Design A DRAM Backend for The Impulse Memory System

Lixin Zhang
E-mail: lizhang@cs.utah.edu

UUCS-00-002

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

January, 2000

Abstract

The Impulse Adaptable Memory System is a new memory system that exposes DRAM access patterns not seen in conventional memory systems. Impulse can generate huge number of small DRAM accesses, which will not be handled effectively by a conventional cache-line-size-access-oriented DRAM backend. In this paper, we describe and evaluate an Impulse DRAM backend design that exploits the potential parallelism of the DRAM accesses in an Impulse system and reduces the average DRAM access latency using the latest DRAM technologies such as hot row. We also study the effects of several important factors in the DRAM backend: interleaving of DRAM banks, dynamic reordering of DRAM accesses, hot row policy, and DRAM organization. The experimental results of five representative benchmarks running on the execution-driven simulator Paint [11] show that different DRAM backend configurations can yield huge impacts, saving up to 98% on average DRAM access latency, 90% on memory cycles, and 80% on execution time.

Contents

1 Introduction **3**

2 Overview of The Impulse Memory System **3**
2.1 Hardware . . . 5
2.2 Remapping Algorithms . . . 6

3 DRAM Basics **7**
3.1 **Synchronous DRAM** . . . 7
3.2 **Direct Rambus DRAM** . . . 7

4 DRAM Backend for The Impulse Memory System **9**
4.1 DRAM Dispatcher . . . 10
4.2 Slave Memory Controller and DRAM Chip . . . 11
4.3 Algorithms in the DRAM Backend . . . 12
4.3.1 Hot Row Policy . . . 12
4.3.2 Bank Waiting Queue Reordering . . . 13
4.3.3 Memory Bank Interleaving . . . 14

5 Experimental Framework **14**
5.1 Simulation Environment . . . 14
5.2 Benchmarks . . . 15
5.3 Methodology . . . 16

6 Performance **16**
6.1 The Impact of DRAM Organization . . . 17
6.2 The Impact of Slave Busses . . . 19
6.3 The Effect of Hot-row Policy . . . 21
6.4 The Effect of Bank-queue Reordering Algorithm . . . 23
6.5 The Effect of Interleaving Schemes . . . 25
6.6 Putting It All Together . . . 27

7 Conclusion and Future Work **28**

1 Introduction

Impulse is a new memory system that adds two important features to a traditional memory system. First, Impulse supports application-specific optimizations through configurable physical address remapping. By remapping physical addresses at the memory controller, applications control how their data are accessed and cached, thereby improving cache performance and bus utilization. Second, Impulse supports prefetching at the memory controller, which can hide majority of the memory latency. The Impulse memory system exhibits special DRAM access patterns different with what a conventional memory system exhibits. For example, the Impulse memory controller may gather a 128-byte cache line by generating 16 eight-byte DRAM accesses directed to 16 different memory locations. Since a DRAM access in a conventional system always fetches a cache line, the conventional DRAM backend is specifically designed to handle cache-line-size access effectively and may not work well with smaller DRAM accesses. So it is necessary to evaluate the necessity of redesigning the conventional DRAM backend for Impulse.

In order to handle the huge number of small DRAM accesses in the Impulse memory system, the DRAM backend for Impulse must be able to exploit the inherent parallelism of those small DRAM accesses. The most important factor of the DRAM backend is the DRAM organization which includes how the DRAM banks are connected together, how the DRAM backend communicates with the memory controller, and how the functionality like access scheduling, bank interleaving, and DRAM refreshing, is distributed inside the DRAM backend. Another important factor of a DRAM backend is the access scheduling which reorders the DRAM accesses in order to achieve the maximum parallelism as closely as possible, therefore reaching the ultimate goal – reducing the memory latency perceived by the processor. Hot row policy is one important factor too. Nowadays' DRAM technology allows quick reference to a hot row. Keeping the hot row open whenever it's needed and closing it whenever it's not needed may reduce the average DRAM access latency dramatically [7]. Hot row policy predicts whether or not the next access is going to hit a hot row and then sends a command to the DRAM bank accordingly. Another important factor is the interleaving of memory banks. How to number memory banks and under which level they are interleaved may affect the performance dramatically because it directly affects the potential parallelism that a sequence of DRAM accesses has. For example, assuming we have two banks and a sequence of two accesses (0, 128), these two accesses can be served in parallel if the banks are interleaved in cache-line-level, but they can only be served serially if the banks are interleaved in page-level. In the rest of this paper, we are going to study all these important factors described in this paragraph.

The remainder of this paper is organized as follows. Section 2 briefly overviews the basic architecture of the Impulse memory system, focusing on the master memory controller. Section 3 provides basic knowledge about DRAM and briefly studies two common types of DRAMs: Synchronous DRAM and Direct Rambus DRAM. Section 4 illustrates a DRAM backend design. Section 5 describes the simulation environment and the benchmarks used in our experiments. The performance results and analyses are reported in Section 6. Section 7 discusses future work and concludes this paper.

2 Overview of The Impulse Memory System

The most distinguishable feature of Impulse is the addition of another level of address translation at the memory controller. The key insight exploited by this feature is that "unused" physical addresses can undergo a translation to "real" physical addresses at the memory controller. For example, in a

conventional system with 4 gigabytes of physical address space (32-bit physical address) and only 1 gigabytes of installed DRAMs, there is 3 gigabytes of "unused" physical address space which is not directly backed up by DRAMs and will generate errors if presented to a conventional memory controller. The "unused" physical addresses are called *shadow addresses*, and they constitute a *shadow address space.* In an Impulse system, applications can reorganize their data structures in shadow address space to explicitly control how their data are accessed and cached. When the Impulse memory controller receives a shadow address, it will translate the shadow address into "real" physical addresses (a.k.a physical DRAM addresses) instead of generating an error as a conventional memory controller does. In current Impulse design, the mapping from shadow address space to real physical address space can be in any granularity as long as it is a power of two and is less than or equal to a page size.

Data items whose virtual addresses are not contiguous can be mapped to contiguous shadow addresses, so that sparse data in virtual address space can be compacted into dense cache lines in shadow address space before being transferred to the processor. To map elements in these compacted cache lines back to physical memory, Impulse must recover their offsets within the virtual layout of original data structure. To do this, Impulse first translates *shadow addresses* to *pseudo-virtual addresses* which mirrors the virtual address space in its page layout, and then translates these *pseudo-virtual addresses* to physical DRAM addresses. The *shadow* $\rightarrow$ *pseudo-virtual* $\rightarrow$ *physical* mappings all take place within the Impulse memory controller. The *shadow* $\rightarrow$ *pseudo-virtual* mapping involves some simple arithmetic operations and is implemented by ALU units. The *pseudo-virtual* $\rightarrow$ *physical* mapping involves page table lookups and is implemented by a small *table lookaside buffer* (TLB) at the memory controller.

