchmarks written in C and C++ languages for measuring compute-intensive integer performance. The CFP2000 comprises 14 benchmarks written in FORTRAN (77 and 90) and C languages for measuring compute-intensive

floating point performance. The two suites measure the performance of a computer's processor, memory architecture and compiler. Compared to the previous version SPEC CPU95, CPU2000 suite offers longer run times, larger problems for benchmarks and more application diversity.

SPEC CPU2000 benchmark suite is selected and used in SDRAM address mapping and access reordering studies, except for the `sixtrack` benchmark which could not complete due to insufficient floating point precision of the simulators. Some benchmarks are not memory intensive, thus memory optimization techniques begin studied in this thesis may have little impacts on them. However, results from the rest 25 SEPC CPU2000 benchmarks will all be presented in Chapter 4 and Chapter 5 to prevent biasing the results [12]. Pre-compiled little-endian Alpha ISA SPEC2000 binaries with reference input sets are used [11]. Table 3.3 lists the all simulated benchmarks and their command line parameters.

Besides SPEC CPU2000 benchmarks, SPEC CPU95 and other benchmarks such as the STREAM are also used during the studies [64, 32]. Because these benchmarks are mainly used for testing and debugging, their simulation results are not shown in this thesis.

3.4.1 Number of Instructions to Simulate

With reference input sets, simulation times of some SPEC CPU2000 benchmarks can be extremely long, especially with the M5 simulator. Therefore only a selected number of instructions are simulated.

The choice of the number instructions to simulate is based on simulation times and how fast the caches are warmed up. The simulations should be able to complete within reasonable times, meanwhile enough instructions should be simulated to expose the major behaviors of the benchmarks. Fast forwarding, commonly used to skip cache warming up stage, is not employed in studies presented in this thesis. This is because during cache warming up stage there will be many cache misses (main memory accesses), meaning more

Table 3.3: SPEC CPU2000 benchmark suites and command line parameters

CINT2000, 11 applications written in C and 1 in C++ (252.eon)		
Name	Remarks	Input sets
164.gzip	Data compression utility	gzip.input.source 60
175.vpr	FPGA circuit placement and routing	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2
176.gcc	C compiler	166.i -o 166.s
181.mcf	Minimum cost network flow solver	mcf.inp.in
186.crafty	Chess program	< crafty.in
197.parser	Natural language processing	2.1.dict -batch < parser.ref.in
252.eon	Ray tracing	chair.control.cook chair.camera chair.surfaces chair.cook.ppm ppm pixels_out.cook
253.perlbmk	Perl	-I./lib diffmail.pl 2 550 15 24 23 100
254.gap	Computational group theory	-l ./ -q -m 192M < gap.ref.in
255.vortex	Object Oriented Database	lendian1.raw
256.bzip2	Data compression utility	bzip2.input.source 58
300.twolf	Place and route simulator	ref
CFP2000, 14 applications (6 Fortran-77, 4 Fortran-90 and 4 C)		
Name	Remarks	Input sets
168.wupwise	Quantum chromodynamics	
171.swim	Shallow water modeling	< swim.in
172.mgrid	Multi-grid solver in 3D potential field	< mgrid.in
173.applu	Parabolic/elliptic partial differential equations	< applu.in
177.mesa	3D Graphics library	-frames 1000 -meshfile mesa.in -ppmfile mesa.ppm
178.galgel	Fluid dynamics: analysis of oscillatory instability	< galgel.in
179.art	Neural network simulation; adaptive resonance theory	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10
183.equake	Finite element simulation; earthquake modeling	< equake.inp.in
187.facerec	Computer vision: recognizes faces	< facerec.ref.in
188.ammp	Computational chemistry	< ammp.in
189.lucas	Number theory: primality testing	< lucas2.in
191.fma3d	Finite element crash simulation	
200.sixtrack	Particle accelerator model	< sixtrack.inp.in
301.apsi	Solves problems regarding temperature, wind, velocity and distribution of pollutants	

performance improvement space is available to memory optimization techniques.

For SDRAM address mapping studies, the first 2^{32} (about 4.3-billion) instructions are simulated for all benchmarks. The first 2-billion instructions of CPU2000 benchmarks are simulated for access reordering studies.

3.4.2 Main Memory Access Behaviors of Simulated Benchmarks

It is desirable, and even necessary, to know the memory access pattern of the benchmarks that will be used in the studies of SDRAM address mapping and access reordering.

Figure 3.6 shows the total number of read memory accesses and write memory accesses for the first 2-billion simulated instructions of SPEC CPU2000 benchmarks (except for `sixtrack`). The data are obtained by the revised M5 simulator using the baseline machine configuration as listed in Table 3.2. Because main memory accesses are cache misses, the number of main memory accesses are significantly fewer than the number of executed load/store instructions after being filtered by the caches.

Among 25 benchmarks, `gcc`, `mcf`, `swim`, `mgrid`, `applu`, `art`, `facerec`, `ammp` and `lucas` are the most memory intensive benchmarks. Benchmark `ammp` has the most number of main memory accesses, about one main memory access every 14 instructions. These memory intensive benchmarks are expected to show more significant performance differences than other less memory intensive benchmarks when the proposed SDRAM address techniques and access reordering mechanisms are applied.

Figure 3.7 shows the read write ratio in percentage. The most accesses of `gcc` and `apsi` are writes, while `eon` and `ammp` have little writes. The average read write ratio of simulated SPEC CPU2000 benchmarks is 58/42. Read write ratio can be used in memory access scheduling to make scheduling decision as Chapter 5 will discuss. Other characteristics of main memory access stream, such as localities, will be presented in Chapter 4 during the studies of SDRAM address mapping.

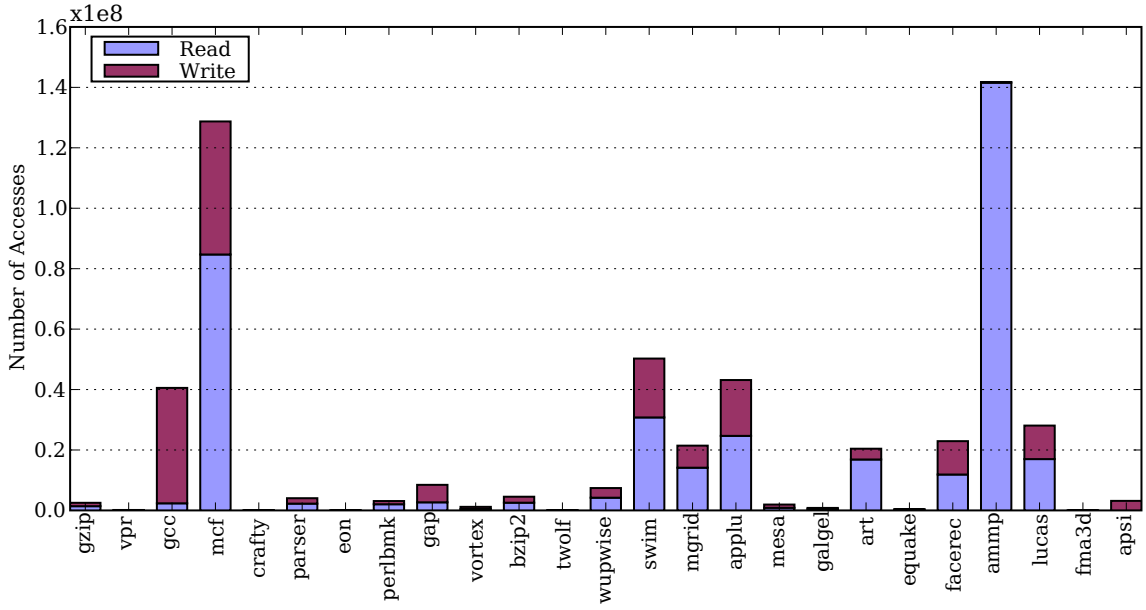

Figure 3.6: Total number of main memory accesses of SPEC CPU2000 benchmarks

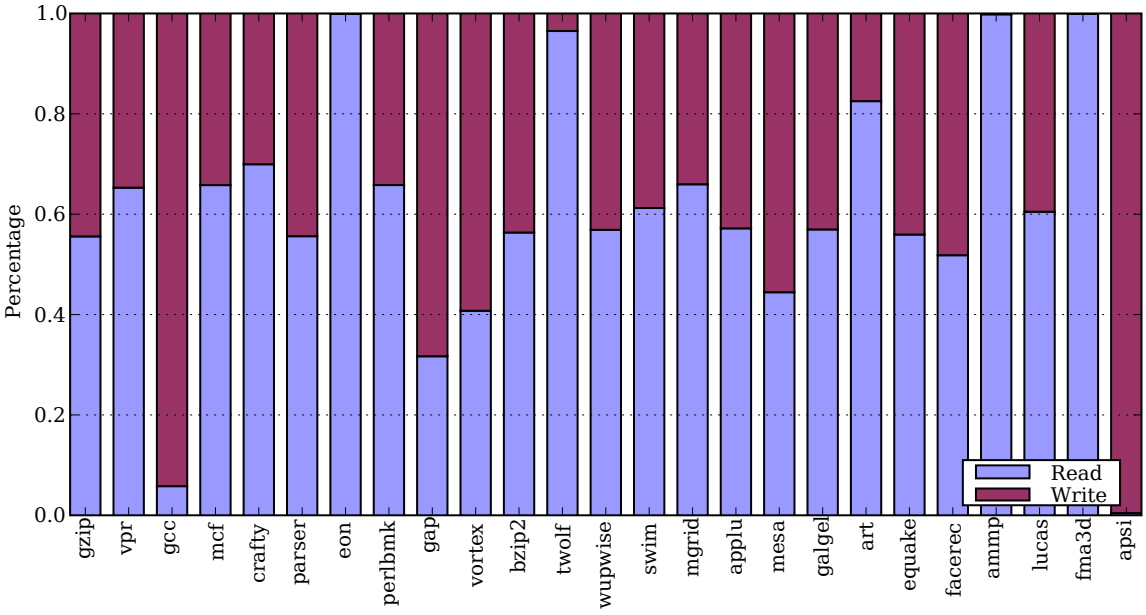

Figure 3.7: Main memory accesses read write ratio of SPEC CPU2000 benchmarks

3.5 Validation

This section briefly discusses the method which is used to validate the implementations of simulation modules and proposed techniques, as well as the method to verify simulation results.

3.5.1 Validating the Implementations

Both the SDRAM module v1.0 and v2.0 are capable of generating various trace files and detailed debug information, which are used to validate the implementations of both the SDRAM module and proposed techniques.

Figure 3.8 illustrates a segment of an SDRAM bus transaction trace file of benchmark `gzip`, which is generated by the revised M5 simulator v1.1 with the SDRAM module v2.0 using the configuration shown in Table 3.2, except that the SDRAM device has a 3-3-3 timing. The trace file records the bus transactions happened on the SDRAM address bus and data bus at each memory cycle. Numerous information are recorded, including the memory cycle, transaction type, and the information about the access which the transaction is associated with, such as access ID, type and access's SDRAM address.

This segment shows the bus usages of *channel*0 from memory clock 331 to 365. During this period of time, *access*42 to *access*46 are scheduled. *Access*42 to *access*44 are directed to the same bank (*bank*0 of *rank*0), while *access*45 and *access*46 are directed to another bank (*bank*0 of *rank*2). Because the bank in ordering scheduling is used, accesses within the same banks are scheduled in order but accesses between different banks can be interleaved. For example, *access*46 goes to a different bank as *access*43 and *access*44, the data transaction of *access*46 fits perfectly in the interval between *access*43 and *access*44.

By carefully examining the profiles of these transactions, all timing constraints are found to have been met, therefore the simulation module works correctly.

```
Tick  Access ID    Type,Rk,Bk,Rw,Co   Address bus            Access ID  Data bus
-------------------------------------------------------------------------------
 331: Access #42 (MemRead,0,0,8,1816) Precharge             Access #41 DataRead
 332:                                                       Access #41 DataRead
 333:
 334: Access #42 (MemRead,0,0,8,1816) RowActivate
 335: Access #45 (MemRead,2,0,16,1160) ColRead
 336:
 337:
 338:                                                       Access #45 DataRead
 339: Access #42 (MemRead,0,0,8,1816) ColRead               Access #45 DataRead
 340:                                                       Access #45 DataRead
 341:                                                       Access #45 DataRead
 342: Access #43 (MemRead,0,0,10,1696) Precharge            Access #42 DataRead
 343:                                                       Access #42 DataRead
 344:                                                       Access #42 DataRead
 345: Access #43 (MemRead,0,0,10,1696) RowActivate          Access #42 DataRead
 346:
 347:
 348: Access #43 (MemRead,0,0,10,1696) ColRead
 349:
 350:
 351:                                                       Access #43 DataRead
 352: Access #46 (MemRead,2,0,16,1168) ColRead              Access #43 DataRead
 353: Access #44 (MemRead,0,0,8,0) Precharge                Access #43 DataRead
 354:                                                       Access #43 DataRead
 355:                                                       Access #46 DataRead
 356: Access #44 (MemRead,0,0,8,0) RowActivate              Access #46 DataRead
 357:                                                       Access #46 DataRead
 358:                                                       Access #46 DataRead
 359: Access #44 (MemRead,0,0,8,0) ColRead
 360:
 361:
 362:                                                       Access #44 DataRead
 363:                                                       Access #44 DataRead
 364:                                                       Access #44 DataRead
 365:                                                       Access #44 DataRead
```

Figure 3.8: Example of SDRAM bus transaction trace file

3.5.2 Verifying the Simulation Results

Statistic variables are used in simulations to collect information such as execution time or row hit rate. Some statistics are made to be redundant. For example, the SDRAM module v2.0 has the statistics for the total number of accesses received, as well as the total number of occurrences of row hit, row empty and row conflict. The summary of row hits, row empties and row conflicts should be equal to the total number of access received. By doing cross checking of these redundant statistics, the simulation results are verified.

Chapter 4

SDRAM Address Mapping

SDRAM address mapping translates a physical address of a requested memory block into an SDRAM address to locate the block in the main memory. Due to SDRAM's nonuniform access latency, the locations of the memory blocks result in various access latencies, therefore SDRAM address mapping impacts the performance. This chapter studies SDRAM address mapping techniques and proposes the bit-reversal address mapping. Simulation results show that bit-reversal address mapping outperforms existing SDRAM address mapping techniques through exploiting parallelism and reducing row conflicts.

4.1 Localities in Main Memory Access Stream

Due to the manner in which programs are created, hot sports in data structures are usually exhibited, such as arrays and stacks where data are stored in consecutive locations in the memory and are likely to be accesses successively, leading to spatial locality in data references. Also data in these hot spots tend to be reused frequently, leading to temporal locality in data references. On the other hand, programs normally are executed sequentially, thus instructions are read from the memory one by one. Meanwhile, the same sets of instructions (i.e. procedures) are likely to be called again in the near future. Therefore

spatial and temporal locality exist in memory references to program instructions as well.

The memory hierarchy is designed to exploit these localities of memory references. For example, a copy of recently accessed data is kept in a cache which has a shorter access latency than the main memory. When the data is accessed again as implied by temporal locality, the cache provides a fast access to the data. Spatial locality is also exploited by the cache. When a data is loaded into the cache, adjacent data falling into the same cache line are also loaded into the cache at the same time. Therefore when these adjacent data are needed in the near future according to spatial locality, they can be accessed quickly from the cache.

With the existence of caches, main memory accesses are all cache misses from the lowest level cache[1]. Does the main memory access stream still contain temporal and spatial locality especially after being filtered by caches? The following two sections perform statistical analysis on the main memory access stream and answer this question.

For the following studies of main memory access stream localities, a revised SimpleScalar v3.0d with the SDRAM module v1.0 is used, as introduced in Section 3.2. The simulated machine has the same configuration as shown in Table 3.1. The simulator generates trace files which contain the sequence of main memory accesses of SPEC CPU2000 benchmark suite with a maximal one billion simulated instructions and reference input sets. The results presented in Section 4.1.1 and Section 4.1.2 are averaged across all simulated benchmarks.

4.1.1 Temporal Locality in Main Memory Access Stream

Memory block *reuse distance* is used here to refer to the number of accesses between two memory accesses that request to the same memory block. In other words, reuse distance presents temporal locality by showing how often a memory block will be reused.

Figure 4.1 shows distances for various sizes of memory blocks from 64-byte to 16KB

[1]Main memory accesses from other devices such as DMA controllers are not considered in this thesis, although they are just other sources of main memory access streams

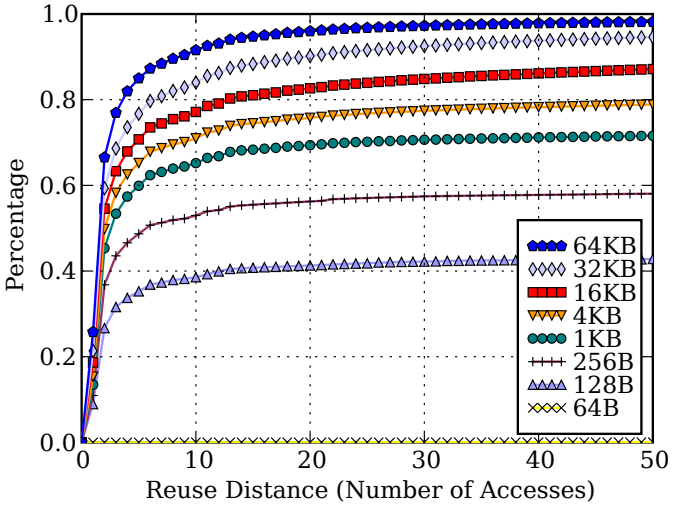

Figure 4.1: Memory block reuse distance

based on SPEC CPU2000 benchmarks. The simulated machine has 64-byte L2 cache line size and 16KB effective SDRAM row size[2]. 4KB is a typical virtual page size although virtual paging is absent for the simulations presented in this section.

For cache line size memory blocks (64B), it is highly unlikely that they will be reused within 50 accesses because they are still resident in the cache. For SDRAM row size memory blocks (16KB), approximately 77% of accessed rows will be accessed again within the next 10 accesses. 87% of accessed rows will be reused within the next 50 accesses. This illustrates that temporal locality is available in the main memory access stream even after the filtering effects of two levels of cache.

4.1.2 Spatial Locality in Main Memory Access Stream

Normally memory blocks are allocated sequentially either in the virtual address space when virtual paging is present, or in the physical address space when virtual paging is absent. So far there is no virtual paging used in the simulations, thus the term address is referred to

[2]An effective SDRAM row is a logical row concatenating the same index rows across all devices within a rank, which equals to the row size of each individual bank multiplied by the number devices in the rank.

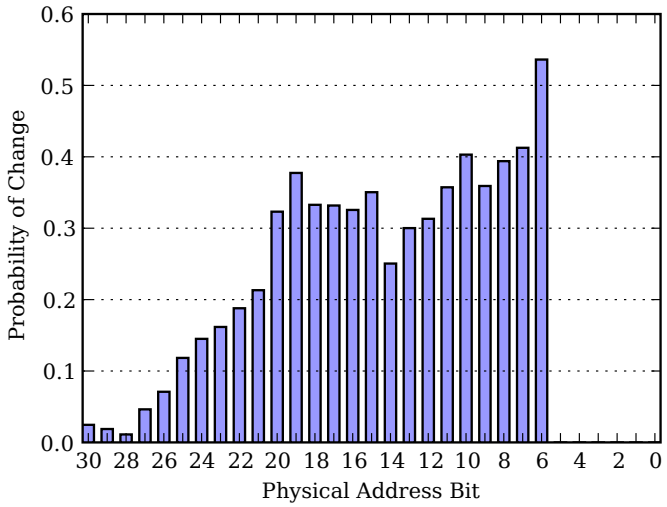

Figure 4.2: Address bits change probability between adjacent main memory accesses

a physical address unless other specified.

Due to the sequential allocation, addresses of two memory blocks that are spatially close to each other will have the most identical bits in the higher order address bits. The more different higher order address bits two memory blocks have, the more far away they are in address space.

Based on the same trace files used in Section 4.1.1, the probability of change of each physical address bit between two temporally adjacent main memory accesses is studied and shown in Figure 4.2. Due to the access granularity of 64-byte (a cache line size), the lowest 6-bit physical address bits (bit{5:0}) are fixed at zero, therefore they have zero probability to change. From Figure 4.2, the lower order physical address bits (except for the lowest 6-bit) generally have a higher statistical probability of change from access to access than the higher order address bits.

This confirms the availability of spatial locality in main memory access stream, because spatially close memory blocks are likely to be accessed temporally together and their addresses are only different in the lower order address bits.

4.1.3 Exploiting Localities with SDRAM Device

Both temporal and spatial locality are available in main memory access stream after being filtered by two level caches as shown in Section 4.1.1 and Section 4.1.2. The question is how SDRAM address mapping techniques can efficiently exploit these localities.

With flat SDRAM address mapping, as described in Section 2.6, memory blocks are allocated sequentially in SDRAM address space, filling from the smallest components (columns) to the biggest components (channels).

Using the OP policy, temporal locality and spatial locality within SDRAM rows can be captured by SDRAM sense amplifiers. As introduced in Section 2.1, sense amplifiers function like a row cache. As long as the row containing the previously accessed data is still in the sense amplifiers, subsequent memory accesses, either requesting the same data (temporal locality) or adjacent data (spatial locality), result in row hits and fast access.

However, when a program accesses data crossing the boundary of SDRAM rows, spatial locality may cause row conflicts and degrade performance. Consider if there is no spatial locality at all above SDRAM rows, the probability of an access to be a row conflict (going to a different row of the same bank as the previous access) is given in Equation 4.1, where n is the total number of banks and m is the total number of rows in each bank.

$$P_{row\_conflict} = \frac{1}{n} \times \frac{m-1}{m} \tag{4.1}$$

When spatial locality presents, the probability of row conflict is greatly increased because adjacent rows within a bank are likely to be accessed temporally together.

The nonuniform characteristic of SDRAM device results in a situation where the objective is to maintain temporal and spatial locality within SDRAM rows meanwhile destroy spatial locality above SDRAM rows. SDRAM address mapping techniques, especially the bit-reversal address mapping to be presented in the next section, preserve SDRAM row

locality and convert harmful spatial locality into bank parallelism which can be exploited to reduce access latency through interleaving.

4.2 Bit-reversal Address Mapping

Based on the observation that the lower order physical address bits have higher probabilities of change for temporally adjacent accesses than the higher order bits, and the discussion about locality with SDRAM in Section 4.1.3, the bit-reversal SDRAM address mapping is hereby proposed [58].

Bit-reversal address mapping maps channel, rank, bank and the higher order bits of row index in a reversed order from the highest order physical address bits, as illustrated in Figure 4.3. Channel index is not available in a single channel configuration. Compared to the flat address mapping, as shown in Figure 2.8, bit-reversal address mapping adds an extra step on top of flat address mapping, which reverses the v-bit highest physical address bits, then interprets the reversed physical address to an SDRAM address in the same manner that flat address mapping does.

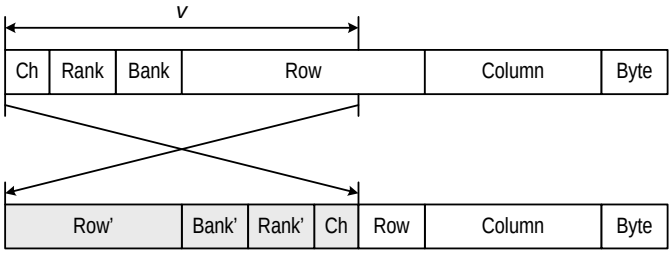

Figure 4.3: Bit-reversal SDRAM address mapping

The total number of reversed bits v is called the *depth of reversal*, which will be shown to have impacts on performance in Section 4.3.1. The lower order bits of row index, column index and byte index are mapped from the lowest order physical address bits at the same positions.

4.2.1 Philosophy of Bit-reversal Address Mapping

The objective of bit-reversal address mapping is to achieve the greatest reduction in access latency through changing access distribution in SDRAM address space.

Observations of Section 4.1.2 show that the lower order physical address bits change more frequently than higher order physical bits from access to access. These lower order physical bits which have the highest probability of change are mapped as column index, as most existing address mapping methods do. While row index are mapped from those physical address bits that are mostly unlikely to change. As a result, with high probabilities, the addresses of two adjacent accesses only differ on the column indexes. When that happens these two accesses fall in to the same row, resulting row hits and short access latency.