The second important feature of Impulse is that it supports prefetching — Memory-Controller-based prefetching (MC-based prefetching). A small amount of SRAM – so-called Memory Controller cache or MCache – is integrated at the memory controller to store data prefetched from the DRAMs. The MC-based prefetching uses a very simple scheme: sequential prefetching for non-remapped data; configurable-stride prefetching for remapped data. All prefetchings are one cache line ahead only. A prefetching is issued at two situations: when an access misses on the MCache, fetch the requested cache line and prefetch the next one; when an access hits a prefetched cache line, prefetch the next one. For non-remapped data, prefetching is useful for reducing the memory latency of sequentially-accessed data. For remapped data, prefetching enables the controller to hide the cost of remapping: some remapping can require multiple DRAM accesses to fill a single cache line.

The shadow address space is managed by the operating system in a way similar to real physical address space. The operating system guarantees any remapped data's shadow address space image to be contiguous even it spans multiple pages. This guarantee not only simplifies the translation at the memory controller, but also allows the CPU to use superpage TLB entries to translate remapped data, thereby improving the CPU TLB performance. The operating system provides an interface for applications to specify optimizations for their particular data structures and configures the Impulse memory controller to reinterpret the shadow addresses presented to it. The programmer (or the compiler, in the future) inserts directives into the applications codes to configure the Impulse memory controller. In order to keep the memory controller simple and fast, Impulse restricts the remappings in two ways: first, the size of the data item being remapped must be a power of two; second, an application (or compiler/OS) that uses Impulse ensures data consistency through appropriate flushing of the caches.

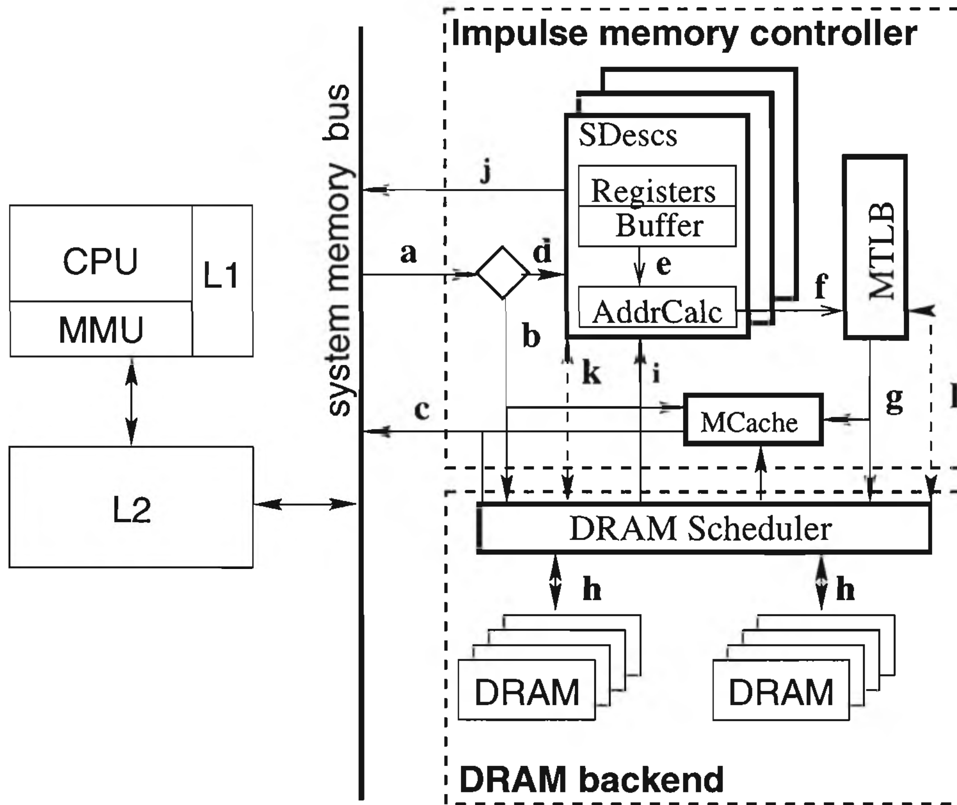


Figure 1: The Impulse memory architecture. The arrows indicate how data flow within an Impulse memory system.

2.1 Hardware

Figure 1 illustrates the Impulse memory architecture. The Impulse memory controller includes following components:

- a small number of *Shadow Descriptors (SDescs)*, each of which contains some *registers* to store remapping information, a small *SRAM buffer* to hold remapped data prefetched from DRAMs, the logic to assemble sparse data retrieved from DRAMs into dense cache lines, and a simple *ALU unit (AddrCalc)* to translate shadow addresses to pseudo-virtual addresses;
- a *Memory Controller TLB (MTLB)*, which is backed up by main memory and maps pseudo-virtual addresses to physical DRAM addresses, along with a small DRAM buffer to hold prefetched page table entries;
- a *Memory Controller Cache (MCache)*, which buffers non-remapped data prefetched from DRAMs;
- a *DRAM Scheduler*, which contains circuitry that orders and issues accesses to the DRAMs;
- DRAM chips, which constitute the main memory.

The extra level of address translation at the memory controller is optional, so an address appearing on the system memory bus may be a real physical or a shadow address (**a**). A real physical address passes untranslated to the MCache/DRAM scheduler (**b**). A shadow address has

to go through the matching shadow descriptor (**d**). The AddrCalc unit in the shadow descriptor translates the shadow address into a set of pseudo-virtual addresses using the remapping information stored in control registers (**e**). These pseudo-virtual addresses are translated into real physical addresses by the MTLB (**f**). Then the real physical addresses pass to the DRAM scheduler (**g**). The DRAM scheduler orders and issues the DRAM accesses (**h**) and sends the data back to the matching shadow descriptor (**i**). Finally, the appropriate shadow descriptor assembles the data into a cache line and sends it over the system memory bus (**j**).

2.2 Remapping Algorithms

The key of the Impulse project is finding the remapping schemes which can effectively improve applications' performance. Currently, the address translation at the memory controller can take four forms, depending on how Impulse is used to access a particular data structure: *direct, strided, scatter/gather using an indirection vector, or transpose.* Exploration of more efficient remapping algorithms is underway.

- *Direct mapping* maps one contiguous cache line in shadow address space into one contiguous cache line in real physical memory. The relationship between a shadow address *saddr* and its pseudo-virtual address *pvaddr* is (*pvaddr* = *pvsaddr* + (*saddr* − *ssaddr*)), where *pvsaddr/ssaddr* is the starting address (assigned by the OS) of the data structure's *pseudo-virtual address space image/shadow address space image.* Examples of using this mapping include recoloring physical pages without copying [5] and constructing superpages from non-contiguous physical pages without copying [12].

- *Strided mapping* creates dense cache lines from data items that are not contiguous but stridedly distributed in virtual address space. The mapping function maps the cache line addressed by a shadow address *saddr* to multiple pseudo-virtual addresses: (*pvsaddr* + *stride* × (*saddr* − *ssaddr*) + *stride* × *i*), where *i* ranges from 0 to ((*size of cache line*) / (*size of data item*) −1). This mapping can be used to create tiles of a dense matrix without copying [5] or to compact strided array elements.

- *Scatter/gather using an indirection vector* packs dense cache lines from array elements according to an indirection vector. The mapping function first computes the offset of a shadow address *saddr* in shadow address space *soffset* = *saddr* − *ssaddr*, then uses the indirection vector *vector* to map the cache line addressed by the shadow address *saddr* to several pseudo-virtual addresses (*pvsaddr* + *vector*[*soffset* + *i*]), where *i* ranges from 0 to ((*size of cache line*) / (*size of array element*) −1). The OS moves the indirection vector into contiguous physical memory so that the address translation for the indirection vector is not needed. One example of using this mapping is on sparse matrix-vector product algorithm [5].

- *Transpose mapping* creates the transpose of a two-dimensional matrix by mapping the element *transposed_matrix*[*j*][*i*] of the transposed matrix to the element *original_matrix*[*i*][*j*] of the original matrix. This mapping can be used wherever a matrix is accessed in a major different with what it is stored [15]. This mapping also can be easily expanded to support higher-dimension matrices. For example, taking each vector as an array element allows the same mapping to be applied in three-dimensional matrices.

Obviously, in direct mapping, each shadow address generates exactly one DRAM access; in other three mappings, each shadow address generates (*cache-line-size* / *sizeof(data item)*) DRAM ac-

cesses if (*cache-line-size* > *sizeof(data item)*), or one DRAM access if (*cache-line-size* ≤ *sizeof(data item)*).

3 DRAM Basics

To put the analyses in Section 6 in perspective, this section describes the basics of DRAM (Dynamic Random Access Memory) and two common types of DRAMs: Synchronous DRAM and Direct Rambus DRAM.

DRAM is arranged as a matrix of "memory cells" laid out in rows and columns, and thus a data access sequence consists of a *row access strobe* signal (RAS) followed by one or more *column access strobe* signals (CAS). During RAS, data in the storage cells of the decoded row are moved into a bank of sense amplifier (a.k.a *page buffer* or *hot row*), which serves as a row cache. During CAS, the column addresses are decoded and the selected data are read from the *page buffer*. Consecutive accesses to the current page buffer – called *page hits* – only need column addresses, saving the RAS signals. However, the hot row must first be closed before another row can be opened. DRAM also has to be refreshed hundreds of times each second in order to retain data in its memory cells.

3.1 Synchronous DRAM

SDRAM synchronizes all input and output signals to a system clock, therefore making the memory retrieval process much more efficient. In SDRAM, RAS and CAS signals share the same bus. SDRAM supports burst transfer to provide a constant flow of data. The programmable burst length can be two, four, eight cycles or a full-page. It has both "automatic" and "controlled" precharge commands, so a read or a write command can specify whether or not to leave the row open.

Figure 2 shows the sequences of some SDRAM transactions, assuming all transactions access the same bank. Part 1 of Figure 2 displays the interleaving of two read transactions directed to the same row without automatic precharge commands. The second read hits on the hot row, so it does not need RAS signals. Part 2 of Figure 2 shows the interleaving of two read transactions directed to two different rows without automatic precharge commands. Since the second read needs a different row, the previous hot row has to be closed (i.e., a precharge command must be done) before the second read can open a new row. Part 3 of Figure 2 shows two read transactions with automatic precharge commands (i.e., the row is automatically closed at the end of an access). When the automatic precharge is enabled, the sequence of two read transactions will be same no matter whether they access the same row or not. Part 4 of Figure 2 displays a write transaction followed by a read transaction which accesses a new row. An explicit precharge command must be inserted before the second transaction starts. Two restrictions are introduced by the write transaction: first, a delay ($tDPL$) must be satisfied from the start of the last write cycle until the precharge command can be issued; second, the delay between the precharge command and the next activate command (RAS) must be greater than or equal to the precharge time (tRP). Figure 2 also shows the key timing parameters of the SDRAM [3]. Their meanings and typical values in SDRAM clock cycles are described in Table 1. Assume SDRAM's clock rate is 147 MHz.

3.2 Direct Rambus DRAM

The Direct Rambus DRAM is a high speed DRAM developed by Rambus, Inc [6]. The RDRAM has independent pins for row address, column address, and data. Each bank can be independently opened, accessed, and precharged. Data and control information is transferred to and from the