The address bits that have the next highest probability to change are used as indexes for components bigger than row, i.e. channel, rank and bank. Therefore adjacent accesses, which do not fall into the same row as described above, are most likely to be directed into different channels/ranks/banks. Especially with multiple channels, distributing accesses to different channels balances the load to each channel.

Bit-reversal address mapping preserves row hits thus locality within SDRAM rows can be captured by SDRAM sense amplifiers. For potential row conflicts, bit-reversal address mapping distributes them into different banks, therefore harmful spatial locality above SDRAM rows are converted into bank parallelism, which can be exploited to reduce access latency through interleaving accesses to different banks.

4.2.2 Hardware Implementation

The hardware implementation of bit-reversal address mapping is straightforward. In an embedded system with a fixed amount SDRAM installed and a separate memory controller, bit-reversal address mapping can be accomplished with routing traces between the microprocessor and the memory controller on the PCB. In a desktop system designed to support

flexible SDRAM modules and various processors, the memory controller can manage bit-reversal address mapping. The operation of reversing v-bit highest order physical address bits does not introduce any significant gate delay, as required by some other address mapping techniques. Bit-reversal address mapping costs very little to implement and has only one parameter (the depth of reversal v) to set, yet improves system performance significantly as Section 4.3 will present.

4.3 Performance Evaluation

The performance of bit-reversal address mapping is evaluated through a revised SimpleScalar v3.0d with the SDRAM module v1.0 as introduced in Section 3.2. The configuration of the baseline machines is listed in Table 3.1. Bit-reversal address mapping is compared with existing address mapping methods, including page interleaving, rank interleaving, permutation-based page interleaving and Intel's 925X chipset address mapping, as introduced in Section 2.6.1. Table 4.1 summarizes all simulated SDRAM address mapping techniques.

Table 4.1: Simulated SDRAM address mapping techniques

Address mapping	Address bits from the highest to the lowest
Flat	Channel, rank, bank, row, column and byte index
Page interleaving	Channel, rank, row, bank, column and byte index [69, 66]
Rank interleaving	Channel, row, rank, bank, column and byte index [58]
Permutation-based page interleaving	Same as page interleaving with bank index XORed with row index [77]
Intel's 925X chipset	Same as page interleaving with reordered row index bits and bank index bits [27]

4.3.1 The Depth of Reversal

Bit-reversal address mapping has one parameter, the depth of reversal, which is studied first to find out the impact of the parameter before comparing bit-reversal with other address

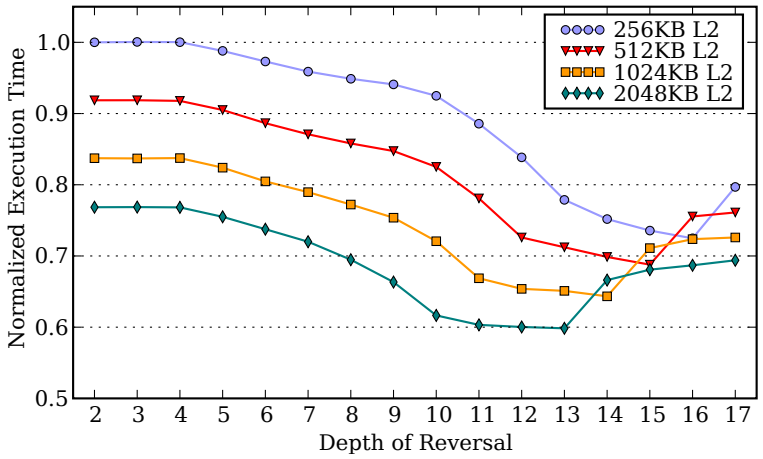

Figure 4.4: The depth of reversal with various L2 cache sizes

mapping techniques.

Simulations with various depths of reversal and L2 cache sizes are performed with 1 billion maximal instructions for each benchmark. Depth of reversal varies from 2 to 17, meanwhile L2 cache size changes from 256KB to 2048KB. Execution time of each combination is normalized to a baseline (depth of 2 with 256KB L2) for each benchmark. Then execution times are averaged crossing all simulated benchmarks. Results are shown in Figure 4.4.

With a 256KB L2 cache, bit-reversal with a depth of 16 reduces the execution time more than 28% over the depth of 2 with the same size L2. As L2 cache size increases, the depth that has the shortest execution time reduces. With a 2048KB L2 cache, depth 13 reduces the execution time by 22%, excluding the improvement achieved by the enlarged cache size. The depths that yield the best performance improvement for each cache size are summarized in Table 4.2.

The third line of Table 4.2 shows that the performance improvement attributable to bit-reversal decreases as L2 cache size increases. This is because bit-reversal address mapping can only reduce latency of main memory accesses which are L2 cache misses. A larger

Table 4.2: The depth having the shortest execution time under various cache sizes

L2 cache size	256KB	512KB	1024KB	2048KB
Depth having shortest execution time	16	15	14	13
L2 cache tag width	17	16	15	14
Improvement over depth 2 with same L2	28%	25%	23%	22%

L2 cache results in fewer main memory accesses. Therefore, bit-reversal is more useful for systems that do no have a large L2 cache, such as embedded systems.

Assuming a set associative L2 cache using physical address (physically indexed and physically tagged), according to Figure 2.7, L2 cache tag width can be calculated using Equation 4.2, where *memory_size* is the size of total physical memory. Given that the simulated machine has a 16-way L2 cache and 2GB physical memory, cache tag width of each L2 cache is also listed in Table 4.2.

$$cache\_tag\_width = log_2(memory\_size) - log_2(\frac{cache\_size}{cache\_associativity}) \tag{4.2}$$

Based on simulation results, the depth of reversal which yields the lowest average execution time is one less than the L2 cache tag width, as shown in Equation 4.3.

$$depth\_of\_reversal = lowest\_level\_cache\_tag\_width - 1 \tag{4.3}$$

Equation 4.3 is learned from experiments. Here is an explanation of the relationship between depth of reversal and L2 cache tag width. Given a write-back n-way associativity L2 cache, when a conflict miss occurs, the cache controller first writes the dirty cache line back to main memory (or write buffer if applicable), then reads a new cache line from main memory and loads it into the same cache line. The read and write memory access during a conflict miss have the identical cache set index but different cache tags. If bit-reversal maps channel, rank and bank index from the address bits corresponding to the identical cache set

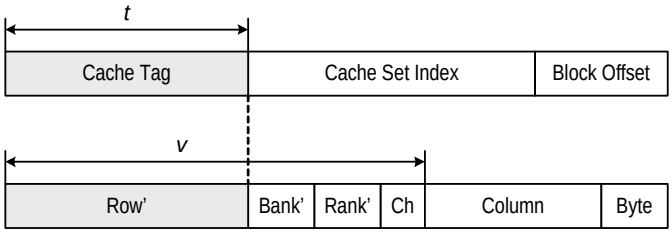

Figure 4.5: Relationship between depth of reversal and cache tag width

index, and maps row index from those cache tag bits, as illustrated in Figure 4.5, then the read and write access pair due to a cache conflict miss will be directed to different rows of the same bank, resulting in a row conflict. Therefore depth of reversal should not exceed tag width of the lowest level cache.

Another advantage of mapping read write pairs into different banks is that memory access scheduling could be easier and more efficient if reads and writes are separate. Because SDRAM devices (except for DDR2) usually have different profiles of read transactions and write transactions on the address/data buses, mixing reads and writes to the same bank may introduce idle bus cycles due to the difference between read write profiles as well as other timing constraints such as t_{WTR} as shown in Table 2.1.

According to Equation 4.2 and Equation 4.3, the most desired depth of reversal can be easily derived from obtainable information, including the total size of main memory, the lowest level cache size and its associativity. Therefore a flexible implementation of bit-reversal address mapping that supports various processors and auto configuring the depth of reversal is feasible.

For the following simulation results, a depth of 15 is used because the baseline machine has a 512KB L2 cache. Although the selected depth of reversal delivers the best performance crossing all simulated benchmarks, it may not be the best depth for each individual benchmark.

4.3.2 Remapped Physical Address Bits Change Pattern

In Section 4.1.2, bits change probabilities of physical address are shown in Figure 4.2. To illustrate the impact of address mapping, a *remapped physical address* is created back from an SDRAM address. A remapped physical address is recomposed, from the highest order bits to the lowest order, by SDRAM channel, rank, bank, row, column and byte index, just like in a reversed way of translating a physical address into an SDRAM address using the flat address mapping as shown in Figure 2.8. Using remapping physical address, impacts of various SDRAM address mapping techniques can be easily examined and compared.

According to baseline machine configuration (Table 3.1), the highest 2 bits (bit{30:29}) of a remapped physical address are rank index. The next highest 2 bits (bit{28:27}) are bank index, followed by 13 bits row index (bit{26:14}) and 11 bits column index (bit{13:3}). The lowest 3 bits are byte index (bit{2:0}). There is no channel index due to the single channel configuration. The lowest 6 bits (bit{5:0}) of the remapped physical address are zero because of memory access granularity which equals to a 64B cache line size. Figure 4.6 shows bits change probabilities of remapped physical address after applying simulated SDRAM address mapping techniques.

For flat address mapping, bit change pattern of remapped physical address is identical to that of physical address. Therefore Figure 4.6(a) is the same as Figure 4.2.

With page interleaving address mapping, bank index (bit{28:27}) has a high probability of change from access to access, meaning accesses are distributed well between banks, as shown in Figure 4.6(b). However, because rank index (bit{30:29}) is still unlike to change, memory accesses are only distributed well between banks inside a rank, but not crossing different ranks.

Similar to page interleaving, permutation-based page interleaving and Intel's 925X chipset address mapping, as shown in Figure 4.6(c) and Figure 4.6(d) respectively, both have high probabilities of change for bank index (bit{28:27}) but lower probabilities for

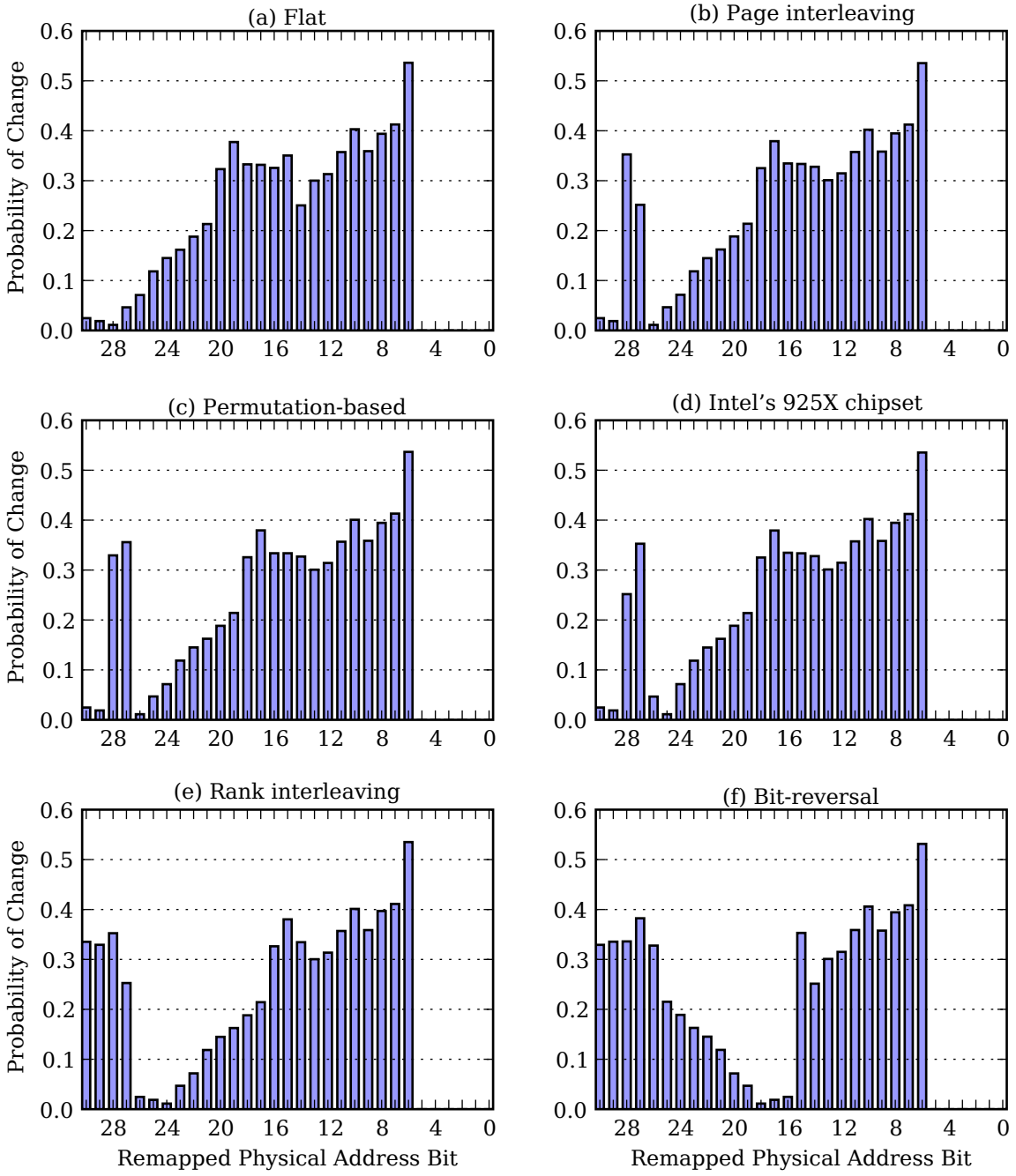

Figure 4.6: Remapped physical address bits change probability between adjacent main memory accesses

rank index (bit{30:29}). Therefore their address bit change patterns are similar to that of page interleaving.

Due to the XOR operation as introduced in Section 2.6.1.4, permutation-based page interleaving has a more evenly change pattern throughout bank index bits than page interleaving and Intel's 925X chipset. Intel's 925X chipset swaps the two highest row index bits (bit{26-25}) [27][3]. However, the row index bit order has no effects on the performance.

With rank interleaving and bit-reversal address mapping, both rank index and bank index have high probabilities of change, as illustrated in Figure 4.6(e) and Figure 4.6(f). Temporally successive memory accesses are likely to be mapped into different banks not only intra ranks but also crossing different ranks, enabling bank parallelism.

Differences between rank interleaving and bit-reversal are, first, rank interleaving does not change bit order of row index bits. However, changing bit order of row index bits has no impacts on performance. Second and most important, rank interleaving maps all row index from the highest address bits, as shown Figure 2.10. While bit-reversal, as shown in Figure 4.3, uses depth of reversal to control how many row index bits are mapped from the highest address bits with the considerations of cache conflict misses, as discussed in Section 4.3.1. Therefore bit-reversal address mapping will be shown to have a better performance than rank interleaving in Section 4.3.6.

4.3.3 Access Distribution in SDRAM Space

Bit-reversal address mapping attempts to map temporally adjacent accesses to different banks crossing all available ranks and channels, thus an even distribution of accesses in the SDRAM address space is expected. Figure 4.7 shows the average percentage of memory accesses directed to each bank across all simulated benchmarks. The simulated machine has a total of 16 banks which are shown in different colors.

[3]The row index bit order of Intel's 925X chipset various between different memory modules [27]. Swapping the two highest row index bits is only applicable to the memory configuration used in the simulations.

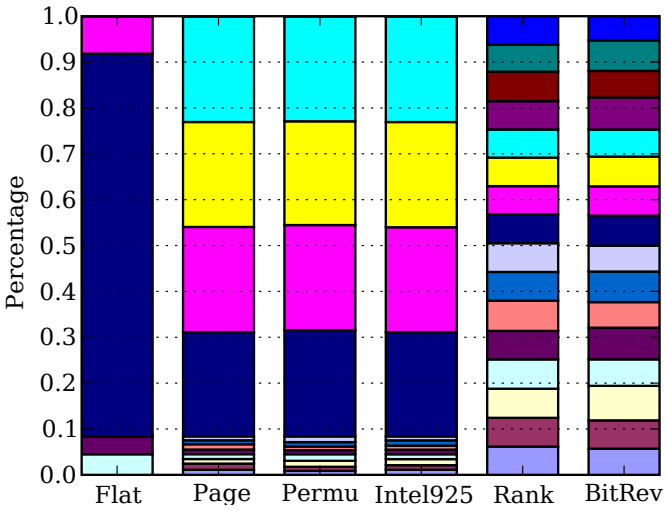

Figure 4.7: Address distribution across all banks

Flat address mapping represents the original memory access distribution in the physical address space, which is determined by program's organization, such as loading text segment, data segment and stack segment into different locations. As shown in Figure 4.7, with flat address mapping only 4 banks have memory accesses and one bank has 84% of total accesses, leaving the rest 12 banks unused.

Page interleaving, permutation-based page interleaving and the Intel's 925X chipset address mapping distribute memory accesses evenly intra ranks as previously discussed. As shown in Figure 4.7, the 4 banks that have the most accesses belong to the same rank. Although these 4 banks share accesses quite equally, memory accesses to different ranks are still unbalanced.

Rank interleaving and bit-reversal both achieve an evenly distributed memory accesses to all available banks. As most performance improvements of computer systems are made by parallelism, i.e. pipeline and superscalar processors, rank interleaving and bit-reversal enable the parallelism available in main memory and therefore improve the performance.

4.3.4 Row Hit and Row Conflict

While evenly distributed accesses enable bank parallelism, localities existing in memory access stream can also be exploited by SDRAM address mapping.

As introduced in Section 2.1.3, a memory access could be a row hit, row empty or row conflict, depending upon the current state of the bank which the access is directed to. Row conflicts can be further divided into two subcategories, hard row conflict and soft row conflict. A *hard row conflict* is a row conflict that immediately follows the previous access which it conflicts with. If there are any accesses directed to other banks between a row conflict and the previous access it conflicts with, then this row conflict is a *soft row conflict*.

The difference is that the latency of a soft row conflict can be partially or completely hidden by the other accesses between the soft row conflict and the previous access. However the latency of a hard row conflict can not be hidden. Figure 4.8 illustrates examples of soft row conflict and hard row conflict.

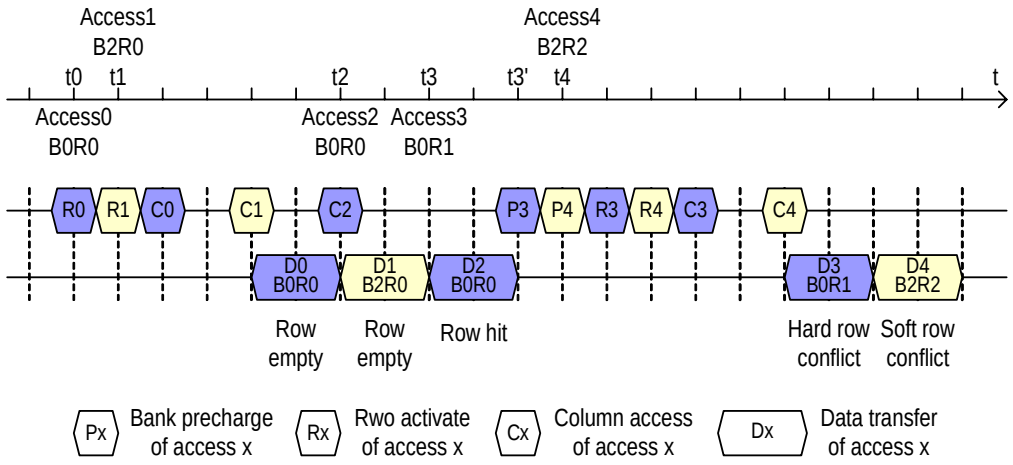

Figure 4.8: Hard row conflict and soft row conflict

Five accesses arrive in order at $t0$ to $t4$, as shown in the top of Figure 4.8. Given that the SDRAM device in this example has a 2-2-2 (t_{CL}-t_{RCD}-t_{RP}) timing and all accesses are scheduled in the same order as they arrive, their actual bus transactions are shown on the

bottom of Figure 4.8. *Access*0, *access*2 and *access*3 are directed to *bank*0, while *access*1 and *access*4 are directed to *bank*2. Assuming *access*0 and *access*1 are row empties, then *access*2 is a row hit, *access*3 and *access*4 are row conflicts. Because *access*3 conflicts with *access*2 and there is no other accesses between them, *access*3 is a hard row conflict. On the other hand, *access*4 conflicts with *access*1 too, but due to the existence of other accesses (*access*2 and *access*3) directed to a different bank, *access*4 is a soft row conflict.

Please note that *access*3 arrives at $t3$, but it can not start the bank precharge until the previous access (*access*2) completes at $t3'$. This is because the bank they both access can not be precharged until the data transfer of the previous access finishes. The bank precharge and row activate of *access*4 are interleaved well with transactions of *access*3, therefore the latency of *access*4 is largely overlapped with the latency of *access*3.

The reason of separating hard row conflicts from soft row conflicts is that the penalty of hard row conflict can not be hidden by interleaving accesses to different banks. Thus hard row conflicts have more negative impacts on performance. Figure 4.9 shows the average row hit and hard row conflict rate crossing all benchmarks for simulated address mapping techniques.

Compared to flat, all simulated SDRAM address mapping techniques significantly increase row hit rate and reduce hard row conflict rate. Column index, which is the only difference between the addresses of row hits, is intact during address mapping, thus row hits are not affected by address mapping. Hard row conflicts are contiguous accesses whose addresses have identical rank and bank index but different row indexes. With address mapping, especially bit-reversal address mapping, rank and bank index are mapped from address bits that have high probabilities to change between contiguous accesses, therefore potential row conflicts are most likely to be mapped into unique banks or ranks. Because bit-reversal has the most evenly distributed accesses over all banks, as shown in Figure 4.7, it achieves the most reduction in hard row conflicts among simulated address mapping techniques.

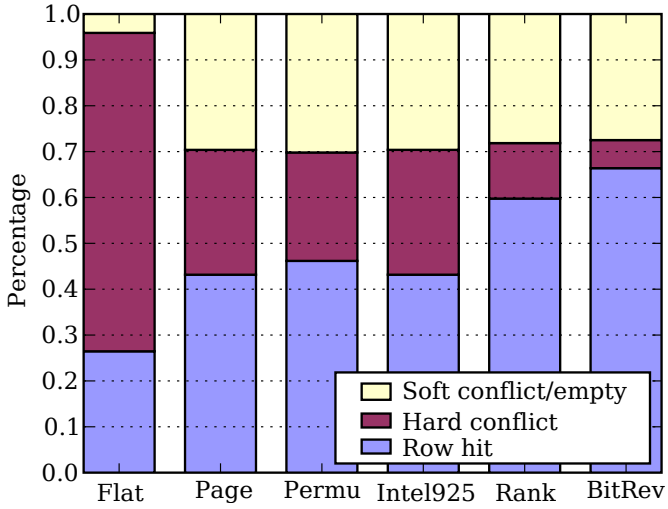

Figure 4.9: Row hit and hard row conflict rate

As shown in Figure 4.9, bit-reversal address mapping greatly reduces the hard row conflict rate from page interleaving's 27% to 6%, resulting in a 78% reduction in hard row conflicts over page interleaving. As a result of reduction in row conflicts, the row hit rate increases. Bit-reversal increases the row hit rate from page interleaving's 43% to 66%, a 53% improvement. Rank interleaving also increases the row hit rate to 59% and reduces the hard row conflict rate to 12% due to its evenly distributed accesses.

4.3.5 Access Latency and Bus Utilization

The major objective of this thesis is to reduce the observed main memory access latency. Figure 4.10 shows the average memory access latency in CPU cycles crossing all simulated benchmarks. The reduced latency is a direct result of increased row hit rate, because row hits have the shortest latency as presented in Section 2.1.3. Bit-reversal reduces the average memory access latency from page interleaving's 142 CPU cycles to 101 cycles, which is the shortest access latency among all simulated address mapping techniques. Rank interleaving achieves the second shortest access latency of 110 CPU cycles.

Figure 4.11 shows the average SDRAM bus utilization. As row hit rate increases, there

will be fewer bank precharge and row activate transactions. Consequentially the number of idle cycles of data bus, which are spent waiting for bank precharge and row activate transactions to complete, is also reduced. Bit-reversal address mapping improves SDRAM data bus utilization from paging interleaving's 24% of to 29%, yielding the maximal data bus utilization as well as the best effective memory bandwidth among simulated address mapping techniques.