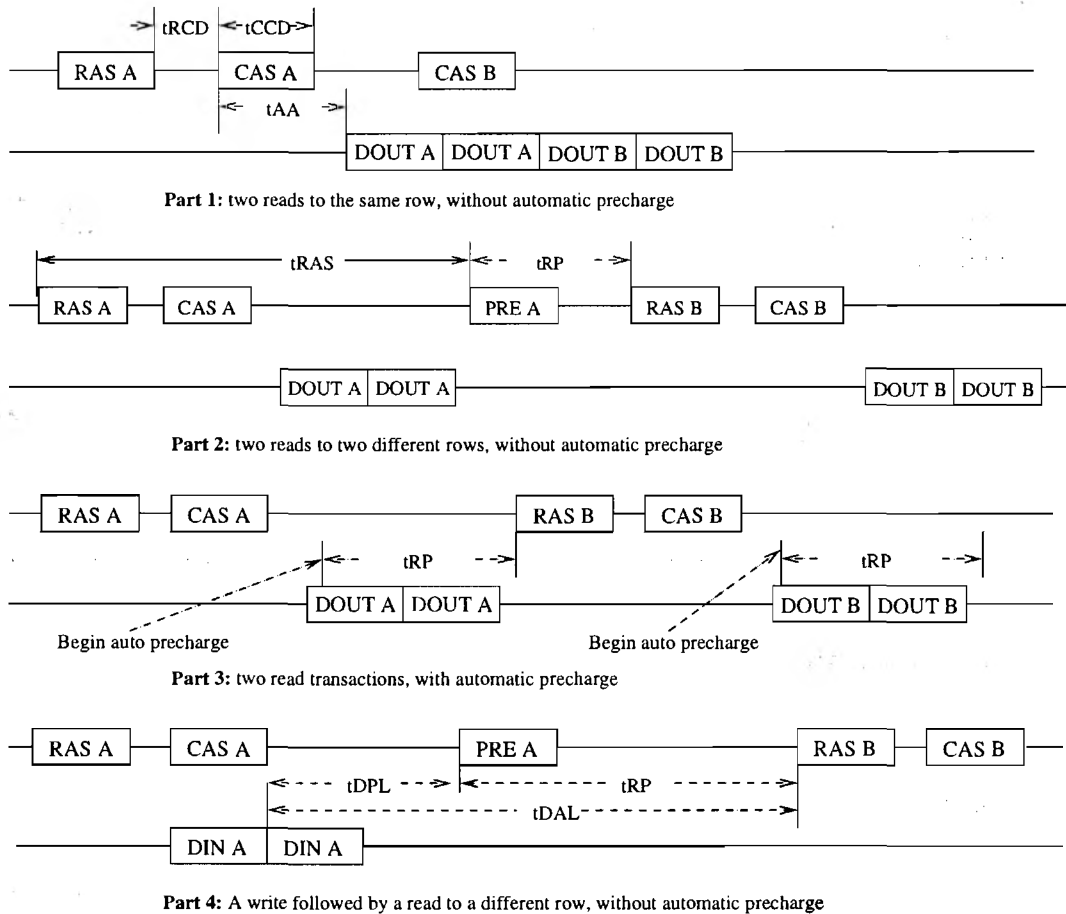


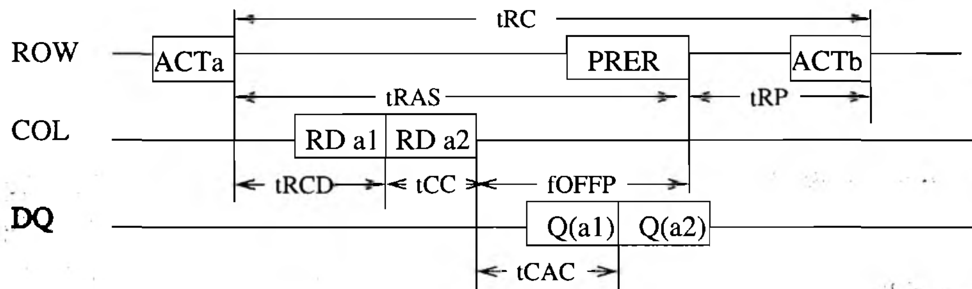
Figure 2: Examples of the sequences of some SDRAM transactions

Symbol	Meaning	Value
$tRAS$	minimum bank active time	7
$tRCD$	RAS to CAS delay time	3
tAA	CAS latency	3
$tCCD$	CAS to CAS delay time	1
tRP	precharge time	3
$tDPL$	data in to precharge time	2
$tDAL$	data in to active/refresh time (equals to $tRP + tDPL$)	5

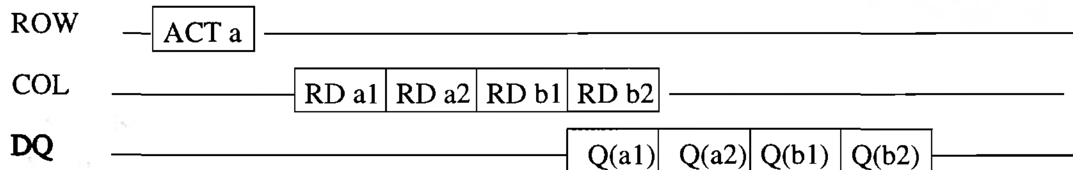
Table 1: Important timing parameters of Synchronous DRAM.

RDRAM in a packet-oriented protocol. Each of the packets consists of a burst of eight bits over the corresponding signal lines of the channel.

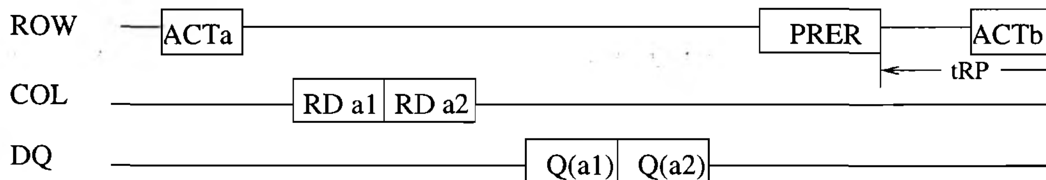
Figure 3 shows the basic operations of some RDRAM transactions, assuming all the transactions access the same chip. The first part of Figure 3 shows a read transaction with a precharge command, followed by another transaction to the same bank. The second part of Figure 3 shows the effective overlapping between two read transactions directed to the same row. The third part of Figure 3 shows a read transaction without precharge command followed by a transaction to a different row.



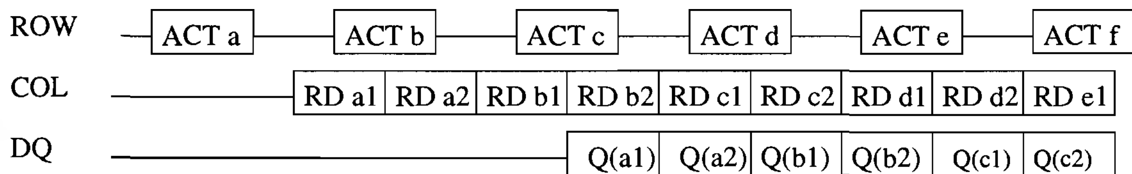
Part 1: A read transaction with precharge followed by another read.



Part 2: Two read transactions to the same row.



Part 3: A read transaction without precharge followed by an explicit precharge command



Part 4: Ideal interleaving of transactions directed to non-adjacent banks

Figure 3: Examples of RDRAM operations

Just as with SDRAM, the hot row has to be explicitly precharged before the second transaction. The fourth part of Figure 3 displays an ideal steady-state sequence of dual-data read transactions directed to non-adjacent banks of a single RDRAM chip. The key timing parameters of RDRAM [2] and their typical values in RDRAM clock cycles are presented in Table 2. Assume RDRAM's clock rate is 400 MHz.

4 DRAM Backend for The Impulse Memory System

The Impulse DRAM backend[1] is constructed from three major components: the DRAM Dispatcher, Slave Memory Controller (SMC), and plug in memory modules — DRAM chips. The DRAM dispatcher, SMCs, and the connecting wires between them – RAM Address bus (RA bus) – constitute the DRAM scheduler shown in Figure 1. A DRAM backend contains one DRAM dispatcher, but

[1]Since the Impulse memory system was modeled based on the HP Kitt-Hawk memory system [1], this paper follows the terminology of Kitt-Hawk memory system.

Symbol	Meaning	Value
tRC	the minimum delay from the first ACT command to the second ACT command	28
$tRAS$	the minimum delay from an ACT command to a PRER command	20
$tRCD$	delay from an ACT command to its first RD command	7
tRP	the minimum delay from a PRER command to an ACT command	8
$tCAC$	delay from a RD command to its associated data out	8
tCC	delay from a RD command to next RD command	4
$tOFFP$	the minimum delay from the last RD command to a PRER command	3
$tBUB1$	bubble between a RD and WR command	4
$tBUB2$	bubble between a WR and RD command to the same device	8

Table 2: Important timing parameters of Rambus DRAM.

can have multiple SMCs, multiple RA busses, and multiple plug in memory modules. Figure 4 shows a simple configuration with four SMCs, four DRAM chips (each of them has two banks), and two RA busses. Bear in mind that the DRAM dispatcher and SMCs don't have to be in different chips. Figure 4 just shows them in a way easy to understand. Whether or not to implement the DRAM scheduler in a single chip is an open question.

The Master Memory Controller [14] (MMC) is the core of the Impulse memory system. It communicates with the processors and I/O adapters over the system memory bus, translates shadow addresses into physical DRAM addresses, and generates DRAM accesses. A DRAM access can be either a shadow access or a non-shadow access. The MMC sends requests to the DRAM backend via Slave Address busses (SA bus) and passes data from or to the DRAM backend via Slave Data busses (SD bus). In our experimental model, the number of SA busses/SD busses can vary from one to one plus the number of total shadow descriptors. If there is only one SA bus or SD bus, both non-shadow accesses and shadow accesses will share it. If there are two SA busses or SD busses, non-shadow accesses will use one exclusively and shadow accesses will use the other one exclusively. If there are more than two SA busses or SD busses, one will be exclusively used by non-shadow accesses and each of the rest will be used by a subset of the shadow descriptors. The contention on SA busses is resolved by the MMC and the contention on SD busses is resolved by the DRAM dispatcher. One goal of this project is to find out how many SA busses/SD buses are needed to avoid heavy contention on them. Later experimental results will show that more than one SA bus or SD bus won't give significant benefit over single SA bus or SD bus.

4.1 DRAM Dispatcher

The DRAM dispatcher is responsible for sending memory accesses coming from SA busses to the relevant SMC via RA busses and passing data between SD busses and RAM Data busses (RD bus). If there is more than one SA bus, contention on RA bus occurs when two accesses from two different SA busses simultaneously need the same RA bus. For the same reason, contention on SD busses or RD busses will occur if there is more than one RD bus or more than one SD bus. The DRAM dispatcher resolves all the contentions by picking a winner according to a designated algorithm and queuing the others. If a waiting queues becomes critically full, the DRAM dispatcher will stop the sender (either MMC or SMC) from sending more requests. No sophisticated scheduling algorithms have been applied on the waiting queues. All waiting queues work in First-Come-First-Serve (FSFC) order. Normally, most waiting transactions are on RD busses, so the DRAM

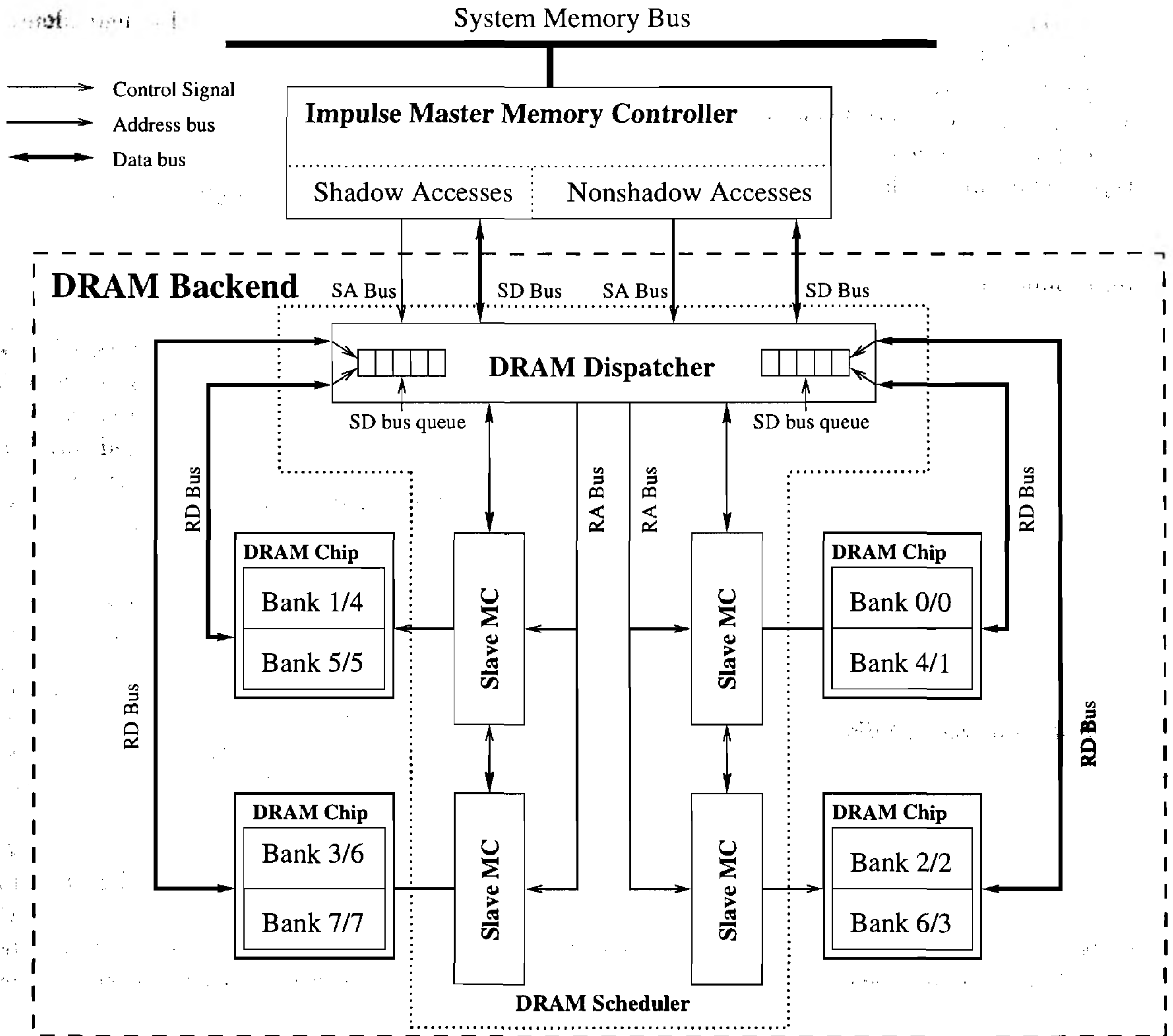


Figure 4: Impulse DRAM Backend Block Diagram

dispatcher holds a configurable number of waiting queues (called *SD bus queue*[2]) for transactions coming off RD busses. The SD bus queue has two uses: to buffer transactions coming from RD busses so that the RD busses can be freed up for other transactions; to resolve contention on SD busses by queuing all other contenders except the winner. Each SD bus queue is connected to an exclusive subset of RD busses. Later results will show one SD bus queue is enough to make itself not be the bottleneck in the DRAM backend.

4.2 Slave Memory Controller and DRAM Chip

Each Slave Memory Controller controls one RD bus and several DRAM chips sharing that RD bus. The SMC has independent control signals for each DRAM chip. The basic unit of memory

[2]Conceptionally, *SD bus queue* equals to the Jetway in old technical reports.

is a *memory bank*. Each memory bank has its own page buffer and can be accessed independently from all other banks. Some RDRAM chips let each page buffer to be shared between two adjacent banks, which introduces the restriction that adjacent banks may not be simultaneously accessed. We approximately model this type of RDRAM by making the effective independent banks be half of its physical number of banks. How many banks each DRAM chip has depends on the DRAM type. Typically, each SDRAM chip contains two to four banks and each RDRAM chip contains eight to 16 banks.