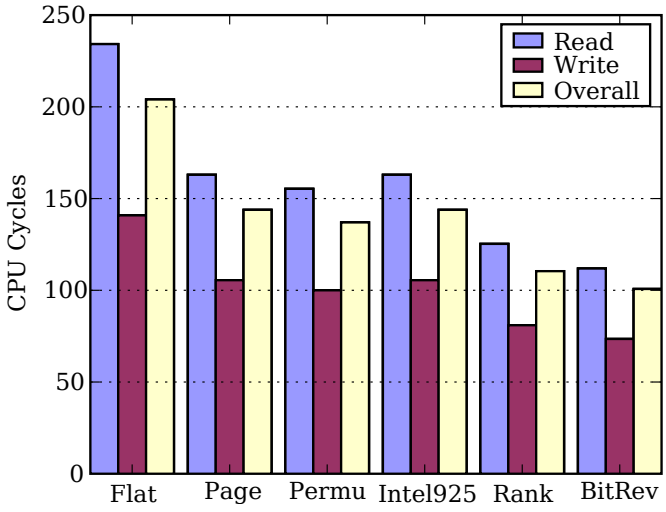

Figure 4.10: Average main memory access latency in CPU cycles

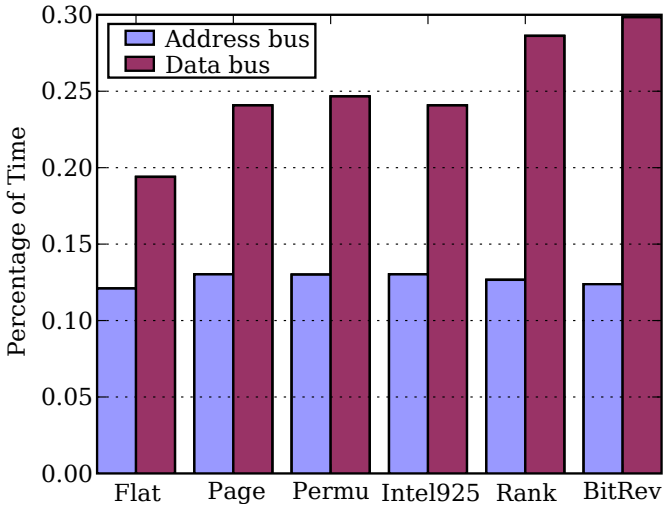

Figure 4.11: Average SDRAM address and data bus utilization

4.3.6 Execution Time

Execution time is the most important metric for performance evaluation. Figure 4.12 shows the execution time of all simulated benchmarks under various address mapping techniques. Execution times of different address mapping techniques are normalized to page interleave for each benchmark. Flat address mapping is now shown here because it is inefficient and has not been used by any real systems. The average execution times for each address mapping across all benchmarks are shown in the rightmost columns. Bit-reversal address mapping shown in the figure has a depth of 15 as discussed in Section 4.3.1.

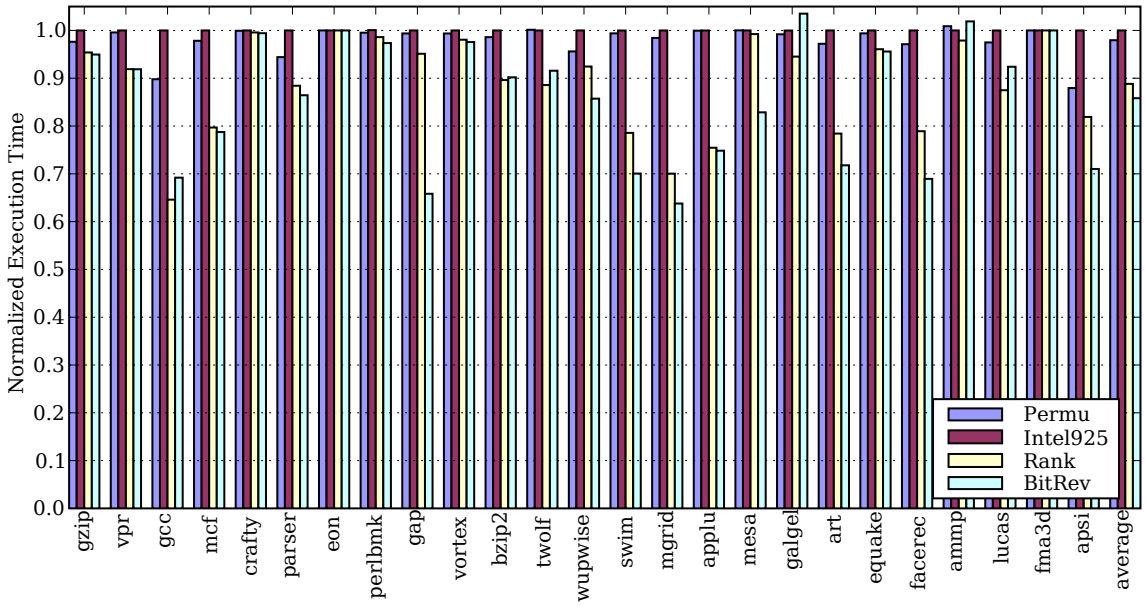

Figure 4.12: Normalized execution time of various address mapping

Among 25 simulated SPEC CPU2000 benchmarks, bit-reversal only increases the execution time of two benchmarks, `galgel` and `ammp`, by 4% and 2% respectively. This is because the specified depth of reversal (15) is selected to achieve the minimal average execution time over all benchmarks. Depth of 15 may not yield the shortest execution time for benchmark `galgel` and `ammp`.

For the rest 23 benchmarks, bit-reversal address mapping does a very good job in reducing execution time. It reduces the execution time by more than 10% for 12 benchmarks over page interleaving. For some benchmarks, including `gcc`, `gap`, `swim`, `mgid` and `facerec`, the reductions of execution time are more than 30%. On average, bit-reversal reduces the execution time by 14% over page interleaving.

Rank interleaving also achieves a 11% reduction of execution time on average over page interleaving. Rank interleaving outperforms bit-reversal on some benchmarks, such as `gcc`, `galgel`, `lucas` and etc. Again, this is because the depth of 15 used in the simulations may not be the best performed depth for these benchmarks.

Intel's 925X chipset address mapping achieves the exactly same performance as page interleaving. According to Figure 2.13 and the memory configuration used in the simulations, the only difference between Intel's 925X chipset and page interleaving is the swapping of the 2 highest row index bits [27]. As discussed in Section 4.3.2, the bit order of row index has no effects on performance. Therefore, given the simulated memory configuration, Intel's 925X chipset virtually uses the same address paging as page interleaving.

Due to the XOR operation, permutation-based page interleaving achieves an average of 2% performance improvement over page interleaving. For some benchmarks, such as `gcc` and `apsi`, permutation-based page interleaving reduces the execution time by 10% over page interleaving.

Based on the simulation results, bit-reversal achieves the best performance among all address mapping techniques examined. Bit-reversal address mapping creates an evenly distributed accesses over all available banks, enables the parallelism provided by main memory. By interleaving accesses directed to different banks, access latencies are partially hidden and thus reduced. Bit-reversal address mapping also attempts to direct potential row conflicts into unique banks. The increased row hit rate leads to greater row locality, which is exploited by SDRAM sense amplifiers to further improve the performance.

4.4 Address Mapping Working under Other Techniques

SDRAM Address mapping techniques enables bank parallelism and exploits row locality to reduce memory access latency. SDRAM controller policy as discussed in Section 2.3 has impacts on how the SDRAM row locality is exploited. The operating system virtual paging also affects the distribution of memory accesses in memory space. The effects of SDRAM controller policy and the OS virtual paging system, as well as how SDRAM address mapping techniques perform when working in conjunction with them are studied in this section.

4.4.1 Address Mapping with Controller Policy

After an access completes, the SDRAM controller makes the decision whether to leave the accessed row open or not subjected to the controller policy, as described in Section 2.3. Two static controller policies, OP and CPA, are commonly used and can be selected through the BIOS. For example, enabling a BIOS option, called SRAM Page-Mode, selects the OP policy, which keeps the activated row open and allows fast access to the same row [71]. The program is expected to contain significant row locality in main memory access stream in order to take the advantage of the OP policy. Otherwise the OP policy will result in frequent row conflicts. For programs that have little locality in main memory access stream, the CPA policy, which closes each accessed row and precharges the bank automatically, can change potential row conflicts into row empties, alleviating the penalty caused by row conflicts.

The controller policy that yields the best performance largely depends on an application's memory access pattern. In Section 3.2.1.3, an ideal dynamic controller policy, Dynamic Upper Bound (DYN-UPB), is introduced. The DYN-UPB policy uses future access information only available in simulations to select the most appropriate policy for each access. If the next access is known to be a row hit, then the OP policy is chosen; otherwise, the CPA policy is chosen. The OP policy is used for all previous simulation results. Impacts of the CPA and DYN-UPB policy are hereby evaluated.

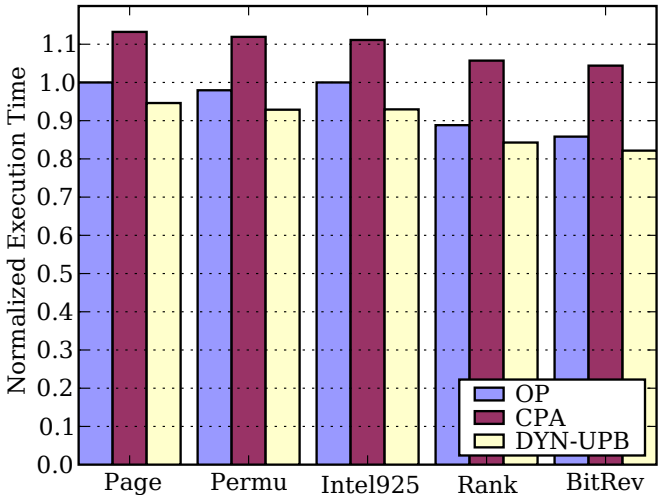

Figure 4.13: Address mapping techniques under controller policies

Figure 4.13 shows the execution times corresponding to the OP, CPA and DYN-UPB controller policy under different address mapping techniques. The execution times are normalized to page interleaving with the OP policy, then averaged crossing all benchmarks, to illustrate the effects of both address mapping techniques and controller policy.

For each controller policy, bit-reversal address mapping reduces the execution time by 14% over page interleaving with the OP policy, by 8% with the CPA policy and 13% with the DYN-UPB policy. Other address mapping techniques, such as rank interleaving and permutation-based page interleaving, also show consistent performance improvements with different controller policies. This means that address mapping techniques can work together with various controller policies to achieve a maximal reduction of execution time.

Also from Figure 4.13, the OP policy constantly outperforms the CPA policy because most SPEC CPU2000 benchmarks exhibit significant locality in main memory access. As an ideal dynamic controller policy, the DYN-UPB policy has the best performance and gives an upper bound on performance improvement that a predictive dynamic controller policy could provide. One dynamic controller policy predictor proposed by Ying Xu uses a history based predictor to make the decision and has shown some performance improvements [74].

4.4.2 Address Mapping with Virtual Paging

The previously simulated machine does not incorporate virtual paging system. Embedded systems frequently do not incorporate virtual paging, whereas most PC operating systems do support virtual paging, which can significantly impact memory access distribution.

In Section 3.2.1.4, a preliminary virtual paging system is introduced and implemented into the simulator. Two page allocation algorithms, sequential allocation and random allocation, are implemented. With sequential allocation pages are allocated sequentially from the lowest address to the highest address. Random allocation allocates pages across all available physical pages completely randomly. Impacts of virtual paging system with these two page allocation algorithms on SDRAM address mapping are hereby studied.

The simulated machine for virtual paging studies has the same configuration as the baseline machine shown in Table 3.1, except for the size of main memory. 2GB main memory is replaced by 512MB, which consists of two ranks of eight 256Mbit technology (32Mx8) SDRAM devices. The reason of reducing main memory size is to limit the size of the page table. The virtual page size is 4KB. Page swapping is unnecessary because 512MB memory is large enough for all simulated benchmarks and only one benchmark is simulated at a time.

Figure 4.14 shows the average execution times of simulated address mapping techniques under different virtual paging systems. Execution times are normalized to page interleaving with no virtual paging to show both the impacts of address mapping techniques and virtual paging system. When virtual paging is absent, bit-reversal reduces the execution time by 14% over page interleaving. With the sequential allocation virtual paging, bit-reversal reduces the execution time by 6%. However, when the random allocation virtual paging is used, all simulated address mapping techniques including the flat show less than 2% performance difference.

This is because spatial locality is partially destroyed during virtual to physical address

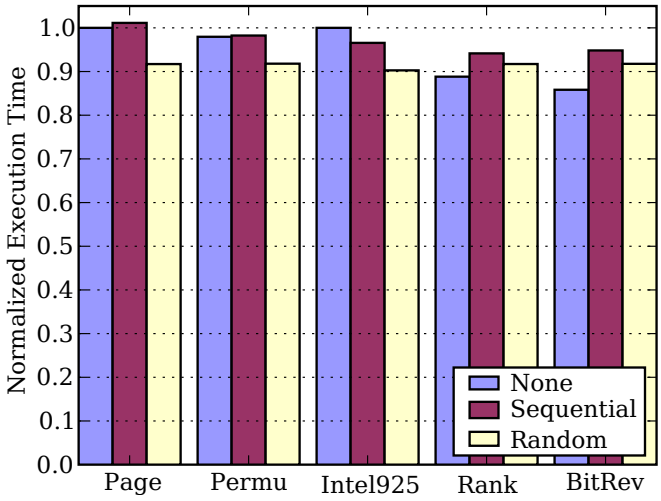

Figure 4.14: Address mapping techniques under virtual paging systems

translation, depending upon the page allocation algorithm used. A random allocation algorithm places pages randomly and thus evenly over all SDRAM banks. Therefore only spatial locality inside virtual pages is preserved. Any spatial locality above the size of virtual pages (4KB) is completely destroyed by the random page allocation algorithm, leaving little performance improvement space to SDRAM address mapping.

Bit-reversal address mapping has been shown working well with no virtual paging as well as with virtual paging using sequential page allocation. SDRAM address mapping technique has little effects on performance with the random allocation virtual paging system. Page placement in a modern operating system will likely fall somewhere between the random and sequential allocation. Therefore SDRAM address mapping, especially the bit-reversal, will still be able to improve performance, although the performance gain contributed will be less significant and non-deterministic dependent upon the actual page placement.

An intelligent page allocation algorithm that is aware of main memory structure and nonuniform access latency could achieve a better performance than the sequential or random allocation. Obviously that requires the incorporation with the compiler and/or the operating system, and will be a part of future work of this thesis.

Chapter 5

Access Reordering Mechanisms

Conventional memory controllers serve memory request in the same order as they received. While the in order scheduling is easy to implement, it is obviously inefficient considering the nonuniform characteristics of main memory and the fact that modern processors always have a set of accesses to choose from. Access reorder mechanisms attempt to schedule outstanding memory accesses in an order that will increase row locality, therefore resulting in a reduced overall execution time. The proposed burst scheduling is presented, which clusters row hits into bursts to maximize the utilization of SDRAM buses. Design space of burst scheduling is exploited and optimizations are proposed. The performance of burst scheduling is examined and compared with existing access reordering mechanisms. Finally the combination of access reordering mechanisms and SDRAM address mapping techniques is studied.

5.1 Philosophy of Burst Scheduling

In a packet switching network, data are encapsulated in packets which are commonly composed by header and payload (data). The effective throughput of the network is usually less than the theoretical network bandwidth because a fraction of the bandwidth is used

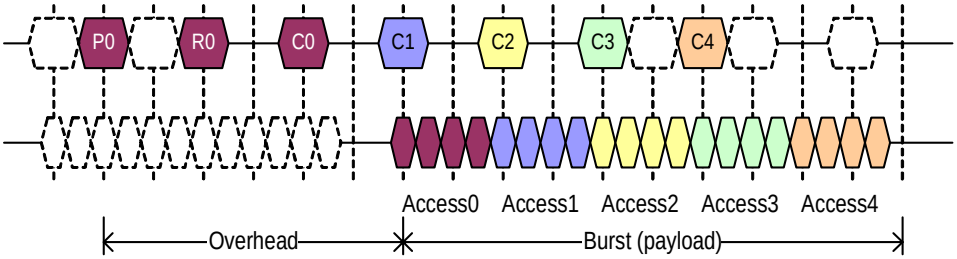

Figure 5.1: Creating bursts from row hits

to transmit packet header. One way to increase the throughput is to use large packets. Because the fraction of the overhead due to packet header reduces as packet sizes increase, large packets usually result in high throughput.

Consider a main memory access, if the bank precharge, row activate and column access transactions are considered as the overhead of a packet, and the actual data transaction is considered as the payload, then the above theory about packet switching network can also be used in memory access scheduling, which is a larger payload will result in a higher data bus utilization.

The data transaction size of each memory access is usually equal to the lowest level cache line size, so a large payload can be created from multiple data transactions from different accesses. As show in Figure 5.1, accesses that are directed to the same row of the same bank are selected from all available outstanding accesses and clustered together to form a *burst*. With a OP controller policy, data transactions of the accesses inside a burst can be performed on back to back cycles, resulting in a large payload and a high data bus utilization. The size of bursts can grow as newly arrived accesses join existing bursts which are being scheduled. The larger a burst is, the higher data bus utilization it has.

One drawback of using large packets in a packet switching network is the slow response time. Especially when variable sized packets are allowed, small packets will experience long latency if they follow large packets. This issue becomes worse in burst scheduling where the size of a burst can increase dynamically. Starvation may occur to small bursts when new

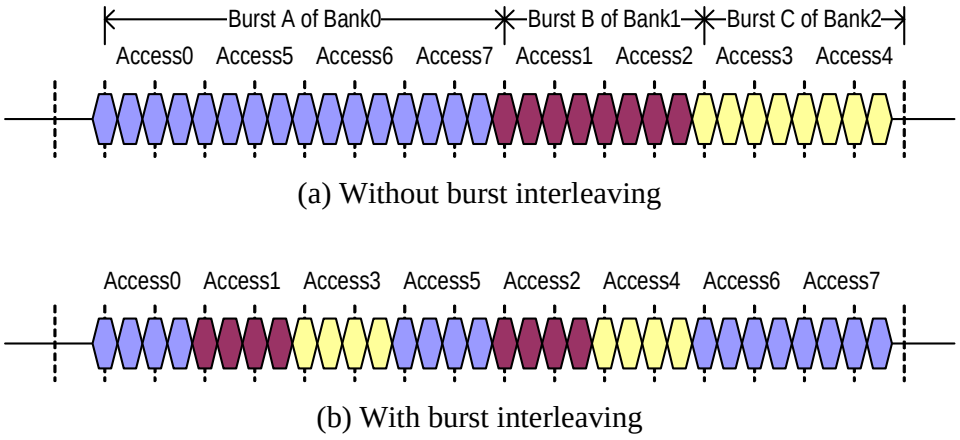

Figure 5.2: Interleaving bursts from different banks

accesses keep joining a burst being scheduled. The solution is to interleave bursts.

As shown in Figure 5.2, three bursts are created from eight accesses directed to three different banks. Without burst interleaving, *access*5, *access*6 and *access*7 that arrive later join *burstA* and are scheduled earlier than the accesses in *burstB* and *burstC*, resulting in long latencies to those older accesses of small bursts, as illustrated in Figure 5.2(a). *BurstB* and *burstC* could be starving if *burstA* keeps increasing. To reduce the latency to old accesses and prevent starvation, three bursts are interleaved as shown in Figure 5.2(b). Latency of older accesses (*access*1 to *access*4) are reduced. While burst interleaving does not affect the data bus utilization, it allows different sized bursts from unique banks to be served in relatively equal opportunity, preventing starvation.

Burst interleaving needs to be performed carefully as bubble cycles may be introduced. For example, DDR2 devices require a rank-to-rank turnaround cycle to be inserted between two data transactions from different ranks [30]. Therefore interleaving bursts between different ranks will cause significant rank-to-rank turnaround cycles and degrade the performance. Also, read accesses and write accesses usually have different profiles. Additional timing constraints, such as t_{WTR} as shown in Table 2.1, need to be met when mixing reads and writes. Therefore interleaving reads and writes may also cancel the performance gained

by burst scheduling.

Existing access reordering mechanisms, including the row hit scheduling and Intel's out of order scheduling, also attempt to combine row hits to exploit row locality and improve bus utilization. However, their row hit first policy is more like a best effect in creating a large burst. There is no guarantee that data transactions of selected row hits are transferred in back to back cycles.

5.2 Evolution of Burst Scheduling

As introduced in Section 2.7.3, previous studies have proposed access reordering mechanisms for stream-oriented systems [56, 25, 41, 55], web servers [54], network processors [22], embedded systems [36, 33] and other applications [79, 39, 46, 34]. Chipset manufacturers also have proprietary memory access scheduling solutions [57]. Motivated by the memory wall, the studies of access reordering mechanisms examine these existing access reordering mechanisms with the goals of exploiting their best characteristics and addressing their shortcomings.

5.2.1 Preliminary Study

The first study of access reordering mechanisms is to explore the design space of the row hit access scheduling as introduced in Section 2.7.3.2. As shown in Figure 2.17, with the row hit scheduling, outstanding memory accesses are stored in unique bank queues and selected with a row hit first reordering policy. A global scheduler chooses one access from all bank queues using a bank selection policy such as round robin.

The design space is explored by using various reordering policies as well as bank selection policies. *Reordering policy* determines the access scheduling order within each bank, while accesses from different banks are chosen subjected to *bank selection policy*. Table 5.1 lists some of the possible reordering policies and bank selection polices.

Table 5.1: Possible reordering policies and bank arbiter policies

Reordering policies	
Oldest first	Memory accesses leave queue in order
Row hit first	Row hit access first
Most dependent	Access having the most dependent instructions first
Bank selection policies	
Round robin	Round robin style (same as least recently used)
Most recently used	The most recently used bank first
Longest queue	The bank having the longest queue first
Shortest queue	The bank having the shortest queue first
Most dependent	The bank having the most dependent instructions first
Random	Randomly select bank

Preliminary simulation results show that reordering policy has impacts on performance. However, bank selection policies expect for the round robin policy do not show any significant performance improvements. This is because listed bank selection policies rely solely on bank queue information to make bank selecting decisions, they may miss the chances to interleave accesses from different banks.

For example, with the most recently used policy, the selector attempts to schedule all pending accesses from one bank before switching to another bank. It does not take advantage of any bank parallelism. The reason why the round robin policy performs well is that adjacently scheduled accesses are from different banks therefore they can be interleaved to hide latency.

Based on the preliminary study, the bank selector needs to consider SDRAM device state, device timing constraints as well as bus usages in order to achieve a good performance, leading to a two-level scheduling.

5.2.2 Burst Scheduling: A Two-level Scheduler

An improved access reordering mechanism uses a two-level scheduler. Firstly, access reordering are performed inside access queues at the access level. This is done by applying

reordering policies to control how accesses enter and/or leave the queues. Secondly, bus transactions belonging to access selected from different banks are scheduled at the bus transaction level with considerations of SDRAM device state and bus contentions. Various access queue structures and reordering policies were considered for access level scheduling; there are also different scheduling algorithms for bus transaction level scheduling.

The advantage of a two-level scheduler is that at each level different information are used to make scheduling decisions. For example, access level scheduling could consider the potential row hit and row conflict, trying to creating more row hits by reordering accesses in the queue; while transaction level scheduling could focus on how to interleave transactions between different banks to maximize the bus utilization.

Burst scheduling, as Section 5.3 will introduce in details, is a two-level scheduler. At the access level, burst scheduling clusters row hits within each bank to create bursts. At the transaction level, it uses a sophisticated bus transaction scheduler to schedule transactions of different accesses with the goal to maximize data bus utilization while maintaining burst structure. Burst scheduling also treats reads and writes differently at both levels through unique read and write queues and different priorities.

5.2.3 Optimizations to Burst Scheduling

As the studies of access reordering continue, optimizations are proposed to burst scheduling, including read preemption, write piggybacking and burst threshold.