SMC is responsible for several important tasks. First, SMC keeps track of each memory bank's page buffer and decides whether or not to leave page buffer open after an access. Second, SMC controls an independent waiting queue for each bank and schedules the transactions in the waiting queue with the intention of reducing the average memory latency. Third, SMC manages the interleaving of memory banks. When an access is broadcasted on a RA bus, all SMCs on the RA bus will see it, but only the SMC which controls the memory to which that the access goes will respond. The interleaving scheme determines which SMC should respond to a specified physical address. Fourth, SMC is responsible for DRAM timing and refreshing DRAM chips periodically.

4.3 Algorithms in the DRAM Backend

This section illustrates several algorithms implemented in our DRAM backend: **hot row policy** which decides whether or not to leave a hot row open at the end of an access; **bank queue reordering algorithm** which reorders the transactions in order to minimize the average memory latency perceived by the processor; **interleaving scheme** which determines how the physical DRAM addresses are distributed among DRAM banks.

4.3.1 Hot Row Policy

In order to save the RAS signals, the Impulse DRAM backend allows hot rows to remain active after being accessed. The size and number of the hot rows vary with the type of DRAM chips and the number of memory banks. The collection of hot rows can be regarded as a cache. Proper management is necessary to make this "cache" profitable. The benefit to leave a row open is that the DRAM access latency is reduced due to eliminating the RAS signals if a DRAM access hits the open row. However, a DRAM access has to pay the penalty of closing the open row if it misses the open row. Three precharge policies were tested in our experiment: *close-page* policy, where the active row is always closed after an access; *open-page* policy, where the active row is always left open after an access; *use-predictor* policy, where predictors are used to guess whether the next access to an open row will be a hit or a miss.

The *use-predictor* policy was initially designed by R.C. Schumann [10]. In this policy, a separate predictor is used for each potential open row. Each predictor records the hit/miss results for the previous several accesses to the associated memory bank. If an access to the bank goes to the same row as the previous access to the same bank, it is recorded as a hit no matter whether the row was kept open or not. Otherwise, it is recorded as a miss. The predictor then uses the multiple-bit history to predict whether the next access will be a hit or a miss. If the previous recorded accesses are all hits, it predicts a hit. If the previous recorded accesses are all misses, it predicts a miss. Since the optimum policy is not obvious for the other cases, a software-controlled precharge policy register is provided to define the policy for each of all the possible cases. Application can set this register to specify the desired policy or can disable the hot row scheme altogether by setting the register to zeros. In our experiment, the precharge policy register is set "open" when there are more hits than misses in the history and "close" when there are more misses than hits or the same

number of misses as hits in the history. For example, if the history has four-bit, the precharge policy register is set to be 1110 1000 1000 0000 upon initialization, which keeps the row open whenever three of the preceding four accesses are page hits.

We expanded the original use-predictor policy with one more feature: when the bank waiting queue is not empty, use the first transaction in the waiting queue instead of the predictor to make decision. If next transaction goes to the same row as current transaction does, the row is left open after current transaction. Otherwise, the row is closed.

4.3.2 Bank Waiting Queue Reordering

There are many different types of DRAM accesses that the MMC can send to the DRAM backend. Figure 1 shows flows to the DRAM backend from different units. Based on the issuer and the nature of a DRAM access, each DRAM access is classified as one of following four types.

- *Direct access* is generated by real physical address directly coming off from system memory bus (arrow **b** in Figure 1). Since each normal memory request is for a cache line, each direct access requests a cache line from the DRAM backend.

- *Indirection vector access* is generated by the shadow descriptors to fetch the indirection vector during the translation for *scatter/gather using an indirection vector* (**k**). Each indirection vector access is for a cache line and the return data are sent back to the relevant shadow descriptor.

- *MTLB access* is generated by the MTLB to fetch page table entries from DRAMs into the MTLB buffers (**l**). To reduce the number of MTLB accesses, each MTLB access requests a whole cache line, not just a single entry.

- *Shadow access* is any DRAM access generated by the Impulse memory controller to fetch remapped data (**g**). The size of each shadow access varies with application-specific mappings. The data of shadow accesses are returned back to the remapping controller for further processing.

This paper also uses another definition – *non-shadow access*, which includes direct access, indirection vector access, and MTLB access. Normally, most of DRAM accesses are either direct accesses or shadow accesses, with a few or none being MTLB accesses and indirection vector accesses. Intuitively, different types of DRAM access should be treated differently in order to reduce the average memory latency. For example, an indirection vector access is depended on by a bunch of shadow accesses and its waiting cycles directly contribute to the latency of the associated memory request, so it had better be taken care of as early as possible. Any delay on a prefetching access will not likely increase the average memory latency as long as the data are prefetched early enough, which is easy to accomplish in most situations, so a prefetching access does not have to complete as early as possible and it can give away its memory bank to more important accesses like indirection vector accesses and MTLB accesses. After having taken consider of those facts, we propose a reordering algorithm with following rules.

1. Ensure the consistency. Make sure no violation of data dependencies — read after write, write after read, and write after write.

2. Once an access is used to make decision that whether or not to leave a row open at the end of the preceding access (see Section 4.3.1), no other accesses can get ahead of it and it's guaranteed to access the relevant memory bank right after the preceding one.

3. Give normal (non-prefetching) access higher priority over prefetching access.
4. Give MTLB access and indirection vector access the highest priority so that these accesses can be finished as early as possible, therefore releasing their dependent shadow accesses as early as possible.
5. It's hard to determine by imagination whether the direct access or the shadow access should be given higher priority over each other. For experimental purpose, this rule has two opposite choices: giving direct access higher priority over shadow access; or giving shadow access higher priority over direct access.
6. Increase each access' priority along with its increasing waiting time. This rule guarantees no access would stay in a bank waiting queue "forever". We make this rule optional so that we can find out if it's useful and if there are cases that some accesses stay in waiting queues for a very very long time.
7. If there is a conflict among some rules, always use the rule which is the earliest in above sequence among those conflicting rules.

4.3.3 Memory Bank Interleaving

The interleaving of memory banks controls the mapping from physical DRAM addresses to memory banks. Figure 4 shows two interleaving schemes. The first one numbers the physically-adjacent memory banks as separately as possible. Its mapping function from the memory banks to the SMCs is (*SMC-id* = *bank-id* MOD *number of SMCs*). The second one numbers the physically-adjacent memory banks with consecutive numbers. Its mapping function is (*SMC-id* = *bank-id* / *banks-per-chip*). We call the first one *modulo-interleaving* and the second one *sequential-interleaving*. Each scheme can be either page-level or cache-line-level. So there are total four interleaving schemes modeled in our simulator: page-level modulo-interleaving, page-level sequential-interleaving, cache-line-level modulo-interleaving, and cache-line-level sequential-interleaving. In the simulator, page size is the size of page buffer in each bank and the cache-line size equals to the size of a second level cache line. When we say the banks are interleaved at the page-level, it means bank 0 has all pages whose address modulo page-size is 0, bank 1 has all pages whose address modulo page-size is 1, and so on.