As originally conceived in burst scheduling, when the scheduling process of an memory access starts, it can not be interrupted until all required transactions are scheduled. As mentioned in Intel's out of order scheduling, a write access could be interrupted by a read access to reduce the read latency [57]. This technique, known as read preemption, is employed in burst scheduling to prioritize read accesses.

As reads are aggressively prioritized over writes, writes begin to impact the performance,

due to the more frequent occurrence of write queue saturation, which may cause CPU pipeline stalls. Another optimization, called write piggybacking, is hereby proposed. Write piggybacking attempts to append qualified writes at the end of read bursts to exploit row hit in writes and prevent write queue saturations without significantly increasing read latency.

Read preemption and write piggybacking will be found working well individually but they may conflict with each other on some benchmarks when working together. Therefore a threshold is introduced to make a tradeoff between two optimizations in order to achieve a performance improvement crossing all benchmarks. These optimizations, read preemption, write piggybacking and burst threshold will be discussed in details in Section 5.3.2.2.

5.3 Details of Burst Scheduling

As introduced in Section 2.7.2, the SDRAM data bus is more critical than the address bus due to access granularity and increased timing constraints. Based on existing access reordering mechanisms, burst scheduling is designed to increase the row hit rate and maximize the SDRAM data bus utilization.

Burst scheduling creates bursts by clustering outstanding accesses directed to the same rows of the same banks. Accesses within a burst, except for the first one, are row hits and only require column access transactions. Data transactions from these accesses can be transferred back to back without any idle cycles on the SDRAM data bus, resulting in a high data bus utilization. Individual accesses that do not belong to any existing bursts are treated as bursts containing single access. Bursts within a bank are sorted based on the arrival time of the first access to prevent starving these single access bursts or small bursts.

Newly arrived accesses can join existing bursts even when the bursts are being scheduled. A large or an increasing burst may delay bursts with in the same bank or from other banks, increasing the latency of old accesses in other bursts. Therefore bursts are interleaved between different banks to give relatively equal opportunity to each burst, reducing latency

to old accesses of small bursts. When bursts to multiple banks are interleaved, accesses except for the first one of each burst are still row hits. Thus the high data bus utilization can still be maintained. Considerations must be taken when interleaving bursts to avoid bubble cycles on the data bus due to timing constraints, i.e. rank-to-rank turnaround cycles.

5.3.1 Hardware Structure

Figure 5.3 shows the structure of burst scheduling, which consist of the memory access queue and the SDRAM bus transaction scheduler, as introduced in Section 3.3.2.

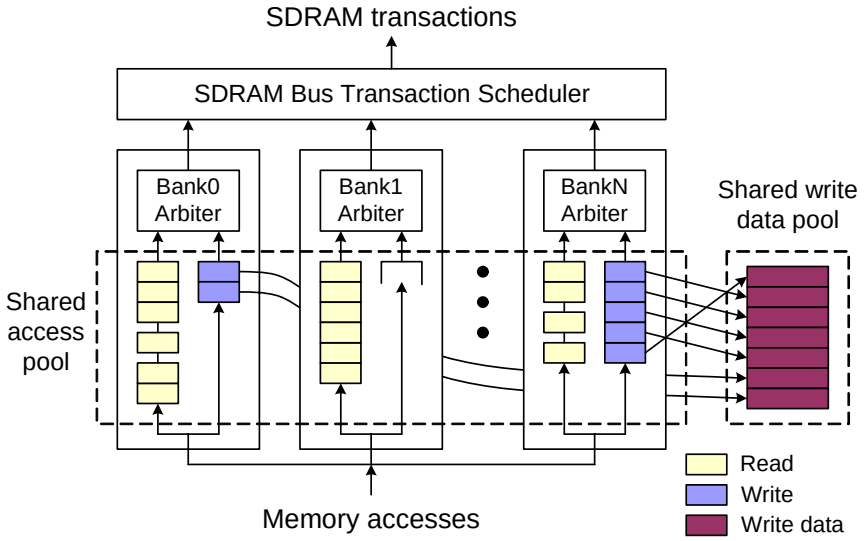

Figure 5.3: Structure of burst scheduling

Memory accesses directed to the same bank are stored in unique read queue and write queue. All read queues and write queues share a global access pool and a write data pool is used to store the data associated with writes. Both read and write queues are implemented in linked lists, therefore there are no limits on queue length as long as the shared access pool and the write data pool are not full. Accesses are organized in bursts in read queues. An extra field in the data structure of the list element is used to indicate bursts. Depending

on the row index of the access address, upon arrival new reads will join existing bursts or new bursts will be created and appended at the end of the read queues. There are bank arbiters at each bank deciding whether a read or a write will be scheduled next. At each memory cycle, the SDRAM bus transaction scheduler choices one access from all accesses selected by bank arbiters and performs the corresponding transaction of that access based the state of SDRAM device.

5.3.2 Scheduling Algorithm

The scheduling algorithm of burst scheduling is composed of three subroutines: first, access enter queue subroutine; second, bank arbiter subroutine; third, SDRAM bus transaction scheduler subroutine. These subroutines could be transformed into Finite State Machine (FSM) for incorporation into the SDRAM controller.

5.3.2.1 Access enter queue subroutine

Figure 5.4 shows the access enter queue subroutine, which is called when new accesses enter the queues. Because the write queue serves as a write buffer to allow reads to bypass writes, reads upon arrival need to search the write queue for possible hits[1]. A write queue hit occurs when a read requests the data at the same location as a preceding write. The data from the latest write (if there are multiple) will be forwarded to the read such that the read can complete immediately. Reads that are missed in the write queue enter the read queue. If a read is directed to the same row as an existing burst, the read will be appended to that burst. Otherwise, a new burst composed of the single new read will be created and appended to the read queue. Bursts in the read queue are sorted by the arrive time of the first access of each burst. All writes enter the write queue in order and are completed immediately from the view of the CPU.

[1] Although write queue hits happen very unfrequently due to the small size of the write queue, reads have to check with the write queue for possible hits to guarantee the correctness of program execution.

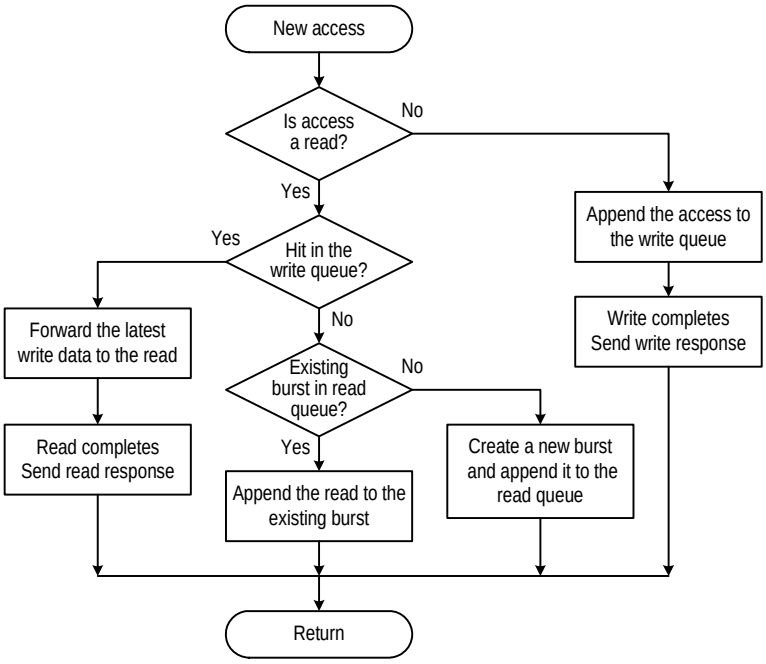

Figure 5.4: Access enter queue subroutine

5.3.2.2 Bank arbiter subroutine

As introduced in Section 3.3.2.1, each bank has one ongoing access, which is the access for which transactions are currently being scheduled, but have not yet been completed. The bank arbiter selects the ongoing access from either the read queue or the write queue. Reads are generally prioritized over writes. Writes are selected only when there are no outstanding reads in the read queue, when the write queue is full or when doing write piggybacking. The algorithm is given in Figure 5.5.

As mentioned in Section 5.2.3, there are optimizations available to burst scheduling. These optimizations are implemented in bank arbiter subroutine. The bank arbiter has two options, read preemption and write piggybacking. *Read preemption* allows a newly arrived read to interrupt an ongoing write. The read becomes the ongoing access and starts immediately, therefore the read will have a shorter latency. Read preemption will not affect the correctness of execution; the preempted write is moved back to the write queue and will

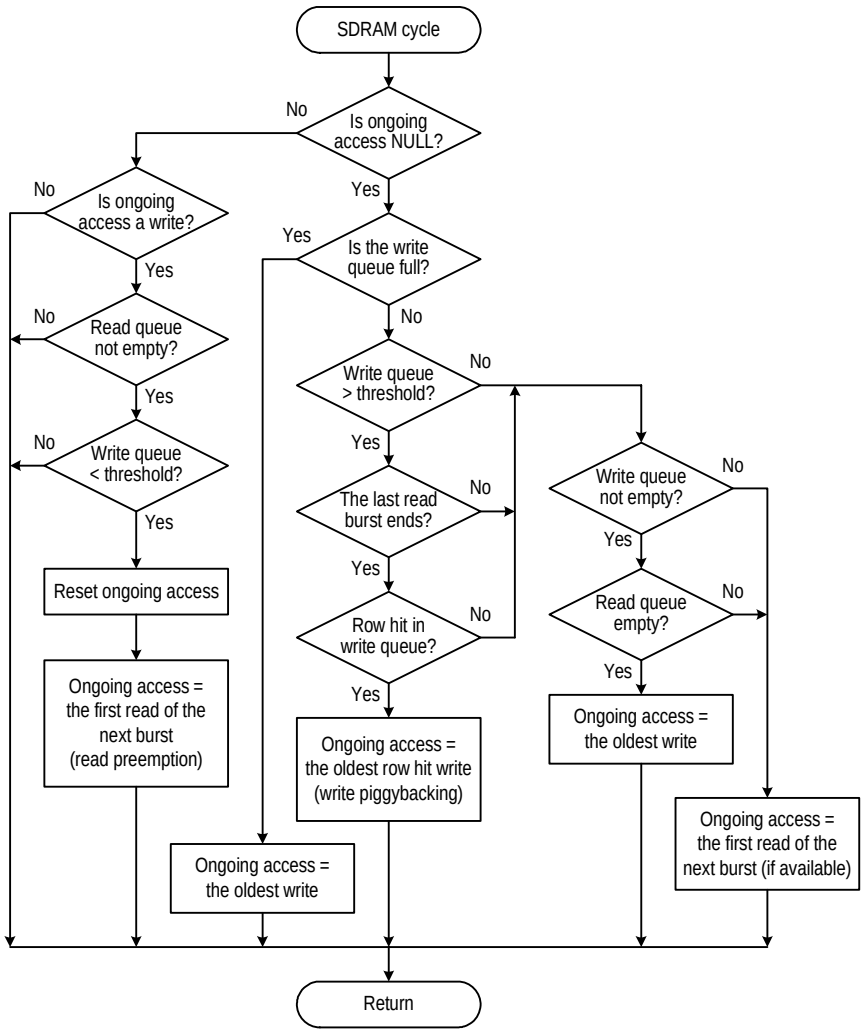

Figure 5.5: Bank arbiter subroutine

be restarted later.

The major functionality of the write queue, besides hiding the write latency and reducing write traffic [63], is to allow reads to bypass writes. The write queue capacity is determined by the size of shared write data pool as shown in Figure 5.3. Due to M5's implementation, when the write queue reaches its capacity, main memory can not accept any new accesses even if there are still rooms in the shared access pool, causing a possible CPU pipeline stall.

Write piggybacking is designed to prevent write queue saturation by piggybacking qualified writes at the end of bursts. The writes being appended must be directed to the same row as the burst, so that they will not disturb the continuous row hits created by the burst. Write piggybacking is implemented by searching for qualified writes in the write queue at the end of each burst. If there are no such writes available, the next burst will be started. There may be idle cycles on the data bus between the last read and the first write in a burst due to the difference in read and write profiles. Write piggybacking can greatly reduce the probability of write queue saturation and also exploit the row locality from writes.

Read preemption and write piggybacking may conflict with each other, i.e. a piggybacked write may be preempted by a new read. A threshold is introduced to allow the bank arbiter to have better control over read preemption and write piggybacking. When the write queue occupancy is less than the *threshold*, read preemption is enabled. Otherwise, write piggybacking is enabled. Section 5.4.6 will have a detailed study of the threshold and determine the threshold which results in the shortest execution for simulated benchmarks.

5.3.2.3 SDRAM bus transaction scheduler subroutine

An SDRAM transaction is considered as unblocked when all timing constraints are met, as introduced in Section 3.3.2.2. At each memory cycle, the SDRAM bus transaction scheduler selects one ongoing access from all banks whose next transaction is unblocked and schedules that transaction.

Table 5.2: SDRAM transactions priority table (1: the highest, 8: the lowest)

Type	Transaction	Same bank	Same rank	Other ranks
Read	Bank Precharge	5	5	5
	Row activate	5	5	5
	Column access	1	2	7
Write	Bank Precharge	6	6	6
	Row activate	6	6	6
	Column access	3	4	8

A priority table, as shown in Table 5.2, is used to select the ongoing access containing the next unblocked transaction to be performed. Among all unblocked transactions, column accesses within the same rank as the last access performed by the scheduler have the highest priorities (priority 1 and 2).

Column accesses from different banks of the same rank are interleaved to create burst interleaving as illustrated in Figure 5.2, so that bursts from different banks are equally served, preventing starvation. The high data bus utilization can be maintained because interleaved accesses are still row hits.

Bank precharge and row activate transactions have the next highest priorities as they do not require SDRAM data bus resource therefore can be overlapped with ongoing column access transactions. The scheduler has priorities set to finish all bursts within a rank before switching to another rank to avoid the rank-to-rank turnaround cycles required by DDR2 devices [30]. Column accesses from different ranks thus have the lowest priority. At each category, read transactions always have higher priorities than write transactions. An oldest access first policy is used to break ties.

Based on the scheduling priority in Table 5.2, the subroutine of the SDRAM bus transaction scheduler is shown in Figure 5.6. In case there are no unblocked transactions from any accesses, the scheduler will switch to the bank which has the oldest access and starts the first transaction of the oldest access when it becomes unblocked.

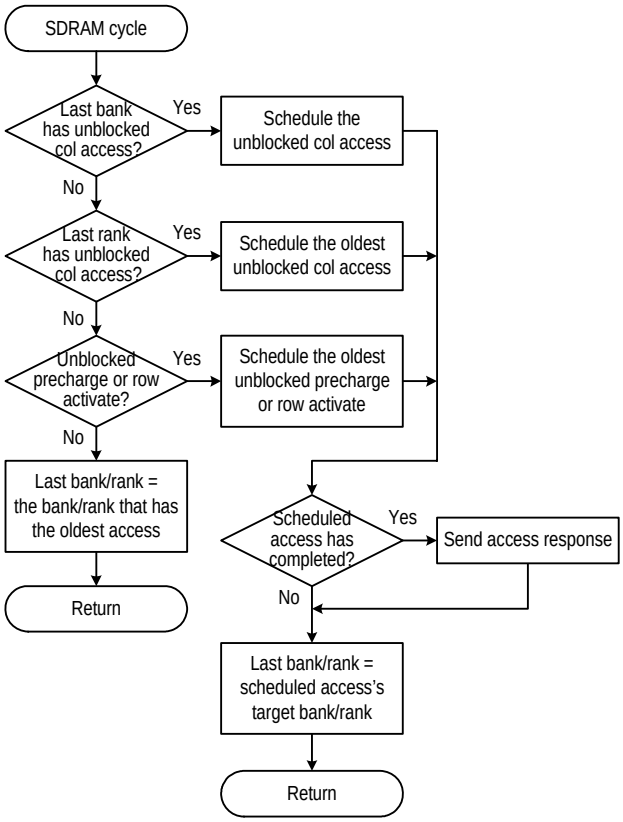

Figure 5.6: SDRAM bus transaction scheduler subroutine

5.3.3 Program Correctness

Burst scheduling will not affect program correctness. There are three types of data hazards or data dependencies: Read after Write (RAW), Write after Read (WAR) and Write after Write (WAW) [61]. Reads are checked in the write queues for hits before entering the read queues. If a read hits in the write queue, the latest data will be forwarded from the write to the read, so read after write (RAW) hazards are avoided. Within bursts, writes are always piggybacked after reads which have previously checked the write queue for read hits. That avoids write after read (WAR) hazards. When piggybacking, the oldest write will be selected first. Writes directed to the same rows are scheduled in program order, therefore write after write (WAW) hazards are also avoided.

5.4 Performance Evaluation

Performance of burst scheduling is evaluated and compared to existing access reordering mechanisms through a revised M5 simulator v1.1 with the SDRAM module v2.0, as introduced in Section 3.3.2. The simulated machine configuration is listed in Table 3.2. Access latency, row hit/conflict rate, bus utilization and execution time are used as the major metrics in performance evaluation.

5.4.1 Simulated Access Reordering Mechanisms

Besides bank in order scheduling (BkInOrder), three existing access reordering mechanisms are also simulated, including the row hit scheduling (RowHit) [56], Intel's out of order memory access scheduling (Intel) [57] and Intel's scheduling with read preemption option (Intel_RP). Table 5.3 summarizes all simulated access reordering mechanisms.

Table 5.3: Simulated access reordering mechanisms

BkInOrder	In order scheduling intra banks, round robin inter banks
RowHit	Row hit first intra bank, round robin inter banks [56]
Intel	Intel's out of order memory access scheduling [57]
Intel_RP	Intel's out of order memory access scheduling with read preemption
Burst	Burst scheduling
Burst_RP	Burst scheduling with read preemption
Burst_WP	Burst scheduling with write piggybacking
Burst_TH	Burst scheduling with a static threshold (52)

As introduced in Section 2.7.3.2, the row hit scheduling uses unified access queues for each bank. The oldest access directed to the same row as the last access to that bank is selected first. Accesses from different banks are scheduled in a round robin fashion.

Intel's out of order memory scheduling, as introduced in Section 2.7.3.8, features unique read queues per bank and a single write queue for all banks. Reads are prioritized over writes to minimize read latency. Once an access is started, it will receive the highest priority so that it can finish as quickly as possible to reduce the degree of reordering [57]. Not proposed

in the original patent, Intel's access scheduling with read preemption, which allows reads to interrupt ongoing writes as described in Section 5.3.2.2, is also simulated.

As discussed in Section 5.3, burst scheduling with read preemption (Burst_RP) allows reads to preempt writes. Burst scheduling with write piggybacking (Burst_WP) carries qualified writes as piggybacks at the end of bursts. Burst scheduling with threshold (Burst_TH) uses an experimentally selected threshold to switch dynamically between read preemption and write piggybacking. The simulation results of Burst_TH to be presented in Section 5.4.6 show that Burst_TH with a static threshold of 52 delivers the best performance for the simulated benchmarks, thus this threshold will be used for the experiments comparing access reordering mechanisms.

5.4.2 Row Hit Rate and Row Conflict Rate

Row hits require only one column transaction and result in shorter latencies than row empties and conflicts, as introduced in Section 2.1.3. Therefore access reordering mechanisms, such as the row hit scheduling, attempt to prioritize row hits over row empties and conflicts. As row empties and conflicts are postponed, chances are they may result in more row hits as new accesses arrive. Figure 5.7 shows the average row hit, row conflict and row empty rate crossing all simulated benchmarks.

All out of order access reordering mechanisms are able to increase row hit rate compared to BkInOrder. Among them, RowHit, Burst_WP and Burst_TH have the highest row hit rates (59-60%) and lowest row conflict rate (33-35%). As a comparison BkInOrder has a 42% row hit rate and a 52% row conflict rate. Intel and Burst without write piggybacking have lower row hit rates although they are still better than BkInOrder. The reason is that in contrast to Intel and Burst which only search row hits in the read queues, RowHit, Burst_WP and Burst_TH seek row hits in both the read queues and the write queues.

With the OP policy, row empties only happen after SDRAM auto refreshing as banks

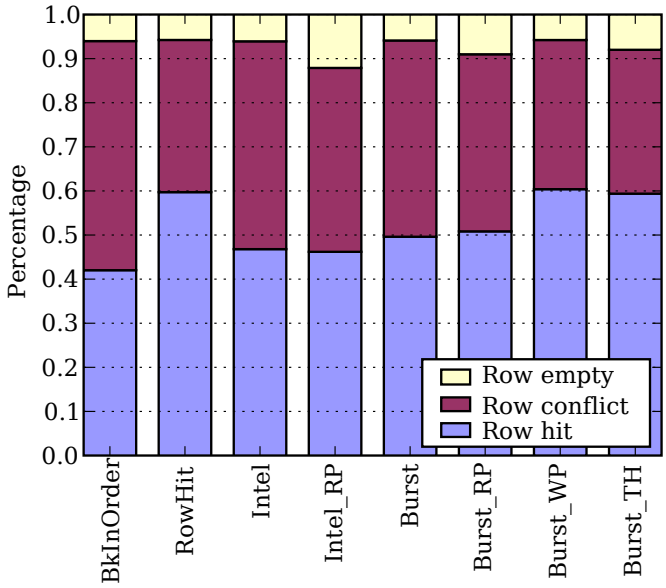

Figure 5.7: Average row hit, row conflict and row empty rate

are precharged after refreshing. With read preemption, an ongoing write interrupted by a read may have precharged the bank while having not yet initiated the row activate, causing the preempting read to be a row empty. Therefore Intel_RP, Burst_RP and Burst_TH have increased row empty rates.

5.4.3 Access Latency

When a read request is issued to the main memory, all in-flight instructions dependent upon this read request are blocked until the requested data is returned. Write requests, however, can complete immediately because no data needs to be returned. Therefore, one of the design goals of access reordering mechanisms is to reduce read latency.

Figure 5.8 shows the averaged read latency and write latency obtained by simulated access reordering mechanisms crossing all benchmarks. Compared to bank in order, all out of order access reordering mechanisms reduce read latency by a range of 20% to 47%; while all write latencies except for that of the row hit scheduling are increased by 54% to 280%.

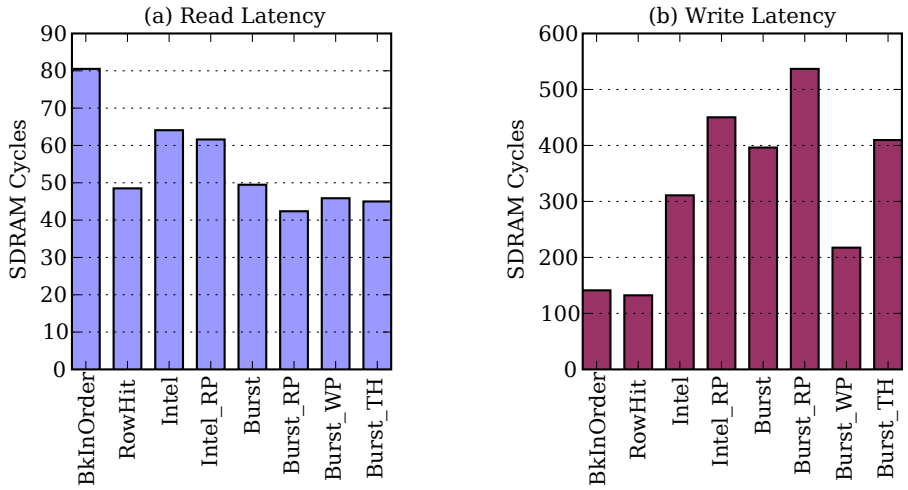

Figure 5.8: Access latency in SDRAM clock cycles

The row hit scheduling treats reads and writes equally and exploits row locality in both reads and writes, thus it reduces read and write latency and achieves the lowest write latency among all simulated access reordering mechanisms.

Intel and burst scheduling prioritize reads over writes. They reduce read latency at the cost of increase in write latency. Burst_RP has the lowest read latency because reads are not only prioritized over writes but also allowed to preempt ongoing writes. Read preemption helps Intel's scheduling to reduce read latency as well. While read preemption makes write latency even longer, write piggybacking greatly reduces write latency for Burst_WP because write are served faster and more row hits from writes are exploited through bursts.

To better understand the relationship between read and write latency, cumulative distributions of access latency and distributions of outstanding accesses for the *swim* benchmark are shown in Figure 5.9. Figure 5.9(a)(b) illustrates the percentage of accesses which experience a given latency. Figure 5.9(c)(d) shows the percentage of time which a given number of accesses are outstanding in main memory. For easy reading only 0% to 99% of cumulative distributions and 0% to 25% of normal distributions are shown in the figures.

The row hit scheduling slightly increases the number of outstanding accesses compared

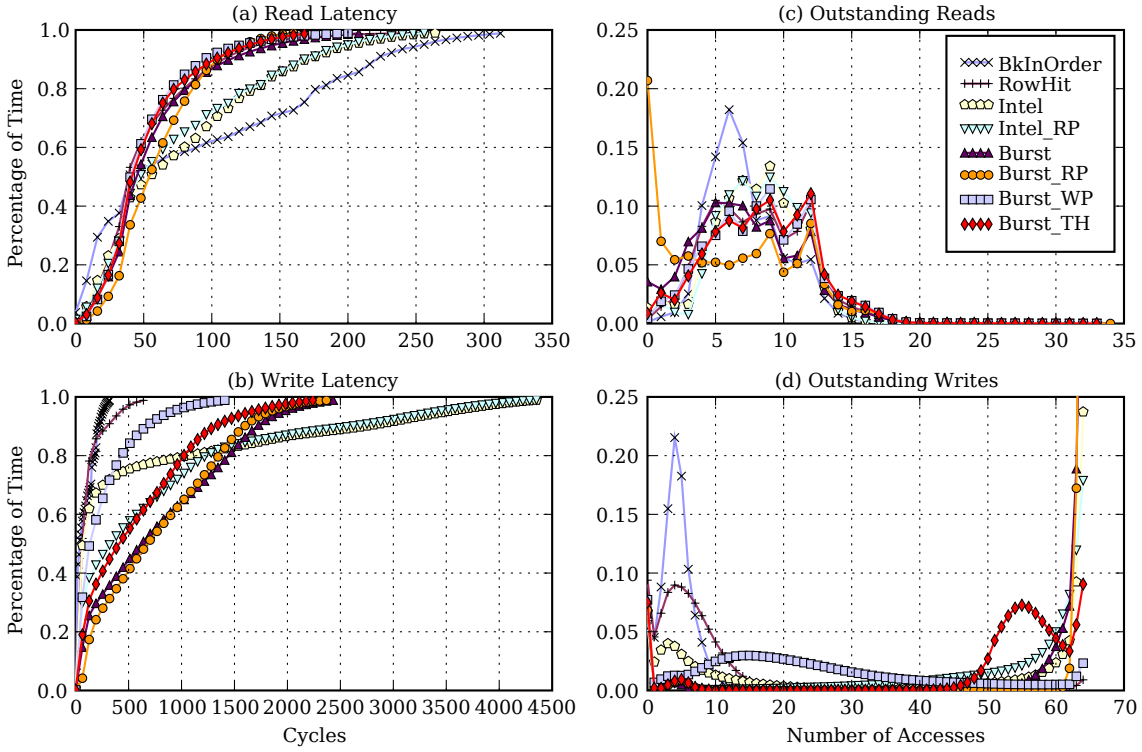

Figure 5.9: Cumulative distribution of access latency and distribution of outstanding accesses for the `swim` benchmark

to bank in order to allow row hits (both in reads and writes) to be served first. Intel and burst scheduling have large number of outstanding writes in the write queue due to postponed writes. Burst is more aggressive in prioritizing reads over writes than Intel's. As a result, Intel and burst scheduling cause write queue saturation 24% and 46% of time respectively for the `swim` benchmark. Read preemption reduces the number of outstanding reads but causes the write queue saturating even more frequently, i.e. Burst_RP causes write queue saturation 70% of time.

Prioritizing reads over writes can improve system performance as read latency which has impacts on performance is reduced. However, postponing writes increases the probability of write queue saturation, which may result in CPU pipeline stalls and cancel out the performance improvement gained by reduced read latency. Write piggybacking is hereby

employed to empty writes from the write queue without causing an undo increase in read latency. Consequently, Burst_WP only causes write queue saturation 2% of time. Burst_TH with a threshold of 52 makes a tradeoff between reducing read latency and preventing write queue saturation, resulting in a 9% write queue saturation rate. Therefore Burst_TH yields the best performance as following sections will show.

5.4.4 SDRAM Bus Utilization

The SDRAM bus utilization, which is the percentage of time that the bus is occupied, is shown in Figure 5.10. While there is less than 2% difference in address bus utilization among all simulated access reordering mechanisms, the data bus utilization varies in a range from 22% to 29%, which confirms that the data bus is more critical than the address bus. Given the simulated DDR2 PC2-6400 SDRAM, Burst_TH achieves the highest data bus utilization of 29%, therefore increases the effective memory bandwidth from 1.4GB/s (BkInOrder) to 1.9GB/s, resulting in a 32% improvement.

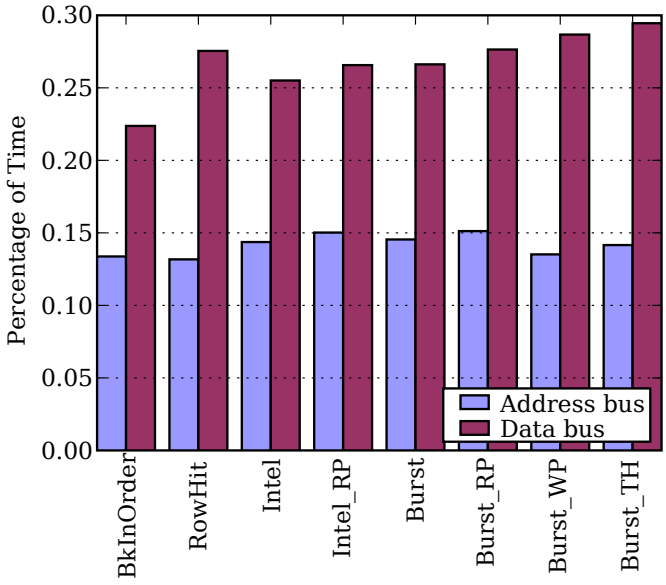

Figure 5.10: SDRAM bus utilization

5.4.5 Execution Time

Previous sections show that access reordering mechanisms can increase row hit rate and reduce access latency, therefore speedups in program executions are expected. Execution time of each individual benchmark under simulated access reordering mechanisms are examined in this section. For comparison, execution times are normalized to BkInOrder, as shown in Figure 5.11.

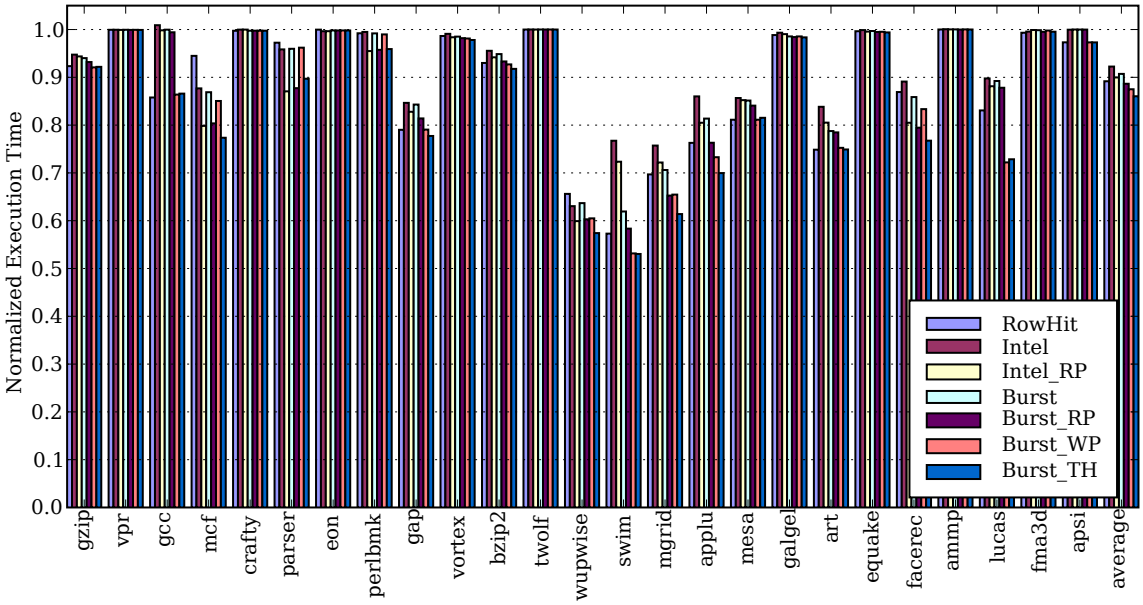

Figure 5.11: Execution time of access reordering mechanisms

RowHit achieves an average 11% reduction of execution time compared to BkInOrder. Intel and Burst (without read preemption and write piggybacking) reduce the execution time by 8% and 9% respectively. Read preemption alone contributes an average of 2% improvement on top of Intel and Burst. Write piggybacking alone contributes other 3% improvement on top of Burst, resulting in a total of 12% reduction of execution time by Burst_WP. Burst_TH which combines read preemption and write piggybacking through a static threshold of 52 yields the best performance among all simulated access reordering

mechanisms, achieving a 14% reduction in execution time crossing all simulated benchmarks, which equals to a 4% improvement over RowHit, 7% and 4% improvement over Intel and Intel_RP respectively.

Read preemption and write piggybacking have a varied impact dependent upon benchmark characteristics. For `mcf`, `parser`, `perlbmk` and `facerec`, read preemption contributes much greater performance improvement compared to write piggybacking. For the remainder of the benchmarks, write piggybacking generally results in more improvement than read preemption. Especially for `gcc` and `lucas`, Burst_WP achieves 14% and 28% reduction in execution time respectively.

It is desirable to take advantage of both read preemption and write piggybacking to achieve a maximal performance improvement. A static threshold is employed to dynamically switch between read preemption and write piggybacking as introduced in Section 5.3.2.2. The next section will show how this threshold affects the performance and how the optimized threshold is determined.

5.4.6 Threshold of Read Preemption and Write Piggybacking

Read preemption and write piggybacking have been shown to perform well on some benchmarks but not on all benchmarks. Which one has greater impact on performance is largely dependent on the memory access patterns of benchmarks. For example, allowing a critical read having many dependent instructions, to preempt an ongoing write may improve the performance. However, completing the ongoing write may prevent CPU pipeline stalls due to a saturated write queue, therefore improving the performance as well.

When the write queue has low occupancy, read preemption is desired to reduce read latency by allowing reads to bypass writes. When the write queue approaches its capacity, write piggybacking can keep the write queue from saturation. Therefore read preemption and write piggyback can be switched dynamically based on the write queue occupancy:

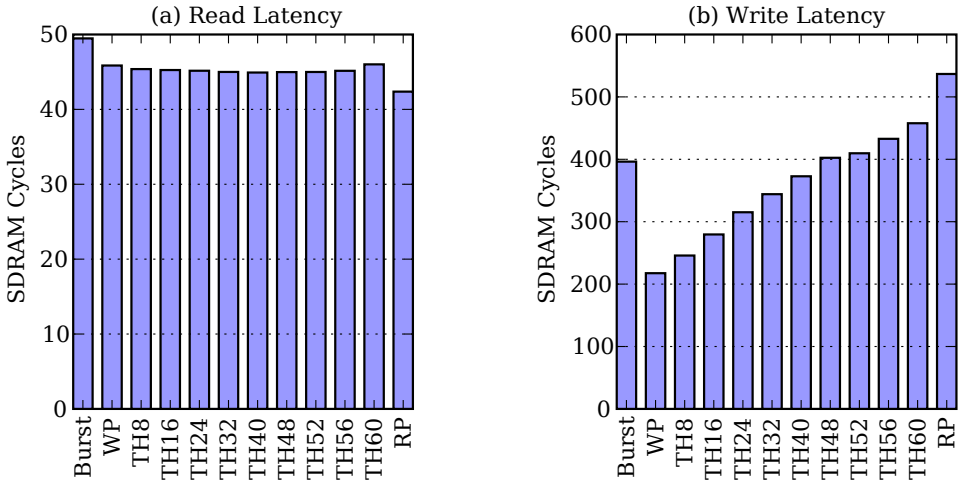

Figure 5.12: Access latency of burst scheduling with various thresholds

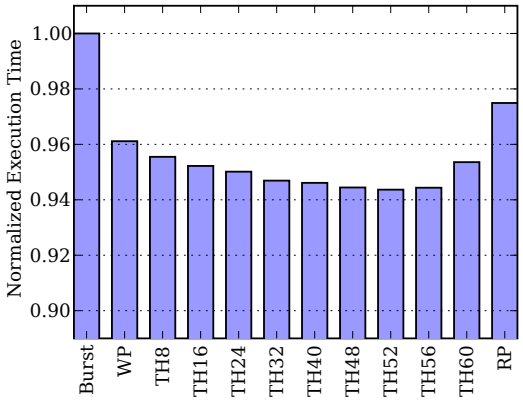

Figure 5.13: Execution time of burst scheduling with various thresholds

when the write queue occupancy is less than a certain threshold, read preemption is enabled; otherwise, write piggybacking is enabled.

To determine the threshold that yields the best performance, simulations with various thresholds are performed and the results are shown in Figure 5.12 and Figure 5.13. The execution times are averaged crossing all benchmarks and normalized to Burst. As the threshold increases, read latency first decreases because there are more reads resulting in shorter latencies by preempting writes. From threshold 40 read latency starts increasing

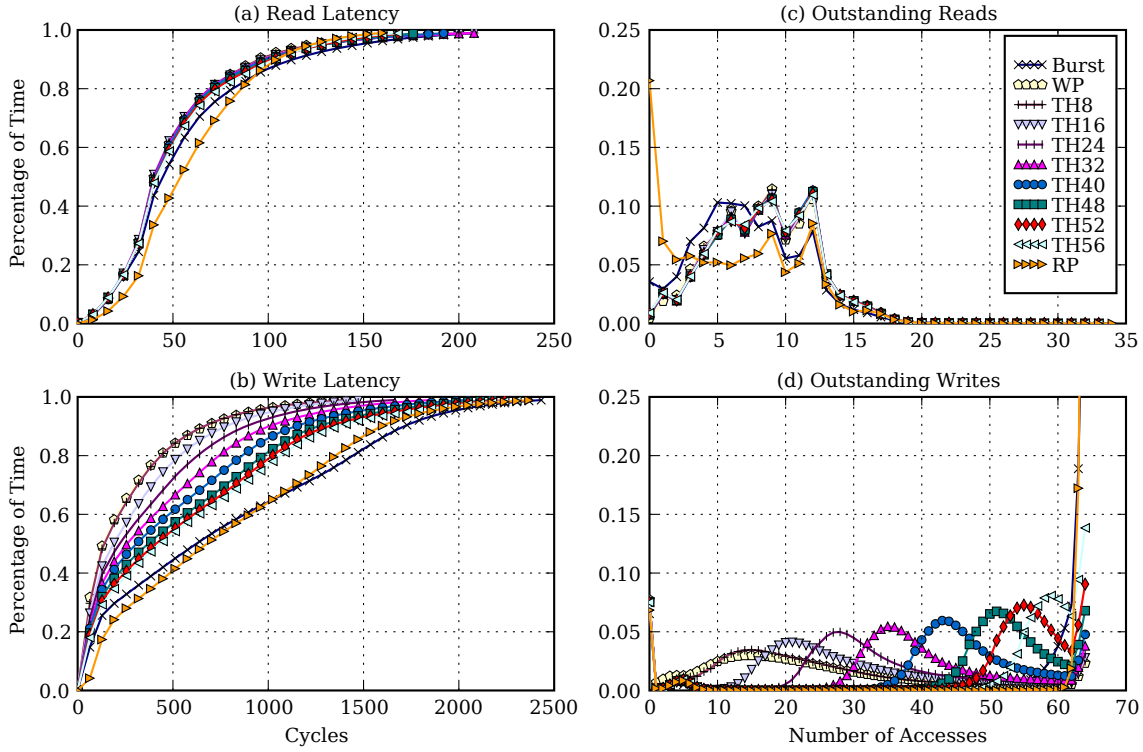

Figure 5.14: Cumulative distribution of access latency and distribution of outstanding accesses for the `swim` benchmark with various thresholds

mainly due to CPU pipeline stalls caused by increased occurrences of write queue saturation. Write latency increases as expected as the threshold increases. Execution time is determined by both read latency and write latency. According to Figure 5.13, the threshold 52 yields the lowest execution time crossing simulated benchmarks.

Using the same example `swim` benchmark as in Section 5.4.3, cumulative distribution of access latency and distribution of outstanding accesses with various thresholds are shown in Figure 5.14. Note that Burst_RP and Burst_WP are equivalents to Burst_TH64 and Burst_TH0 given that the write queue size is 64.

From Figure 5.14, Burst_RP has fewer outstanding reads than other thresholds, however, read latency of Burst_RP is slightly higher than others. This is because when there are fewer reads in the read queue, there are less chances for row hits to occur. In order for burst

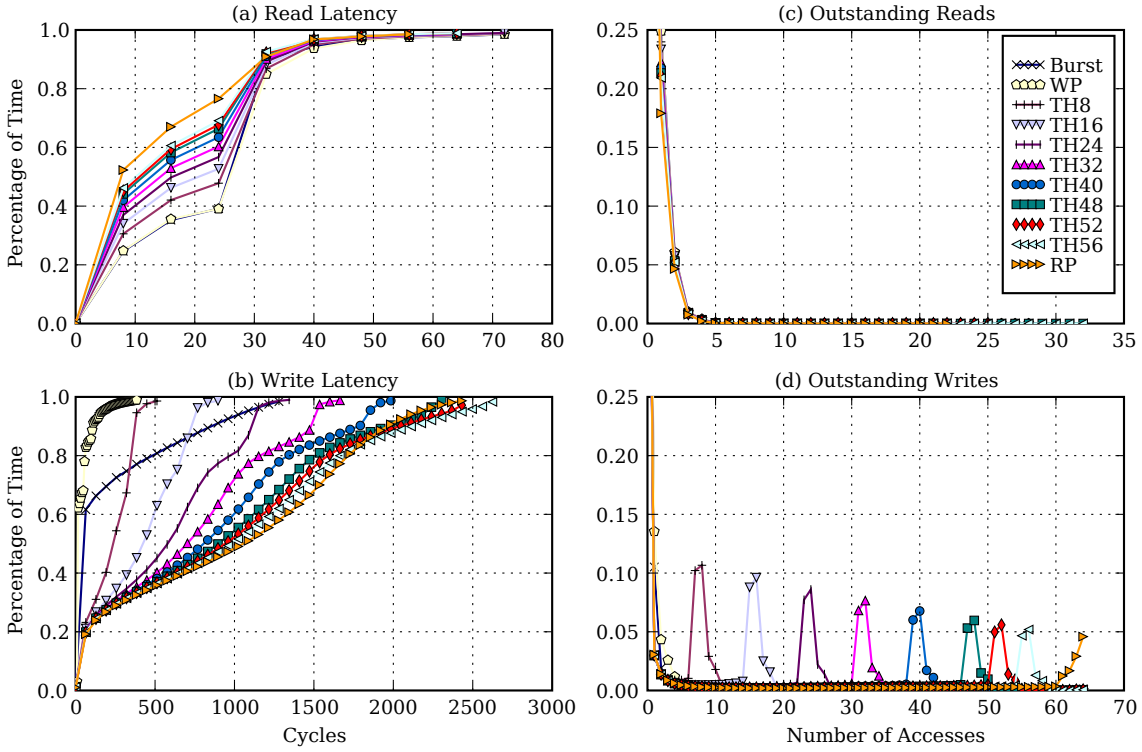

Figure 5.15: Cumulative distribution of access latency and distribution of outstanding accesses for the `parser` benchmark with various thresholds (Read preemption contributes most)

scheduling to create larger bursts and increase row hits, the read queue should contain a certain number of outstanding reads, and these reads should be served at a rate that will not deplete the read queue too quickly.

As the threshold increases from 0 to 64, the peak value of outstanding writes increases as well, as shown in Figure 5.14. The write buffer saturation rate is below 7% when the threshold is less than 48. The saturation rate increases to 14% at threshold 56 then jumps to 70% at threshold 64 (Burst_RP). The earlier write piggybacking is enabled, the less frequently the write queue saturation will occur.

Figure 5.15 and Figure 5.16 show two other benchmarks, `parser` and `lucas`. Among all benchmarks simulated, read preemption contributes the most performance improvement

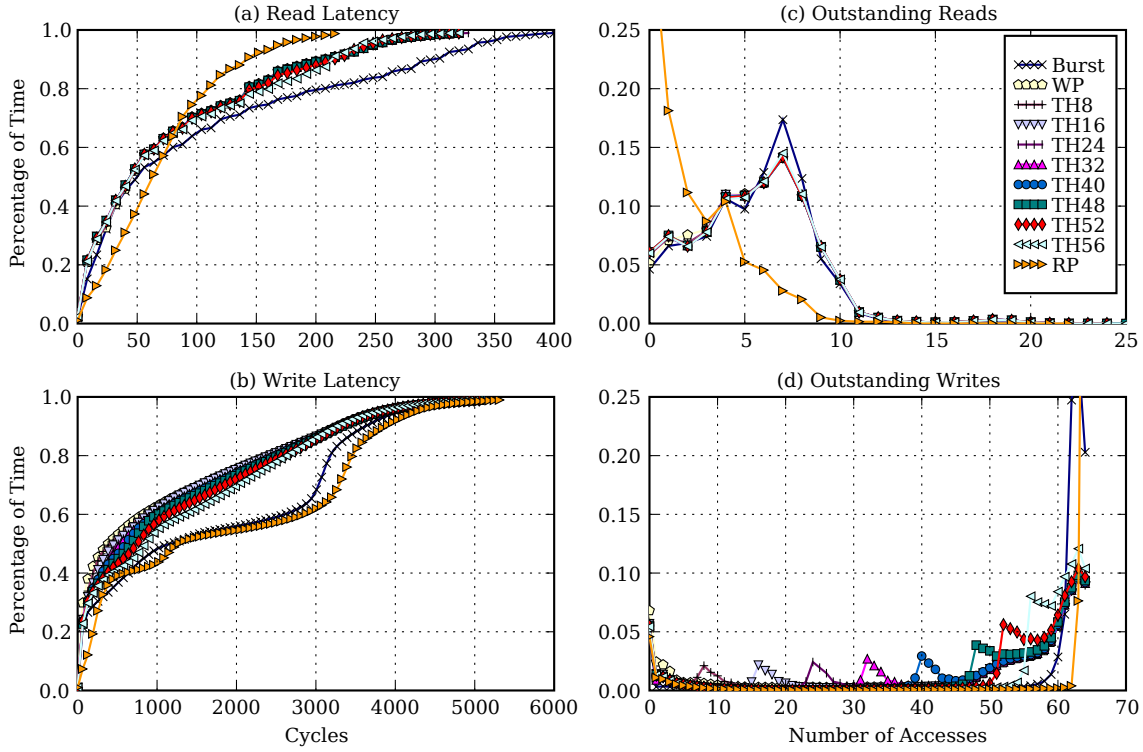

Figure 5.16: Cumulative distribution of access latency and distribution of outstanding accesses for the `lucas` benchmark with various thresholds (Write piggybacking contributes most)

over Burst on `parser`, while write piggybacking contributes the most on `lucas`.

As shown in Figure 5.15, Burst_RP (TH64) has the lowest read latency compared to other thresholds. Burst_RP does have one of the highest write latencies, however, it does not cause too much write queue saturations, which only happens 5% percent of time. Low read latency and acceptable write queue saturation rate explain why Burst_RP has the best performance among all thresholds on `parser`. And obviously the performance of `parser` is largely depends on the read latency.

As shown in Figure 5.16, Burst_RP is able to keep the number of outstanding reads of benchmark `lucas` below 5 for 95% of time, resulting in a relative short read latency. However, the write queue saturates at 78% of time due to read preemption. A lower

threshold reduces the write queue saturate rate. Apparently `lucas` is more sensitive to write latency, therefore Burst_WP achieves the lowest execution time among all thresholds.