5 Experimental Framework

5.1 Simulation Environment

The executive-driven simulator Paint [11, 13] was extended to model the Impulse memory controller and the proposed DRAM backend. Paint models a variation of a single-issue HP PA-RISC 1.1 processor running a BSD-based micro-kernel and an HP Runway bus. The 32K L1 data cache is non-blocking, single-cycle, write-around, write-through, virtually indexed, physically tagged, and direct mapped with 32-byte lines. The 256K L2 data cache is non-blocking, write-allocate, write-back, physically indexed and tagged, 2-way set-associative, and has 128-byte lines. Instruction caching is assumed to be perfect. The TLB is unified I/D, single-cycle, and fully associative, uses a not-recently-used replacement policy, and has 120 entries.

The simulated Impulse memory controller is deprived from the HP memory controller [8] used in servers and high-end workstations. We model seven shadow descriptors, each of which is associated with a 512-byte SRAM buffer. The controller prefetches the corresponding shadow data into these

fully associative buffers of four 128-byte lines. A 4K-byte SRAM holds non-shadow data prefetched from DRAMs. The MTLB has an independent bank for each shadow descriptor. Each MTLB bank is direct-mapped, has 32 eight-byte entries (the same size as the entries in the kernel's page table), and includes two 128-byte buffers used to store consecutive lines of page table entries prefetched on an MTLB miss.

Even though Paint's PA-RISC processor is single-issue, we model a pseudo-quad-issue superscalar machine by issuing four instructions each cycle without checking structural hazards. While this model is unrealistic for gathering processor micro-architecture statistics, it stresses the memory system in a manner similar to a real superscalar processor.

5.2 Benchmarks

Following benchmarks are selected to test the performance of proposed Impulse DRAM backend.

- **CG** [4] from NPB2.3 uses conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. The kernel of CG is a sparse matrix vector product (SMVP) operation. Two Impulse optimizations can be applied on this benchmark independently: scatter/gather through an indirection vector (call it **CG.iv** in following discussion) and direct mapping. Assume it computes $A \times P$, where A is the sparse matrix and P is the vector. Scatter/gather through an indirection vector remaps the vector P to improve its cache performance. Direct mapping implements no-copy page-coloring which maps P to the first half of L2 cache and other data structures to the second half of L2 cache. There are two versions of page-coloring: one remaps only the three most important data structures in CG (call it **CG.pc3**); another one remaps all the seven major data structures in CG (call it **CG.pc7**).

- **Spark98** [9] performs a sequence of sparse matrix vector product operations using matrices that are derived from a family of three-dimensional finite element earthquake applications. It uses scatter/gather through indirection vector to remap the vector used in SMVP operations. In Spark98, each vector element is a 3×1 sub-vector. In order to meet the Impulse restriction that the size of remapped data item must be a power of two, the sub-vector is padded to be 4×1.

- **TMMP** is an Impulse-version implementation of the tiled dense matrix-matrix product algorithm. Impulse remaps a tile of each matrix into a contiguous space in shadow address space using *strided mapping*. Assume it computes $C = A \times B$. Impulse divides the L1 cache into three segments. Each segment keeps a tile: the current output tile from C; the input tiles from A and B. In addition, since the same row of matrix A is used multiple times to compute a row of matrix C, it is kept in the L2 cache during the computation of relevant row of matrix C.

- **Rotation** rotates an image by performing three one-dimensional shears. The second shear operation (the one along the y axis, assuming the image array is stored along x axis.) walks along the column of the image matrix, which gives poor memory performance for large images. Impulse can transpose matrices without copying it. So walking along column in the image is replaced by walking along row in a transposed matrix. Each shear operation involves one input image and one output image. Both input image matrix and output image matrix are remapped using *transpose remapping* during the second shear operation.

- **ADI** implements the naive "Alternating Direction Implicit" integration algorithm. The ADI integration algorithm contains two phases: sweeping along row and sweeping along column. Impulse transposes the matrices in the second phase so that sweeping along column in the original matrices is replaced by sweeping along row in the transposed matrices. The algorithm involves three matrices. All of them are remapped using *transpose remapping* during the second phase.

All of these benchmarks gain decent performance benefits from the Impulse remapping optimizations. Where the benefit comes from and how much benefit each benchmark achieves is beyond this paper. Instead, this paper focuses on how the DRAM backend impacts the performance. These benchmarks represent some different ways to use the Impulse remapping functionality. Table 3 lists the problem size and the Impulse-related resources that each benchmark uses. "Remapping" means the remapping algorithm that this benchmark uses. "Descriptor" represents the total number of shadow descriptors used by this benchmark. "DRAM Accesses" indicates the total number of DRAM accesses needed to compact a cache line in shadow address space, provided 128-byte second level cache line.

Benchmarks	Problem Size	Remapping	Descriptors	DRAM Accesses
CG.iv	Class A	scatter/gather	1	16
CG.pc	Class A	direct	3	1
CG.pc7	Class A	direct	7	1
spark98	sf.5.1	scatter/gather	4	4
TMMP	512x512(double)	strided	3	16
Rotation	1024x1024(char)	transpose	2	128
ADI	1024x1024(double)	transpose	3	16

Table 3: The problem size, the remapping scheme, and the number of shadow descriptors used by each benchmark; and the number of DRAM accesses generated for a shadow address.

5.3 Methodology

The factors that we care about includes the interleaving scheme, the waiting queue reordering algorithm, the hot row policy, the number of DRAM banks, the number of RD busses (or slave memory controllers), the number of SA busses, the number of SD busses, the number of SD bus queues, and the DRAM type (SDRAM or RDRAM?). Each factor can have multiple levels. It's infeasible to use the full factorial design. So we have to pick up a practicable subset of all factors at a time to find out their relative impacts.

In our experiment, we first set up a baseline and then vary one or a few factors at a time. The baseline uses cache-line-level modulo-interleaving, no waiting queue reordering algorithm, and close-page hot row policy, and has four DRAM chips, eight banks (for SDRAM) or 32 banks (for RDRAM), four RD busses, two SD bus queues, one SA bus, and one SD bus.

6 Performance

In our simulations, the CPU clock rate is 400 MHz. The SDRAM works at 147 MHz. Each SDRAM chip contains two banks, each of which has a 16K-byte page-buffer. The RDRAM works at 400

MHz. Each RDRAM chip contains 8 banks, each of which has an 8k-byte page-buffer. The DRAM width is 16bytes, so are the RD busses and SD busses. An SA bus can transfer a request per cycle.

A DRAM access latency, defined to be the interval between the time that the MMC generates the DRAM access and the time the MMC receives the return data from the DRAM backend, is broken down into five pieces:

- *SA cycles* – cycles spent on SA bus, including waiting time and transferring time;
- *SD cycles* – cycles spent on SD bus, including waiting time and transferring time;
- *RD waiting cycles* – cycles waiting for RD bus;
- *Bank waiting cycles* – cycles spent on bank waiting queue;
- *Bank access cycles* – time actually accessing a memory bank.