5.5 Adaptive Threshold Burst Scheduling

A static threshold of 52, as shown in Figure 5.13, results in the shortest execution time crossing simulated benchmarks. Intuitively a static threshold may not deliver the best performance for each individual benchmark, because memory access pattern various between benchmarks and program behavior also alters when execution phase changes. A preliminary study of performance impacts of an adaptive threshold is hereby performed.

Figure 5.17 confirms the above statement that a static threshold may not work best for each benchmark. Execution times are normalized to burst scheduling without read preemption and write piggybacking. To better illustrate the effect of the threshold, 9 benchmarks that show less than 2% difference in execution time under various thresholds are not shown in Figure 5.17.

Among the 16 benchmarks shown, 7 benchmarks exhibit a declining trend in execution time as the threshold increases. This means a higher threshold helps to improve the performance on these benchmarks by allowing more reads to preempt writes. Contrarily 12 benchmarks have a significantly increased execution time with Burst_RP (threshold 64), meaning that turning write piggybacking completely off will degrade the performance.

Based upon the above observations, an adaptive threshold should be able to find a threshold which is best suited to program behavior, allowing reads to prioritize writes meanwhile preventing write queue saturation which may cause CPU pipeline stalls. While program behavior varies between benchmarks, memory access pattern also changes when a benchmark transits between phases of execution. Thus an adaptive threshold should also be able to seek out the threshold appropriate to changes in program memory access pattern relatively quickly.

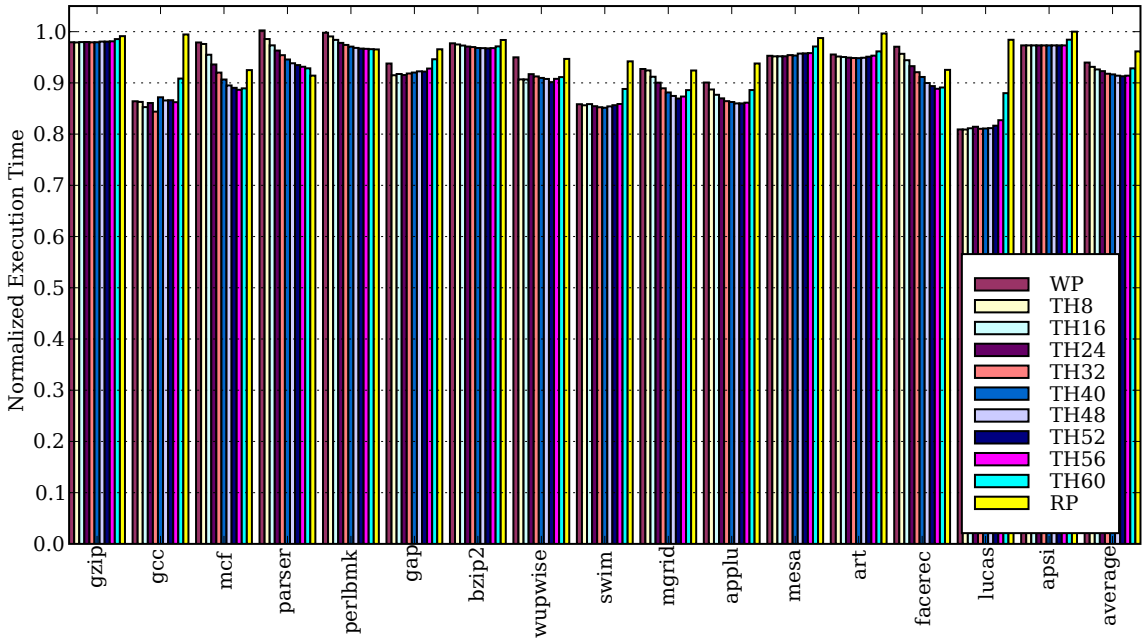

Figure 5.17: Burst scheduling with various static thresholds on selected benchmarks

5.5.1 Performance Improvement Space of Adaptive Threshold

While an upper bound of performance improvement that an adaptive threshold can contribute may be difficult, if not impossible, to be obtained, evaluating various thresholds for each individual benchmark gives one possible estimation of performance improvement which an adaptive threshold could contribute.

As shown in Figure 5.17, if the threshold is individually selected for each benchmark, the average reduction of execution time over Burst is 9.2%, which is 0.5% greater than the 8.7% reduction of execution time achieved by Burst_TH (threshold 52). For some benchmarks such as `gcc` and `parser`, threshold 32 and threshold 64 outperform Burst_TH by 2.2% and 2.1% respectively, meaning an adaptive threshold could improve performance more significantly on these two benchmarks.

However, the above performance estimation of an adaptive threshold only considers the

overall difference in memory access pattern between benchmarks. The variety of access pattern when a program transits between execution phases is not considered. Therefore, a well designed adaptive threshold burst scheduling should be able to contribute more performance improvement than the above estimation.

5.5.2 History-based Adaptive Threshold

The proposed history-based adaptive threshold is one possible adaptive threshold algorithm. Using history information, history-based adaptive threshold adjusts the threshold on the fly to catch up program's read write ratio.

A history record of received and scheduled accesses, including the total number of received as well as scheduled reads and writes in a given sized history window, is maintained during runtime and used to periodically update the threshold.

Figure 5.18 illustrates the algorithm of history-based adaptive threshold. Received read rate and scheduled read rate are calculated based on history information using Equation 5.1 and 5.2. When received read rate is greater than scheduled read rate, more reads then writes are accumulated in the access queue. In this case, the threshold is increased by one, allowing more reads to preempt writes. If the received read rate is smaller than scheduled read rate, writes are queuing in the write queue. To prevent write queue saturation, the threshold is decreased by one, allowing writes to be piggybacked with bursts. Special consideration is given to write queue saturation, because it may cause CPU pipeline stall and degrade the performance. If the write queue is saturated and its overall saturation rate is greater than a preset value (5%), the threshold will be reduced by 50%. This promotes write piggybacking and alleviates the penalty of write queue saturation quickly.

$$received\_read\_rate = \frac{received\_reads}{received\_reads + received\_writes} \tag{5.1}$$

$$scheduled\_read\_rate = \frac{scheduled\_reads}{scheduled\_reads + scheduled\_writes} \tag{5.2}$$

```
1:  if new access received or access scheduled then
2:      update access history
    end if
3:  if time to update threshold then
4:      if write queue is saturated and
           write queue saturation rate > 5% and
           threshold/2 > min_threshold then
5:         threshold = threshold/2
6:      else if received_read_rate > scheduled_read_rate and
              threshold < max_threshold then
7:         threshold = threshold + 1
8:      else if received_read_rate < scheduled_read_rate and
              threshold > min_threshold then
9:         threshold = threshold − 1
        end if
    end if
```

Figure 5.18: A history-based adaptive threshold algorithm

History-base adaptive threshold has two parameters, history length (l) and update interval (i). Information of the last l received accesses and the last l scheduled accesses are maintained in two circular buffers respectively. Circular buffers are employed because of their filter effect which smoothes out accidental changes in access pattern. Threshold is updated at every i memory cycles or every i memory accesses. A small i means the algorithm is sensitive to short-term changes in the access pattern, while a large i results in a better tracking of long-term access pattern changes.

5.5.2.1 Performance of History-based Adaptive Threshold

A performance comparison between static threshold Burst_TH52 and the proposed history-based adaptive threshold are shown in Figure 5.19. In the figure, Burst_AT128_64(c) is referred to the history-based adaptive threshold with a history length of 128 and an update interval of 64 memory cycles. The letter in the last parenthesis denotes the unit of the update interval. A 'c' means memory cycle and a 'm' means received memory access.

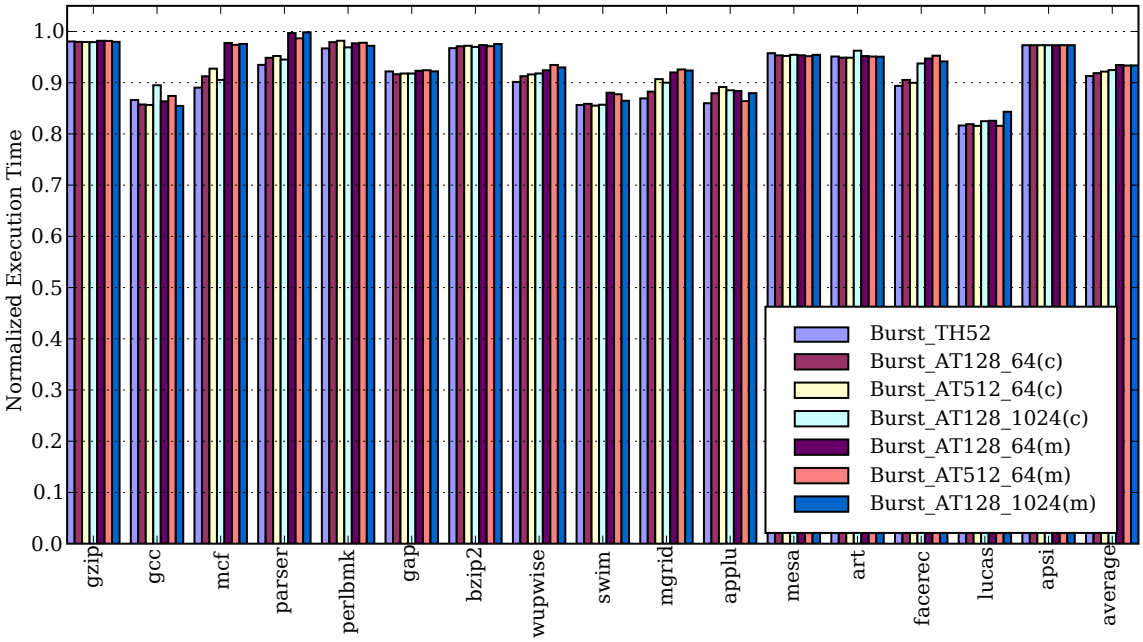

Figure 5.19: Burst scheduling with static and adaptive threshold on selected benchmarks

As a limited exploitation of the design space, the history length is increased to 512 with Burst_AT512_64(c|m). In another scenario, the update interval is enlarged to 1024 with Burst_AT128_1024(c|m). Both memory cycle and memory access are used as update unit.

As shown in Figure 5.19, Burst_AT128_64(c) and Burst_AT512_64(c) are able to outperform Burst_TH52 on selected benchmarks, including `gcc`, `gap`, `mesa` and `art`, although the performance improvements are trivial. For the rest benchmarks, Burst_TH52 still holds the best performance. On average, a long history and a long update interval both degrade the performance, meaning a sensitive and frequently updated threshold is desired.

Switching the update unit from memory cycle to memory access does not help improving the performance. This because the average arrival interval between memory accesses are generally larger than a memory cycle. Using memory access as the update unit results in a slower updating.

5.5.2.2 Distribution of the Adaptive Threshold

Figure 5.20 shows the distribution of the adaptive threshold on four selected benchmarks, on which the adaptive threshold has better performance then Burst_TH52. As shown in Figure 3.7(a), benchmark `gcc` has 94% of writes. As a result, 60% of time the threshold of `gcc` is 0, which means read preemption is disabled and write piggybacking is enabled for the most of time, allowing writes to be served quickly. In contrary to `gcc`, benchmark `art` has 83% of read accesses. Therefore the threshold of `art`, as shown in Figure 5.20(d), is pushed to the upper side, allowing reads to preempt writes.

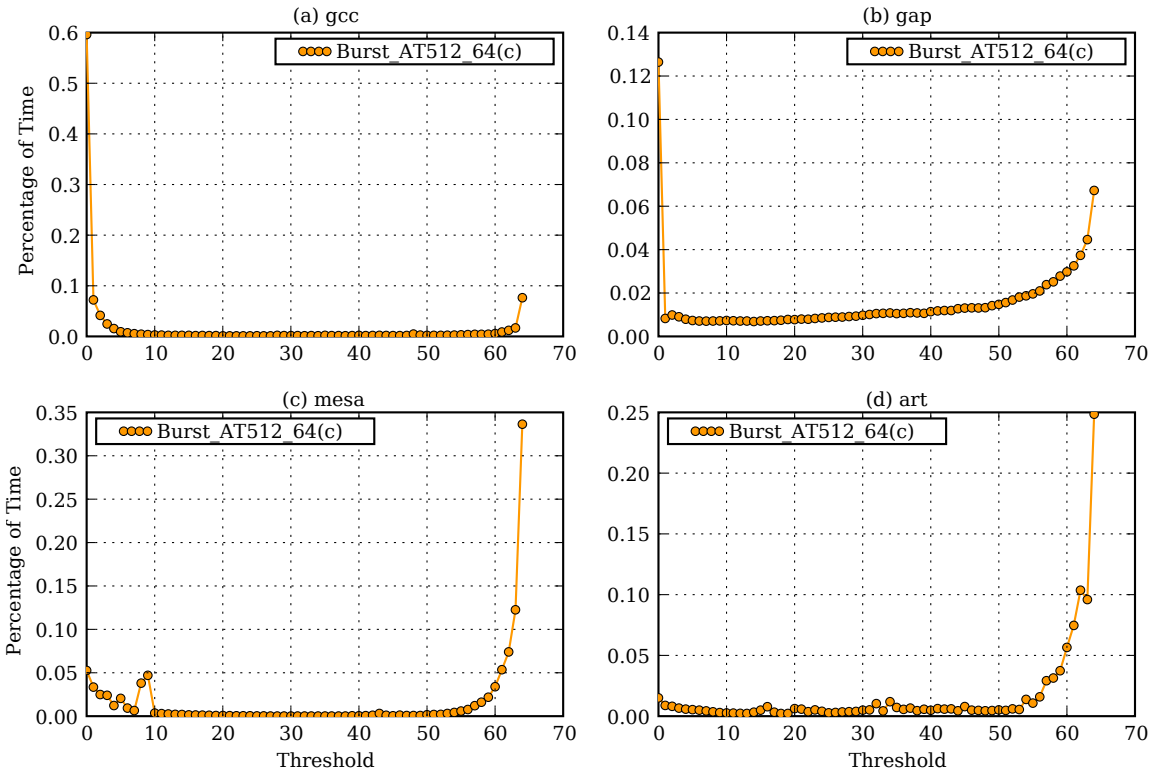

Figure 5.20: Adaptive threshold distribution on selected benchmarks

Benchmark `gap` and `mesa` have mixed read and write accesses. When the program transits its execution phases, the main memory access stream alternates between read intensive and write intensive. The adaptive threshold is able to catch up this transition and

move toward to the direction which will match program's read write rate, as illustrated in Figure 5.20(b)(c).

Although numerous adaptive threshold experiments have been performed, the proposed history-based adaptive threshold does not deliver a better performance than the static threshold on average. However, an improved or fine-tuned adaptive threshold algorithm should have the potential to outperform the static threshold at the cost of increased complexity.

5.6 Access Reordering Combines with Address Mapping

SDRAM address mapping techniques, as presented in Chapter 4, attempt to distribute accesses evenly among all banks, therefore enabling bank parallelism and reducing potential row conflicts. Simulation results shown in Chapter 4 are obtained by a revised SimpleScalar v3.0d with the SDRAM module v1.0 using an in order memory access scheduler.

Access reordering mechanisms studied in this Chapter do not affect access distribution. Instead, access reordering mechanisms change the order in which memory accesses are executed to increase and exploit row locality. SDRAM address mapping and access reordering use different approaches to address the same issue of long main memory access latency. Therefore these two techniques are hereby combined to examine whether they can achieve a performance improvement when working together which is greater than the performance improvements each technique can contribute individually.

Using the revised M5 simulator v1.1 with the SDRAM module v2.0 and the same baseline machine configuration as shown in Table 3.2, five SDRAM address mapping techniques and four access reorder mechanisms, a total of 20 combinations, are simulated. The results are shown in Figure 5.21. Execution time of each combination is normalized to paging interleaving with bank in order scheduling and averaged crossing all simulated benchmarks.

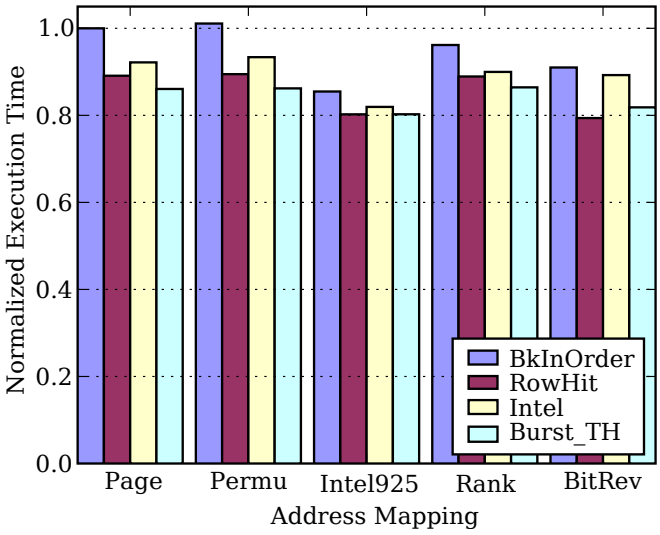

Figure 5.21: Access reordering working in conjunction with address mapping

5.6.1 Performance Variation of SDRAM Address Mapping

Before examining the combined performance of SDRAM address mapping and access reordering, the performance of SDRAM address mapping techniques are found differently in Figure 5.21 as the previous results shown in Figure 4.12 of Chapter 4. This performance variation is mainly due to numerous variances between the two simulation environments and configurations as explained below.

SDRAM address mapping studies, presented in Chapter 4, use a revised SimpleScalar v3.0d with the SDRAM module v1.0, while studies of access reordering use by a revised M5 simulator v1.1 and the SDRAM module v2.0. The two simulated machines have different configurations as shown in Table 3.1 and Table 3.2 respectively. Such differences include simulated CPU, DDR 400 SDRAM vs. DDR2 800 SDRAM, single channel vs. dual channel and etc. In addition, the in order memory access scheduler used by the SDRAM module v1.0, as discussed in Section 3.2.1.2, differs from any access reordering mechanisms simulated in this chapter using the SDRAM module v2.0.

However, the biggest difference is the memory channel configuration. Although a dual

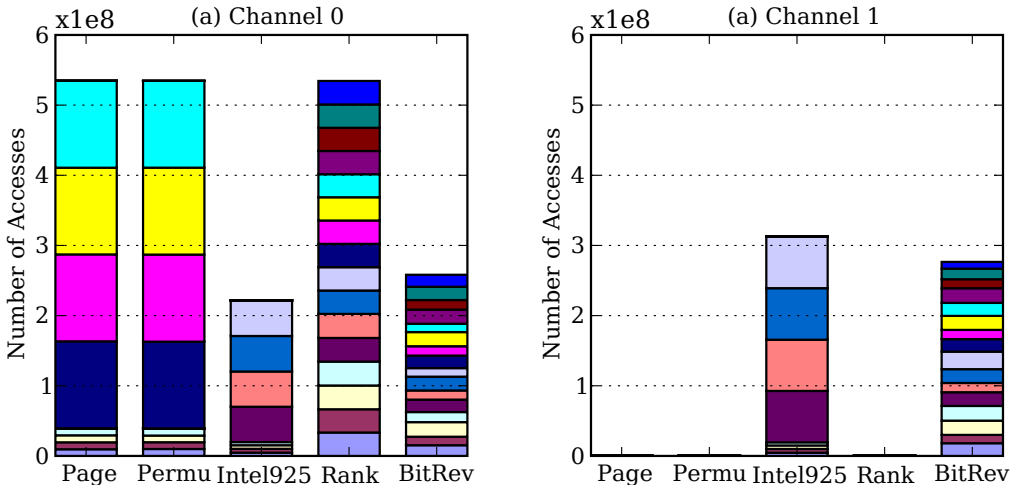

Figure 5.22: Access distribution in a dual channel system

channel main memory is used in simulations whose results are shown in Figure 5.21, the effective usage of the two channels are totally dependent upon the selected address mapping, given no virtual paging system. Figure 5.22 illustrates the main memory access distribution of two channels. According to the memory configuration, each channel has 16 banks, which are shown in different colors in Figure 5.22. The raw numbers of accesses directed to each bank are shown in the figure. Please note access reordering mechanisms do not affect access distribution, thus Figure 5.22 represents any simulated access reordering mechanisms.

From Figure 5.22, page interleaving, permutation-based page interleaving and rank interleaving address mapping only utilize one channel. Intel's 925X chipset and bit-reversal address mapping have roughly balanced loads to each channel. However, the symmetric dual channel mode is used by Intel's 925X chipset, as illustrated in Figure 2.13, while bit-reversal uses dual channels asymmetrically. With symmetric dual channel mode, two 64-bit memory channels are clustered to create one logic 128-bit channel, doubling the available memory bandwidth. Appendix A does a comparison between single channel, asymmetric dual channel and symmetric dual channel. Simulation results of Appendix A show that asymmetric dual channel achieves an average of 6% performance improvement over single

Table 5.4: Top five combinations of address mapping and access reordering

Rank	Access mapping	Access reordering	Performance improvement
1	Bit-reversal	Row hit scheduling	20.6%
2	Intel's 925X chipset	Burst scheduling	19.8%
2	Intel's 925X chipset	Row hit scheduling	19.8%
4	Bit-reversal	Burst scheduling	18.2%
5	Intel's 925X chipset	Intel's scheduling	18.1%

channel, while symmetric dual channel outperforms asymmetric dual channel and achieves a 13% performance improvement, as illustrated in Figure A.1.

The difference in channel usages explains why Intel's 925X chipset address mapping generally outperforms other address mapping techniques as shown in Figure 5.21.

5.6.2 Combined Performance

According to Figure 5.21, among 20 combinations of various address mapping techniques and access reordering mechanisms, the top five combinations which have the shorted execution time crossing all simulated benchmarks are listed in Table 5.4. The performance improvement is measured by comparing the execution time of each combination to that of page interleaving with bank in order scheduling.

The best combination is bit-reversal address mapping with the row hit scheduling, which reduces the average execution time by 20.6% over page interleaving with bank in order scheduling. When used individually, both bit-reversal address mapping and burst scheduling (with static threshold 52) achieve the best performance among simulated address mapping techniques and access reordering mechanisms respectively. When combined, bit-reversal and burst scheduling achieve a 18.2% reduction in average execution time.

Intel's 925X chipset address mapping performs well in the simulations presented here mainly due to its symmetric dual channel mode, as discussed in Section 5.6.1. Bit-reversal address mapping is expected to have a better performance when symmetric dual channel

mode is in use. Despite its simplicity, the row hit access scheduling performs well when combined with various address mapping techniques, which makes the row hit access scheduling a good choice for situations where resource (chip area) is limited.

While there is no guarantee that the combination is optimal, bit-reversal address mapping and burst scheduling can work together constructively to achieve a significant reduction in execution time over conventional techniques.

The simulated machine does not have virtual paging system. As virtual paging system may destroy spatial locality as discussed in Section 4.4.2, SDRAM address mapping may contribute less performance improvements when virtual paging is in use. However, access reordering mechanisms will catch up and continue to improve the performance.

Chapter 6

Conclusions and Future Work

As the performance gap between microprocessors and main memory continues to increase, the main memory access latency remains a factor limiting system performance. SDRAM address mapping and access reordering mechanisms are two techniques that can efficiently reduce main memory access latency. The proposed bit-reversal address mapping and burst scheduling have been shown through simulations to favorably compare with existing address mapping techniques and access reordering mechanisms. From the results presented in this thesis, conclusions of this research work are drawn. Future work are briefly discussed and promising avenues for future study are presented.

6.1 Main Memory Issue and Research Goal

In the last decades, the rate of improvement in microprocessor speed exceeded the rate of improvement in memory speed. As the performance gap between microprocessors and main memory continues to increase, eventually system performance would be entirely determined by memory speed, which is also known as "hitting the memory wall" [73].

The limiting factors of memory are bandwidth and latency. Memory bandwidth is a problem which can be largely solved by increasing resources, such as increasing data

bus width, bus clock frequency, doubling data clocking rate and using multiple channels. Memory access latency, on the other hand, is more difficult to address. In a typical desktop computer with a 2.5GHz or faster CPU, a main memory access typically takes hundreds even thousands of CPU cycles to complete.

There are techniques used to address the latency issue of memory accesses. For instance, caches built with fast SRAM devices are commonly used to reduce the traffic to the slow main memory therefore reduce the average latency of memory accesses. Also an out-of-order execution processor can hide the latency of a memory access by executing subsequently independent instructions. Other techniques attempt to improve the performance of SDRAM devices, i.e. adding a cache on the SDRAM device to reduce the average SDRAM access latency. The research goal of this thesis is to reduce the observed main memory access latency without any modifications to microprocessors or SDRAM devices.

6.1.1 Characteristics of Modern SDRAM Devices

Due the 3-D structure (bank, row and column), SDRAM devices have nonuniform access latencies. An access to an SDRAM device could be a row hit, row empty or row conflict, and each would experience a different latency. Modern SDRAM devices have two key features: multiple internal banks and row cache (sense amplifiers). Multiple internal banks allow accesses to different banks to be executed simultaneously, subject to timing constraints. Row cache keeps the most recently accessed row data so that subsequent accesses to the same row will be row hits and have short latencies.

To create a main memory hierarchy, multiple SDRAM devices are first concatenated to create ranks in order to fill the main memory data bus. Multiple ranks compose a channel, which is then duplicated to create multi-channel configuration. Multiple channels and multiple banks inside each channel provide parallelism in main memory which can be exploited to reduce memory access latency. To access a memory block in the main memory,

the address of the requested memory block needs to be translated into an SDRAM address, which consists of channel index, rank index, bank index, row index, column index and byte index. This translation process, as well as the mechanism used to optimize for reduced latency or power consumption, is known as SDRAM address mapping.

6.1.2 Main Memory Access Stream Properties

With the existence of caches, main memory accesses are actually cache misses. A statistical study of main memory access stream presented in Section 4.1 shows that spatial locality and temporal locality are available in main memory access streams, even after being filtered by caches. As shown in Figure 4.2, lower order address bits are more likely to change between temporally adjacent accesses, implying spatial locality. In Figure 4.1 memory blocks are shown to be reused in the near future, leading to temporal locality.

SDRAM row cache (sense amplifiers) can capture spatial locality and temporal locality within SDRAM rows. Accesses to an open row result in row hits, which have the shortest possible access latency. However, as spatial locality continues beyond SDRAM rows, adjacent SDRAM rows within the same bank are likely to be accessed one by one, causing a series of row conflicts therefore degrading the performance.

6.1.3 Techniques to Reduce Main Memory Access Latency

Because of the nonuniform access latency of SDRAM devices, an application's memory access pattern has a significant impact upon execution time. Studies show that the memory access stream contains spatial and temporal locality. And parallelisms are available in main memory, which may encompass multiple channels and multiple banks. Locality and parallelism are not fully exploited by current SDRAM controllers.

SDRAM address mapping techniques exploit the parallelism provided by main memory by distributing memory accesses evenly to all available SDRAM banks, meanwhile reducing

potential row conflicts. Access reordering mechanisms attempt to create more row hits by changing the order that memory accesses are executed.

6.2 Conclusions of Bit-reversal SDRAM Address Mapping

SDRAM address mapping is a protocol of interpreting physical address bits into an SDRAM address. Address mapping can change the locality presented in the access stream therefore impacts the available parallelism, latency and performance.

Based on the observation that lower physical address bits have statistically higher probability to change between accesses then higher order bits, as shown in Figure 4.2, the bit-reversal address mapping reverses the higher order physical address bits, such that the indexes for large components of main memory (such as channel, rank and bank) are mapped from the physical bits that are most likely to change from access to access. Unlike traditional page interleaving, which only interleaves accesses between internal banks of SDRAM devices, the bit-reversal address mapping attempts to distribute accesses evenly across the entire SDRAM space.

Like other SDRAM address mapping techniques, the bit-reversal uses the lowest order physical address bits which have the highest probability of change as column index, so that spatial locality within SDRAM rows can be captured by the SDRAM row cache, maximizing row hits. On the other hand, bit-reversal directs potential row conflicts to different banks, converting spatial locality above the SDRAM row size, which can cause row conflicts as discussed in Section 4.1.3, into bank parallelism.

6.2.1 Depth of Reversal

A further study shows that during a cache conflict miss, the new cache line to be loaded into the cache and the cache line (assuming it is dirty) to be written back into the main memory have the same cache index but different cache tags. These read/write access pair

during cache conflict misses may cause row conflict as discussed in Section 4.3.1. To address this issue, the depth of reversal is introduced. Identified through experiments, the depth of reversal that yields the shortest execution time across all benchmarks should be one less than the width of the lowest level cache tag, based on the simulation results shown in Figure 4.4.

6.2.2 Performance of Bit-reversal Address Mapping

Using a revised SimpleScalar v3.0d simulator and the SDRAM module v1.0, the bit-reversal address mapping along with other existing address mapping techniques are evaluated on the SPEC CPU2000 benchmark suite. The bit-reversal address mapping with the depth 15 is compared with existing address mapping techniques, including page interleaving, permutation-based page interleaving, Intel's 925X chipset and rank interleaving. According to Figure 4.12, the bit-reversal address mapping achieves a 14% of reduction in execution time over page interleaving across all simulated benchmarks. Bit-reversal also outperforms permutation-based page interleaving, Intel's 925X chipset and rank interleaving by 12%, 14% and 3% respectively.

6.2.3 Bit-reversal under Controller Policies and Virtual Paging

Limited studies of the effects of virtual paging and SDRAM controller policies are performed. The bit-reversal address mapping interacts constructively with static or dynamic SDRAM controller policies to further reduce the memory access latency, as illustrated in Figure 4.13. DYN-UPB policy provides a theoretical upper bound on the performance improvement which a dynamic controller policy could achieve.

As shown in Figure 4.14, virtual paging may reduce the performance improvement contributed by address mapping. This is because virtual paging may disturb the locality presented in main memory access stream. The bit-reversal address mapping works well

with no virtual paging or sequentially allocated virtual paging. Locality is completely lost due to random page allocation, therefore all address mapping techniques simulated show little difference with randomly allocated virtual paging.

Bit-reversal address mapping is easy to implement and requires no modification to SDRAM devices or processors. Nevertheless, it is an effective technique to exploit both locality in the main memory access stream and parallelism between SDRAM banks/channels. The bit-reversal address mapping works well with various controller policies and is especially useful in embedded systems which do not incorporate virtual paging.

6.3 Conclusions of Burst Scheduling Access Reordering

Access reordering mechanisms change the order that memory accesses are executed to reduce the observed access latency. One necessity of access reordering mechanisms is to have multiple memory accesses pending at the main memory, which is common with an out-of-order execution superscalar processor. Unlike SDRAM address mapping techniques which distribute potential row conflicts to different banks to reduce row conflict rate, access reordering mechanisms attempt to create row hits or avoid row conflicts by selecting accesses from all pending accesses.

6.3.1 Key Features of Burst Scheduling

Memory scheduling techniques improve system performance by changing the sequence of memory accesses to increase row hits and avoid row conflicts, therefore reducing average memory access latency. Inspired by existing academic and industrial access reordering mechanisms, burst scheduling access reordering mechanism is hereby being proposed with the goal of improving performance of existing access reordering mechanisms and addressing their shortcomings.

- **A Two-level Scheduler**

  Burst scheduling is a two-level scheduler performing access scheduling at both memory access level and SDRAM transaction level. Access level scheduling takes place at the bank queues, where access execution order within each bank is determined by the bank arbiter. Whether an access is a row hit, row conflict or row empty is determined after access level scheduling. Then accesses from different banks are sent to the bus transaction scheduler, where transactions belonging to different ongoing accesses are interleaved. The bus transaction scheduler performs a fine-tuned transaction level scheduling. It considers SDRAM devices state and handles bus contentions with the goal of maximizing data bus utilization.

- **Read Bursts**

  Within each bank, accesses are stored in unique read queue and write queue. Accesses in the read queue are organized in bursts, which is composed by accesses to the same row of the same bank. Bursts in the read queue are sorted by the arrival time of the head access of each burst.

- **Reads Prioritize Writes**

  The write queue is fully associative, it functions like a write buffer, allowing reads to bypassing writes and forwarding data from writes to subsequent reads in case the reads requesting the same memory blocks as the previous writes.

- **Burst Interleaving**

  Bursts contains all row hits, whose data transactions can be performed on the data bus in back to back cycles. While scheduling an entire burst, the data bus reaches the maximal utilization which may result in long latency to accesses belonging to bursts from other banks. Therefore bursts from different banks are carefully interleaved. High data bus utilization can be maintained with burst interleaving.

6.3.2 Improvements to Burst Scheduling

Two improvements, read preemption and write piggybacking, are made to burst scheduling. Read preemption allows reads to interrupt ongoing writes to reduce read latency. Write piggybacking appends qualified writes at the end of bursts to exploit row hits in writes as well as prevent write queue saturation which may cause CPU pipeline stalls. Subject to a static threshold read preemption and write piggybacking are switched dynamically based on the write queue occupancy to achieve an improved reduction of execution time by these two improvements. Further performance improvements may be obtained at the cost of extra complexity by using an adaptive threshold algorithm, which updates the threshold periodically when the program is running based on certain criteria, such as read write ratio.

6.3.3 Performance of Burst Scheduling

Using a revised M5 simulator v1.1 with the SDRAM module v2.0, burst scheduling access reordering is examined and compared to existing access reordering mechanisms, including bank in order scheduling, the row hit scheduling and Intel's out of order memory scheduling. The performance contributions of read preemption and write piggybacking are studied and identified. The threshold that yields the best performance is determined by experiments. As shown in Figure 5.11, burst scheduling with a threshold of 52 achieves an average of 14% reduction in execution time over bank in order scheduling for all simulated SPEC CPU2000 benchmarks. Burst scheduling also outperforms the row hit scheduling and Intel's out of order scheduling by 4% and 7% respectively.

The combined performance of access reordering and SDRAM address mapping are examined in Section 5.6. The burst scheduling works constructively with the bit-reversal address mapping, achieving an 18% reduction in execution over bank in order scheduling with page interleaving address mapping. The best combination is the bit-reversal address mapping with the row hit scheduling, which reduces the execution time by 21%.

6.4 Future Work

While the studies presented in this thesis of SDRAM address mapping techniques and access reordering mechanisms are complete, there are further improvements and future work to this research, which are briefly discussed in this section.

6.4.1 Future Study of Access Reordering Mechanisms

Burst scheduling with a static threshold works well on average, as shown in Figure 5.11. However, Figure 5.17 illustrates that individually selected thresholds for each benchmark can yield an even better performance then that of a static threshold. An adaptive threshold can automatically match differences in memory access patterns between benchmarks as well as changes of each benchmark during execution phase transitions. Although the proposed history-based adaptive threshold algorithm, as discussed in Section 5.5.2, does not outperform the static threshold on average, a more advanced adaptive threshold algorithm should have the potential to deliver improved performance over the static threshold, at the cost of increased complexity.

Modern microprocessors have a memory controller integrated on the CPU die to increase memory bandwidth and reduce main memory access latency. Because of the tighter connection between integrated memory controller and the CPU, more instruction level information is obtainable to the memory controller. Access reordering mechanisms can take advantages of this additional information. For example, with current burst scheduling, the read accesses of a burst are scheduled in the same order as they are issued. If an access can be identified as more critical that others, i.e. the access has more dependent instructions, then it can be promoted inside the burst to reduce the latency. Similarly the sequence of bursts within banks can also be reordered to reduce latency to critical data. Bursts within the same bank could be sorted by some mechanisms other that the arrival time of the first access of each burst. For example, the number of critical accesses of each burst could

be used to sort the bursts. However, considerations need to be taken into account when performing such inter burst reordering to prevent starvation.

One of the previous studies shows that thread-aware SDRAM access scheduling schemes may improve the overall system performance by up to 30% on memory-intensive applications [78]. Thread level information, such as the number of outstanding memory accesses of each thread, reorder buffer occupancy and the issue queue occupancy, could also be considered when making access scheduling decisions.

6.4.2 Dynamic SDRAM Controller Policy

The study of SDRAM controller policy, as presented in Section 4.4.1, shows that a dynamic controller policy can outperform static controller policies. To make the decision of whether or not to leave the accessed row open, a dynamic controller policy requires the future information of the next access to the same bank. There are two possible ways to obtain the future access information.

With an out-of-order execution superscalar processor, the memory controller is likely to have multiple outstanding memory accesses waiting in the access queue. Therefore, the controller can search the access queue to obtain the information of future accesses when it is the time to make the decision. With burst scheduling, the boundaries of bursts give an indication of when it is appropriate to close the open row and precharge the bank. During a burst, the accessed row should always be left open; when a burst completes, most likely the first access of the next burst will be directed to a different row. Therefore with burst scheduling the memory controller should only perform autoprecharges at the end of bursts.

If the next access is not available at the time to make the decision, i.e. when a burst completes, there are no more accesses/bursts to this bank in the access queue. Then a controller policy predictor could be used to make a prediction. Proposed by Xu, a dynamic controller policy predictor uses history information to predict whether the next access to

the bank would be a row hit or a row conflict [74]. The design and implementation of such dynamic controller policy predictor are undergoing.

6.4.3 Intelligent Data Placement through Software

The proposed SDRAM address mapping and access reordering mechanisms use hardware approaches to reduce main memory access latency. Compiler and operating system also have significant impacts on the performance of memory subsystem. As discussed in Section 4.4.2, the operating system has impacts on access distribution in memory space through virtual paging system. With the knowledge of main memory organization, the operating system could intelligently place virtual pages in the memory space to exploit bank parallelism and increase row locality. Theoretically virtual paging system could be as effective as SDRAM address mapping techniques in evenly distributing accesses to all banks. In contrast to SDRAM address mapping techniques which are static, virtual paging system is capable of dynamically changing data placement to match program behavior changes during runtime.

6.5 Afterward

Four years of research work on memory optimization techniques have demonstrated that system performance can be significantly improved through enabling bank parallelism and exploiting locality in main memory access streams, which have not been fully exploited by conventional memory controllers.

As shown in Table 1.1, the SDRAM device timing keeps increasing in terms of cycles as SDRAM devices and CPUs evolve, resulting in even longer main memory access latency in CPU cycles. Therefore, the memory optimization techniques that have been studied herein, including but not limited to SDRAM address mapping techniques and access reordering mechanisms, will provide even more significant performance improvement in the future than are given by the simulations presented in this thesis.

Appendix A

Dual SDRAM Channels

Prior to 2003, most memory controllers were located on the north bridge and had a single 64-bit data channel configuration. As the speed gap between the processor and the main memory increases, the main memory becomes a bottleneck of system as introduced in Chapter 1. Dual channel technology was first introduced by Intel to address the issue of the bottleneck [68]. A dual channel memory controller utilizes two 64-bit data channels, creating a logical 128-bit data channel and doubling the amount of available memory bandwidth. With two channels working simultaneously, the bottleneck effect is alleviated. As of 2006, modern processors such as AMD Athlon 64 and Opteron [5], IBM Power5 [62] have the memory controller integrated on the CPU die to reduce the memory latency.

A.1 Asymmetric and Symmetric Dual Channel

Dual channel-enabled chipsets, such as Intel 925X chipset [27], can have two modes, asymmetric dual channel and symmetric dual channel.

With *asymmetric dual channel*, two channels work independently. Each channel can serve one memory access at the same. With *symmetric dual channel*, two channels are clustered to create a logical channel doubling the width of the data bus. In order to use

symmetric dual channel mode, memory modules installed in each channel must be identical in terms of size, specification and device organization. Each memory access is served by both channels, e.g. the first half of memory block is provided by *channel*0 while *channel*1 provides the second half. How memory blocks are strode on two channels depends on the actual implementation.

Asymmetric dual channel provides greater parallelism than symmetric dual channel. When *channel*0 is serving an access, another access can be issued to *channel*1 and get served immediately, reducing access latency. Symmetric dual channel, on the other hand, provides better data throughput. Due to the doubled bandwidth, it takes half time to transfer requested memory blocks, yielding a high data throughput, although the latency of memory accesses is not directly affected.

A.2 Performance of Dual Channel

Performance of single channel, asymmetric dual channel and symmetric dual channel are compared. Table A.1 lists the major differences between these three configurations. Other configurations are identical to the M5 baseline machines as shown in Table 3.2. Symmetric dual channel is simulated using a 128-bit single channel. Note that dual channel modes have twice amount of memory than single channel. Bit-reversal address mapping with depth of 14 is used in order to create balanced loads to each channel.

Figure A.1 shows the execution time of asymmetric dual channel and symmetric dual channel. Execution times of tow dual channel configurations are normalized to single channel in order to illustrate the effect of channel configuration.

Generally a dual channel configuration results in a better performance than a single channel configuration. The only exception is `gcc` with which asymmetric dual channel increases the execution time by 31% over single channel. Asymmetric dual channel reduces the execution time by 6% crossing all benchmarks compared to single channel. Symmetric

Table A.1: Configuration of single channel and two modes of dual channels

	SingleChan	AsymDualChan	SymDualChan
SDRAM device	PC2-6400 DDR2 SDRAM		
Channel number	1	2	1
Data bus width	64-bit	64-bit	128-bit
Burst length	8	8	4
Total bank number	16	32	16
Total memory size	2GB	4GB	4GB
Total memory bandwidth	6.4GB	12.8GB	12.8GB
SDRAM Address mapping	Bit-reversal with depth 14		
Access reordering	Burst scheduling with threshold 52		

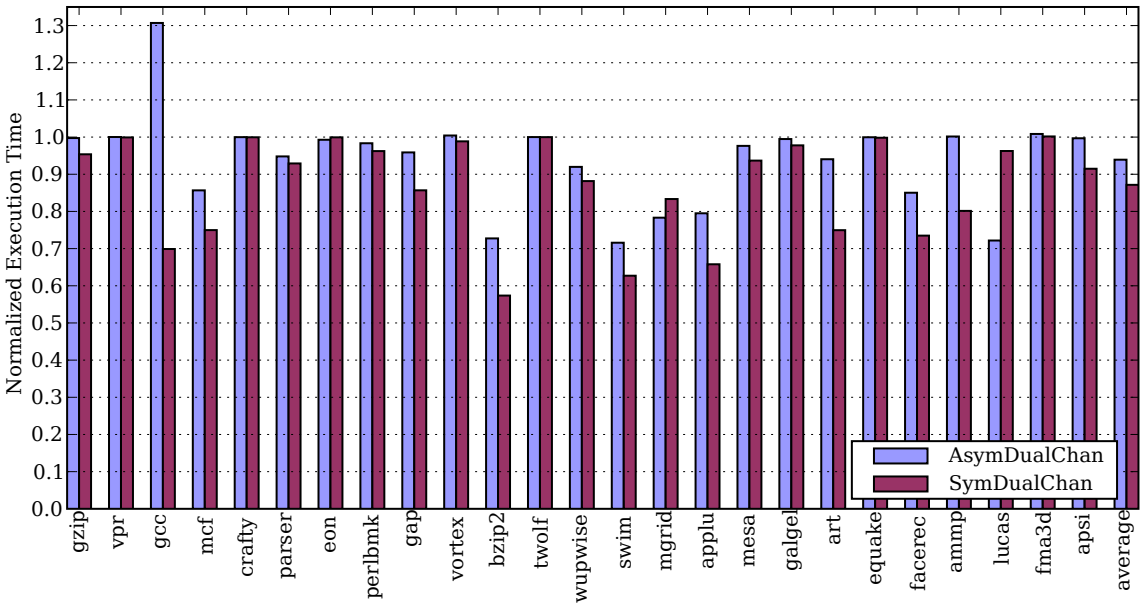

Figure A.1: Execution time of asymmetric dual channel and symmetric dual channel (normalized to single channel)

dual channel outperforms asymmetric dual channel, reducing an average of 13% of execution time over single channel. While symmetric dual channel has better performance than asymmetric dual channel on average, on some benchmarks asymmetric dual channel does perform better than symmetric dual channel, such as `mgrid` and `lucas`.

A previous study by Zhichun Zhu and et al. shows that organizing each channel as an independent one may outperform combining them as a single logic channel by up to 90% on 8-channel DDR SDRAM systems [78].

Multiple SDRAM channels certainly alleviate the bottleneck effect cased by the main memory bandwidth. Channel configuration is determined by the way that channel index is mapped from physical address bits during SDRAM address mapping as discussed in Section 2.6. System performance is largely dependent upon channel organization as well as program's memory access pattern.

Appendix B

SDRAM Power Consumption

Memory power consumption becomes more and more concerned in system design, especially for some embedded systems. DDR2 SDRAM devices provides high data bandwidth with lower system power than DDR. However, it is not always easy to determine the power required by the memory device from a data sheet alone.

There are technical notes providing tools and techniques to estimate memory power consumption in a given system [3, 4]. A memory power consumption module for DDR/DDR2 device is implemented and integrated into the SDRAM module v2.0 based on these technical notes.

B.1 SDRAM Power Components

SDRAM power is composed of background power, activate power, read/write power, I/O and termination power and refresh power. As a preliminary study of SDRAM power consumption, power derating, voltage supply scaling and frequency scaling are not considered. Also SDRAM devices are never put into power-down mode.

B.1.1 Background Power

During normal operation, SDRAM device always consumes one of four background powers depending on whether all banks are precharged or one or more banks are activated, as well as whether the device is in power-down mode or standby mode, as summarized in Table B.1.

Table B.1: SDRAM background powers

$P(PRE\_PDN)$	In power-down mode with all banks precharged
$P(PRE\_STBY)$	In standby mode with all banks precharged
$P(ACT\_PDN)$	In power-down mode with one or more banks activated
$P(ACT\_STBY)$	In power standby mode with one or more banks activated

Background powers can be calculated by multiplying the I_{DD} values by the voltage applied to the device V_{DD}, as shown in following equations. Where $I_{DD}2P/I_{DD}2N$ is precharge power-down/standby current, $I_{DD}3P/I_{DD}3N$ is active power-down/standby current, which can be found in device data sheets.

$$P(PRE\_PDN) = I_{DD}2P \times V_{DD} \quad \text{(B.1)}$$

$$P(PRE\_STBY) = I_{DD}2N \times V_{DD} \quad \text{(B.2)}$$

$$P(ACT\_PDN) = I_{DD}3P \times V_{DD} \quad \text{(B.3)}$$

$$P(ACT\_STBY) = I_{DD}3N \times V_{DD} \quad \text{(B.4)}$$

Due to the assumption that all SDRAM devices are always in standby mode, the background power of an SDRAM device is composed of precharged standby power $P_{sys}(PRE\_STBY)$ and activated standby power $P_{sys}(ACT\_STBY)$, which can be calculated using Equation B.5 and Equation B.6 respectively, where $BNK\_PRE\%$ is the percent of time that all banks of the SDRAM device are precharged.

$$P_{sys}(PRE\_STBY) = P(PRE\_STBY) \times BNK\_PRE\%$$

$$= I_{DD}2N \times BNK\_PRE\% \times V_{DD} \quad \text{(B.5)}$$

$$P_{sys}(ACT\_STBY) = P(ACT\_STBY) \times (1 - BNK\_PRE\%)$$

$$= [I_{DD}3N \times (1 - BNK\_PRE\%)] \times V_{DD} \quad \text{(B.6)}$$

B.1.2 Active Power

Row activates and bank precharges always appear in pairs, although there may be one or more column accesses between them. A row activate decodes the row index, activates the specified row and transfers the data from the SDRAM array to the sense amplifiers. Significant amount of current is used during a row activate. A bank precharge restores the data from the sense amplifiers into the SDRAM array and prepares the bank for the next row activate. Active power which is consumed by row activate and bank precharge pairs can be calculated by Equation B.7.

$$P(ACT) = [I_{DD}0 - \frac{I_{DD}3N \times t_{RAS} + I_{DD}2N \times (t_{RC} - t_{RAS})}{t_{RC}}] \times V_{DD} \quad \text{(B.7)}$$

Where t_{RAS} is the minimal interval between a row activate and the following bank precharge, t_{RC} is the minimal interval between two adjacent row activates. $I_{DD}0$ is the current operating one row activate and bank precharge. Background current $I_{DD}3N$ and $I_{DD}2N$ need to be subtracted from $I_{DD}0$ to identify the power consumed by row activate and bank precharge.

Equation B.7 is correct only when there is no contention on the SDRAM buses and the row activates are scheduled at the minimum t_{RC} specified in the data sheet. In real systems, however, row activates could be scheduled closer than t_{RC} when multiple banks are interleaving, or greater than t_{RC} in order to meet other timing constraints. A scaling factor t_{RRDsch} is introduced, which is the average scheduled row activates interval. $P(ACT)$ can be easily scaled by a ratio of the actual t_{RRDsch} to the data sheet value t_{RC} to represent

the actual active power, as shown in Equation B.8.

$$\begin{aligned} P_{sys}(ACT) &= P(ACT) \times \frac{t_{RC}}{t_{RRDsch}} \\ &= [I_{DD}0 - \frac{I_{DD}3N \times t_{RAS} + I_{DD}2N \times (t_{RC} - t_{RAS})}{t_{RC}}] \times V_{DD} \times \frac{t_{RC}}{t_{RRDsch}} \\ &= \frac{I_{DD}0 \times t_{RC} - I_{DD}3N \times t_{RAS} - I_{DD}2N \times (t_{RC} - t_{RAS})}{t_{RRDsch}} \times V_{DD} \end{aligned} \quad \text{(B.8)}$$

B.1.3 Read/Write Power

When a bank is activated, one or more column accesses (reads or writes) can access the row data. Read/write power is the power for transferring data on the data bus. The associated read/write current is $I_{DD}4R/I_{DD}4W$. When calculating read/write power, background currents need to be subtracted because they have already been counted in background powers. Read/write power can be calculated using Equation B.9 and Equation B.10. Where, *read_cycles* and *write_cycles* are the actual number of read or write cycles that are used to transfer data on the data bus.

$$P_{sys}(RD) = (I_{DD}4R - I_{DD}3N) \times \frac{read\_cycles}{total\_cycles} \times V_{DD} \quad \text{(B.9)}$$

$$P_{sys}(WR) = (I_{DD}4W - I_{DD}3N) \times \frac{write\_cycles}{total\_cycles} \times V_{DD} \quad \text{(B.10)}$$

B.1.4 I/O and Termination Power

Besides $P_{sys}(RD)$ and $P_{sys}(WR)$, I/O power and termination power also contribute to the total power for column accesses. The actual I/O power and termination power vary depending on system configuration and need to be calculated individually. Detailed discussion about I/O power and termination power can be found in Micron technical note "Calculating Memory System Power For DDR2" [4]. Table B.2 summarizes the typical I/O and termination power consumption for a 2-DIMM configuration DDR2 system [4], which

Table B.2: Typical I/O and Termination Power Consumption

$P(RD)$	1.5mW	Output driver power when driving the bus
$P(WR)$	0mW	Power when terminating a write to the SDRAM
$P(RDoth)$	13.1mW	Power when terminating a read from another SDRAM
$P(WRoth)$	14.6mW	Power when terminating write data to another SDRAM

are used in the simulations.

Equation B.11 and Equation B.12 calculate the actual I/O power and termination power. Where num_DQR and num_DQW are the number of signals for reads or writes. For example, given a x8 device with differential strobe enables (DQS/DQS#), num_DQR include 8 DQ and 2 DQS signals for a total of 10, whereas num_DQW totals 11 including the addition data mask. $P(RD)$ and $P(WR)$ are only consumed during data transfer on the data bus therefore they need to be scaled by actual data bus utilization. Two additional terms, $termRDsch\%$ (rate of terminating read data from another SDRAM) and $termWRsch\%$ (rate of terminating write data to another SDRAM), are required to cover termination cases for data to/from another SDRAM. To simplify the calculation, fixed values of 15% for $termRDsch\%$ and 5% for $termWRsch\%$ are used in the simulations.

$$P_{sys}(DQ) = P(RD) \times num\_DQR \times \frac{read\_cycles}{total\_cycles} \tag{B.11}$$

$$\begin{aligned} P_{sys}(TERM) = {} & P(WR) \times num\_DQW \times \frac{write\_cycles}{total\_cycles} + \\ & P(RDoth) \times num\_DQR \times termRDsch\% + \\ & P(WRoth) \times num\_DQW \times termWRsch\% \end{aligned} \tag{B.12}$$

B.1.5 Refresh Power

SDRAM devices need to be periodically refreshed to retina data integrity. Current $I_{DD}5$ is consumed if an SDRAM device is being refreshed at minimum refresh-to-refresh interval, t_{RFC}. Refresh operations are normally distributed evenly over time at a refresh interval

of t_{REFI}, which is equal to the refresh period (64ms typically) divided by the number of SDRAM rows [30]. Background current $I_{DD}3N$ is deducted from $I_{DD}5$ to calculate only the power due to refresh. Refresh power can be calculated using Equation B.13.

$$P_{sys}(REF) = (I_{DD}5 - I_{DD}3N) \times \frac{t_{RFC}}{t_{REFI}} \times V_{DD} \tag{B.13}$$

B.2 Total SDRAM Power

Total SDRAM power consumption is the sum of all power components, as shown in Equation B.14. Note that Equation B.14 is for a single SDRAM device. To calculate the total power of a memory system that consists of multiple channels, multiple ranks per channel and multiple SDRAM devices per rank, the result needs to be multiplied by the total number of SDRAM devices in the system.

$$\begin{aligned} P_{sys}(TOT) &= P_{sys}(PRE\_STBY) + P_{sys}(ACT\_STBY) + P_{sys}(ACT) + P_{sys}(RD) + \\ &\quad P_{sys}(WR) + P_{sys}(DQ) + P_{sys}(TERM) + P_{sys}(REF) \end{aligned} \tag{B.14}$$

Figure B.1 shows the memory power consumption breakout. The simulated machine has a dual symmetrical channel configuration as shown in Table A.1. Each channel contains two ranks of PC2-6400 DDR2 SDRAM device.

On average main memory requires 13.3W of power, excluding the power required by the memory controller. 53% of total power is consumed by background power, which is composed by activated standby power 3.5W (26%) and precharged standby power 3.6W (27%). The next largest power component is active power, required by row activate and bank precharge, consuming 3.2W power (24%). Due to the fixed values of $termRDsch\%$ and $termWRsch\%$ used in the simulations, termination power is identical on all benchmarks and the value (1.7W) is only for reference. Read/Write and DQ power together consume

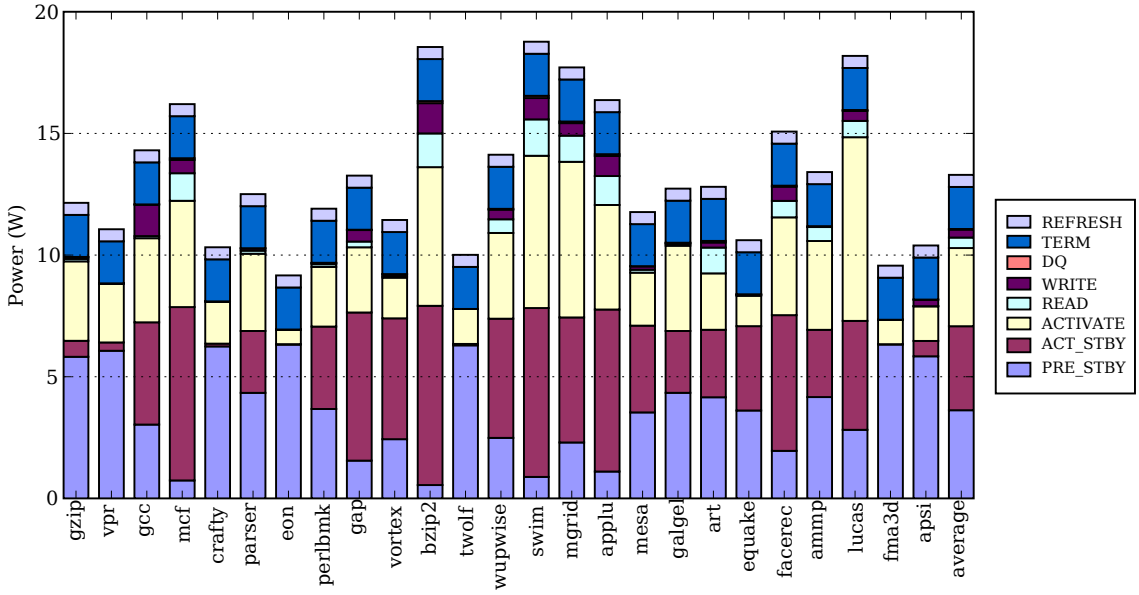

Figure B.1: Main memory power consumption

0.7W power (5%) and refresh requires 0.5W power (4%).

Based on Figure B.1, more than half of memory power is consumed by background power. This is partially because that memory devices are never put into power-down mode in the simulation. With power-down mode, memory devices consume significantly less power. For an instance, simulated DD2 device consumes 7mA current in power-down mode with all bank precharged vs. 55mA current in standby mode. The SDRAM controller could use power-down mode to save power when there are no outstanding accesses in main memory. However, it takes time (specified by t_{XARD} or t_{XARDS}) for an SDRAM device to exit power-down mode [30]. Memory access will experience a longer latency if the device is in power-down mode when the access arrives. Therefore, trade off must be made between performance and power saving.

B.3 Power Consumption vs. Energy Consumption

To illustrate the effects of memory optimization techniques on memory power consumption as well energy consumption, the average power consumption and energy consumption of three channel configurations as discussed in Appendix A are shown in Figure B.2 and Figure B.3 respectively.

Switching from single channel to dual channels generally doubles the memory power consumption. Power consumption of symmetric dual channel (13.3W) is slightly higher than that of asymmetric dual channel (12.9W), as shown in Figure B.2.

According to Figure A.1 symmetric dual channel and asymmetric dual channel can reduce the execution time by 13% and 6% respectively. Because energy is a product of power and time, symmetric dual channel actually consumes less energy (6.8J) than asymmetric dual channel (8J), as confirmed by Figure B.3.

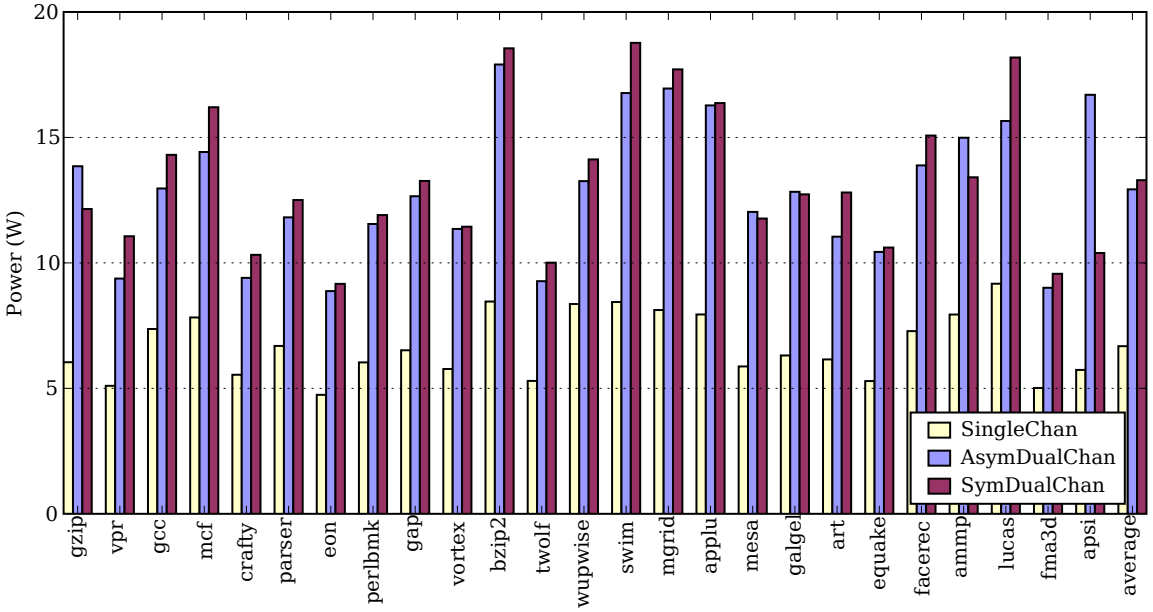

Figure B.2: Power consumption of various SDRAM channel configurations

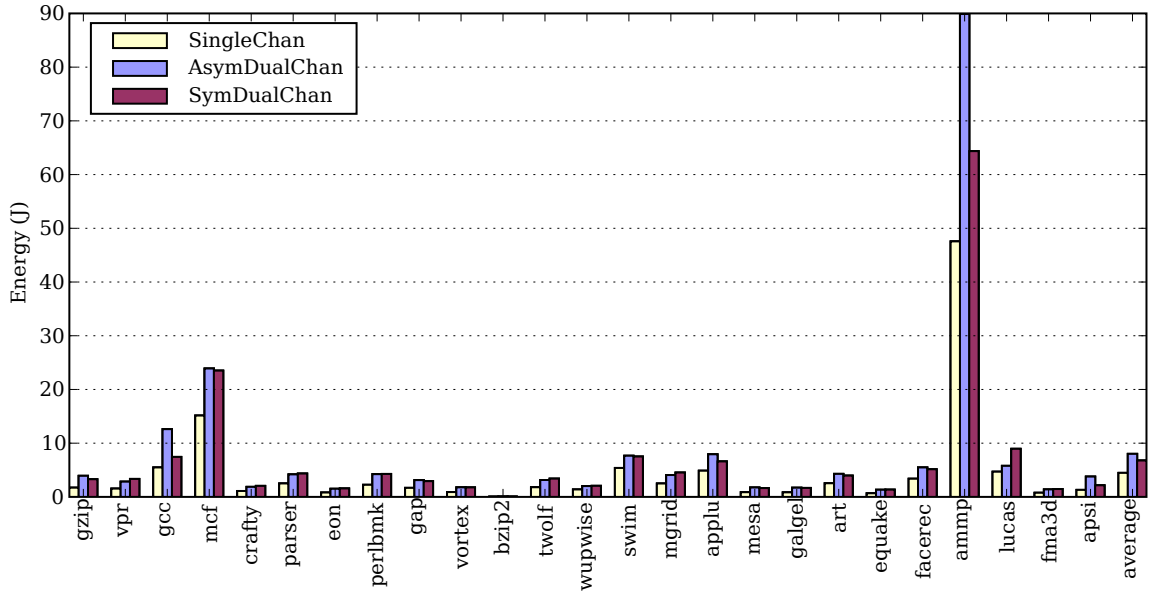

Figure B.3: Energy consumption of various SDRAM channel configurations

Bibliography

[1] SimOS The Complete Machine Simulator. http://simos.stanford.edu.

[2] The M5 Simulator System. http://m5.eecs.umich.edu.

[3] Calculating Memory System Power for DDR. Technical Report TN-46-03, Micron Technology, Inc., 2003.

[4] Calculating Memory System Power for DDR2. Technical Report TN-47-04, Micron Technology, Inc., 2004.

[5] Advanced Micro Devices. *AMD Athlon 64 Processor Product Data Sheet*, September 2006.

[6] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, July 2006.

[7] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-Oriented Full-System Simulation using M5. In *Proceedings of the Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, 2003.

[8] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[9] Dave Bursky. Fast DRAMs Can Be Swapped for SRAM Caches. Electronic Design, pages 55-56, 60-67, July 1993.

[10] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 70, Washington, DC, USA, 1999. IEEE Computer Society.

[11] Chris Weaver. *Pre-compiled little-endian Alpha ISA SPEC2000 binaries.*

[12] Daniel Citron. MisSPECulation: Partial and Misleading Use of SPEC CPU2000 in Computer Architecture conferences. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 52–61, New York, NY, USA, 2003. ACM Press.

[13] Richard Crisp. Direct Rambus Technology: The New Main Memory Standard. *IEEE Micro*, 17(6):18–28, 1997.

[14] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[15] Vinodh Cuppu and Bruce Jacob. Concurrency, latency, or system overhead: which has the largest impact on uniprocessor DRAM-system performance? In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 62–71, New York, NY, USA, 2001. ACM Press.

[16] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. High-Performance DRAMs in Workstation Environments. *IEEE Trans. Comput.*, 50(11):1133–1153, 2001.

[17] Vinodh Cuppu, Bruce L. Jacob, Brian Davis, and Trevor N. Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *ISCA '93: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 222–233, 1999.

[18] William J. Dally and John W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[19] Brian T. Davis. *Modern DRAM Architectures*. PhD thesis, Dept. of Computer Science and Engineering, the University of Michigan, 2001.

[20] Robert Dennard. Field-Effect Transistor Memory. US Patent 3387286, 1968.

[21] Wi fen Lin. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 301, Washington, DC, USA, 2001. IEEE Computer Society.

[22] Jahangir Hasan, Satish Chandra, and T. N. Vijaykumar. Efficient Use of Memory Bandwidth to Improve Network Processor Throughput. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 300–313, New York, NY, USA, 2003. ACM Press.

[23] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.

[24] Sung I. Hong, Sally A. McKee, Maximo H. Salinas, Robert H. Klenke, James H. Aylor, and Wm. A. Wulf. Access Order and Effective Bandwidth for Streams on a Direct Rambus Memory. 9(13):80–89, January 1999.

[25] Ibrahim Hur and Calvin Lin. Adaptive History-Based Memory Schedulers. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 343–354, Washington, DC, USA, 2004. IEEE Computer Society.

[26] IBM. IBM Power Architecture. http://www.ibm.com/chips/power/.

[27] Intel Corporation. *Intel 925X and 925XE Express Chipset Datasheet*, November 2004.

[28] Intel Corporation. *Intel 965 Express Chipset Family Datasheet*, July 2006.

[29] IPRS. International Technology Roadmap for Semiconductors, 2003.

[30] Jeff Janzen. DDR2 Offers New Features and Functionality. DesignLine, 12(2), Micron Technology, Inc., 2003.

[31] JEDEC. Joint Electronic Device Engineering Council. http://www.jedec.org/.

[32] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*.

[33] Jungeun Kim and Taewhan Kim. Memory Access Optimization Through Combined Code Scheduling, Memory Allocation, and Array Binding in Embedded System Design. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 105–110, New York, NY, USA, 2005. ACM Press.

[34] Jochen Liedtke, Marcus Völp, and Kevin Elphinstone. Preliminary Thoughts on Memory-bus Scheduling. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 207–210, New York, NY, USA, 2000. ACM Press.

[35] SimpleScalar LLC. SimpleScalar 3.0d. http://www.simplescalar.com/.

[36] Chun-Gi Lyuh and Taewhan Kim. Memory Access Scheduling and Binding Considering Energy Minimization in Multi-bank Memory Systems. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 81–86, New York, NY, USA, 2004. ACM Press.

[37] Mark Malek. Compiler Optimized Memory Page Placement for Reducing DRAM Latency. Master's thesis, Dept. of Computer Science, Michigan Technological University, November 2005.

[38] Jack A. Mandelman, Robert H. Dennard, Gary B. Bronner, John K. DeBrosse, Rama Divakaruni, Yujun Li, and Carl J. Raden. Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM). *IBM Journal of Research and Development*, 46(2-3):187–222, 2002.

[39] Binu K. Mathew, Sally A. McKee, John B. Carter, and Al Davis. Design of a Parallel Vector Access Unit for SDRAM Memory Systems. In *HPCA '00: Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 39–48, 2000.

[40] Scott McFarling. Combining Branch Predictors. Technical Report TN-36, June 1993.

[41] Sally A. McKee, William A. Wulf, James H. Aylor, Maximo H. Salinas, Robert H. Klenke, Sung I. Hong, and Dee A. B. Weikle. Dynamic Access Ordering for Streamed Computations. *IEEE Trans. Comput.*, 49(11):1255–1271, 2000.

[42] Micron Technology, Inc. *Micron 512Mb: x4, x8, x16 DDR SDRAM Datasheet*, 2005.

[43] Micron Technology, Inc. *Micron 512Mb: x4, x8, x16 DDR2 SDRAM Datasheet*, 2006.

[44] Sun Microsystems. UltraSPARC T1 Processor. http://www.sun.com/processors/-UltraSPARC-T1/.

[45] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[46] Steven A. Moyer. Access ordering and effective memory bandwidth. Technical Report CS-93-18, Computer Science Department of University of Virginia, April 1993.

[47] NEC. *64M-bit Virtual Channel SDRAM Datasheet*, October 1998.

[48] Ray Ng. Fast Computer Memories. IEEE Spectrum, pages 36-39, October 1992.

[49] NVIDIA. NVIDIA nForce4 SLI Media and Communications Processors Intel Edition. http://www.nvidia.com/page/nforce4_sli.html.

[50] David A. Patterson. Latency Lags Bandwidth. *Communications of the ACM*, 47(10):71–75, 2004.

[51] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[52] Charles Price. *MIPS IV Instruction Set, Revision 3.2*. MIPS Technologies, Inc., September, 1995.

[53] Betty Prince. *High Performance Memories: New Architecture DRAMs and SRAMs - Evolution and Function*. John Wiley & Sons, 1999.

[54] Scott Rixner. Memory Controller Optimizations for Web Servers. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 355–366, Washington, DC, USA, 2004. IEEE Computer Society.

[55] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A Bandwidth-efficient Architecture for Media Processing. In *MICRO 31: Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 3–13, 1998.

[56] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, New York, NY, USA, 2000. ACM Press.

[57] Hemant G. Rotithor, Randy B. Osborne, and Nagi Aboulenein. Method and Apparatus for Out of Order Memory Scheduling. Patent US 2005/0091460 A1, Intel Corporation, April 2005.

[58] Jun Shao and Brian T. Davis. The Bit-reversal SDRAM Address Mapping. In *SCOPES '05: Proceedings of the 9th International Workshop on Software and Compilers for Embedded Systems*, pages 62–71, September 2005.

[59] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Professional, 2004.

[60] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, 2004.

[61] Deszö Sima, Terence Fountain, and Péter Kacsuk. *Advanced Computer Architectures: A Design Space Approach*. Addison-Wesley, 1997.

[62] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 System Microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.

[63] Kevin Skadron and Douglas W. Clark. Design Issues and Tradeoffs for Write Buffers. In *HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, page 144, Washington, DC, USA, 1997. IEEE Computer Society.

[64] Standard Performance Evaluation Corporation. *SPEC CPU95 Benchmark Suites* , August 1995.

[65] Standard Performance Evaluation Corporation. *SPEC CPU2000 V1.2*, December 2001.

[66] V. Stankovic and N. Milenkovic. Access Latency Reduction in Contemporary DRAM Memories. *Facta Univ. Ser.: Elec. Energ.*, 17(1), April 2004.

[67] K. U. Stein, A. Sibling, and E. Doering. Storage Array and Sense/Refresh Circuits for Single-Transistor Memory Cells. In *IEEE J. Solid-State Circuits*, pages 336–340. IEEE Computer Society, 1972.

[68] Infineon Technologies and Kingston Technology. Intel Dual-Channel DDR Memory Architecture White Paper, September 2003.

[69] Rokicki Tomas. Indexing Memory Banks to Maximize Page Mode Hit Percentage and Minimize Memory Latency. Technical Report HPL-96-95, Hewlett-Packard Laboratories, June 1996.

[70] VIA Technologies, Inc. *VIA KT880 North Bridge Data Sheet*, September 2004.

[71] Adrian Wong. *Breaking Through the BIOS Barrier: The Definitive BIOS Optimization Guide for PCs*. Prentice Hall, 2004.

[72] Wayne A. Wong and Jean-Loup Baer. Dram caching. Technical Report UW-CSE-97-03-04, Department of Computer Science and Engineering of University of Washington, February 1997.

[73] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[74] Ying Xu. Dynamic SDRAM Controller Policy Predictor. Master's thesis, Dept. of Electrical and Computer Engineering, Michigan Technological University, April 2006.

[75] Tse-Yu Yeh and Yale N. Patt. A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, 1993.

[76] Lixin Zhang, Zhen Fang, Mide Parker, Binu K. Mathew, Lambert Schaelicke, John B. Carter, Wilson C. Hsieh, and Sally A. McKee. The Impulse Memory Controller. *IEEE Trans. Comput.*, 50(11):1117–1132, 2001.

[77] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 32–41, New York, NY, USA, 2000. ACM Press.

[78] Zhichun Zhu and Zhao Zhang. A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 213–224, Washington, DC, USA, 2005. IEEE Computer Society.

[79] Zhichun Zhu, Zhao Zhang, and Xiaodong Zhang. Fine-grain Priority Scheduling on Multi-channel Memory Systems. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 107, Washington, DC, USA, 2002. IEEE Computer Society.