The data transferring cycles on SA/SD/RD busses are inevitable and can never be changed. The bank access cycles are inevitable too, but they can be reduced by an appropriate hot row policy. The cycles waiting for SA/SD/RD bus or DRAM bank are absolutely unnecessary and should be avoided as much as possible. Reducing the waiting cycles is the main goal of our DRAM backend.

6.1 The Impact of DRAM Organization

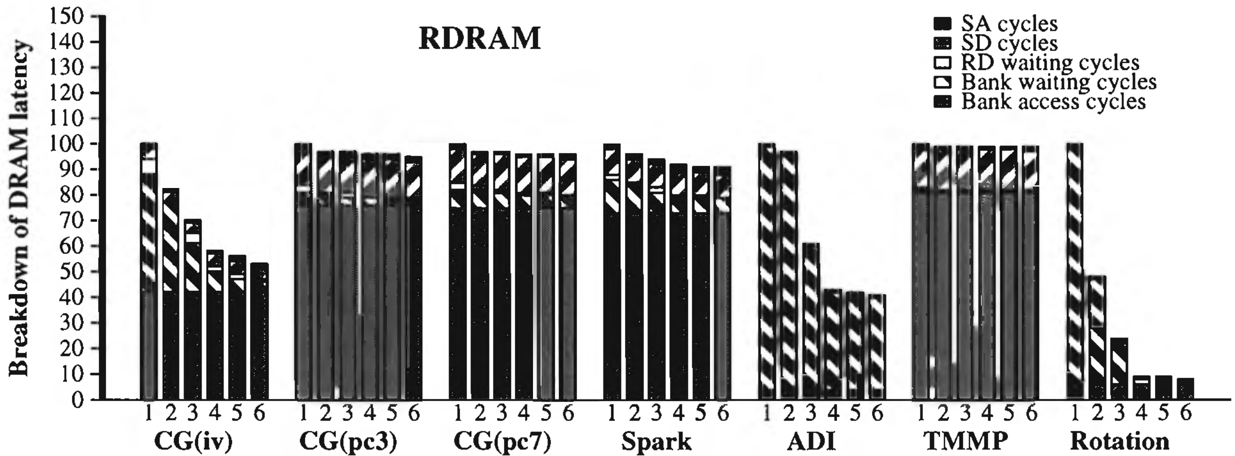


Figure 5: Breakdown of the average RDRAM access latency at 6 different DRAM organizations (in the sequence of memory bank, RD bus, SD bus queue): **1** – 32/2/1; **2** – 32/4/2; **3** – 64/4/2; **4** – 128/8/4; **5** – 256/8/4; **6** – 256/16/8.

The memory bank, RD bus, and SD bus queue are tightly related to one another. So we consider them together as one compound factor — DRAM organization. Apparently, they can affect the bank waiting cycles, RD waiting cycles, and SD cycles, but will not affect the SA cycles and bank access cycles. Intuitively, increasing the number of memory banks should decrease the bank waiting cycles, increasing the number of RD busses should decrease the RD waiting cycles, and increasing the number of SD bus queues should decrease the SD cycles.

Figure 5 shows the breakdown of the average DRAM access latency for each benchmark at six different DRAM organizations based upon RDRAM. Figure 6 shows the same results for SDRAM.

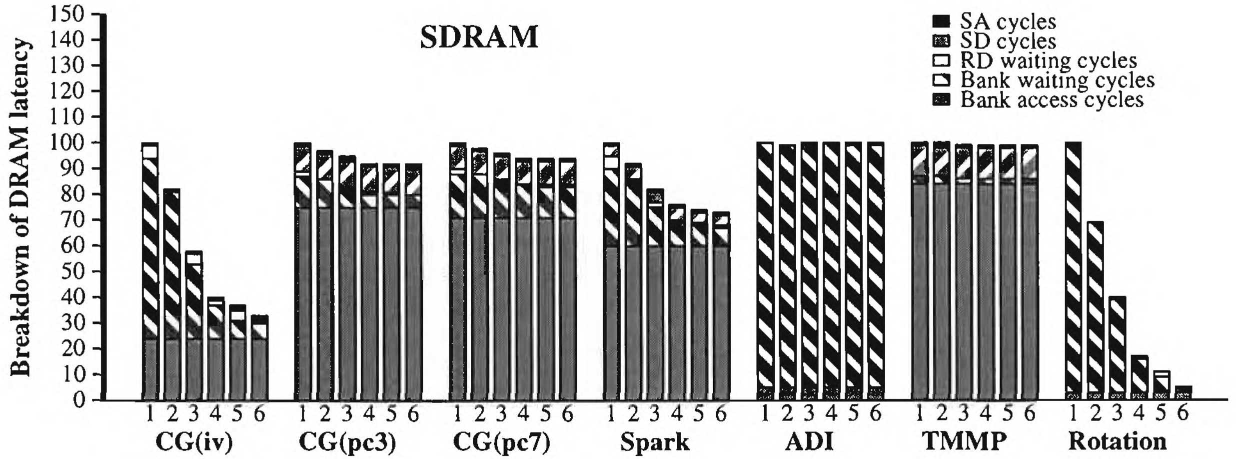


Figure 6: Breakdown of the average SDRAM access latency at 6 different DRAM organizations (in the sequence of memory bank, RD bus, SD bus queue): **1** - 8/2/1; **2** - 8/4/2; **3** - 16/4/2; **4** - 32/8/4; **5** - 64/8/4; **6** - 64/16/8.

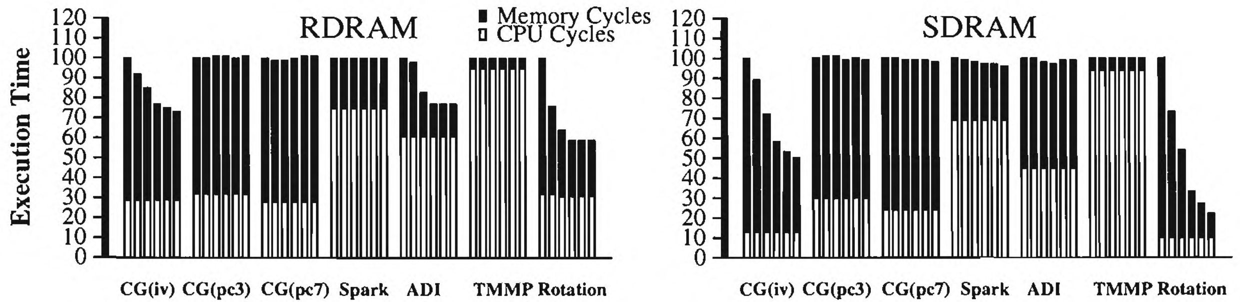


Figure 7: Execution time at 6 different DRAM organizations.

Figure 7 shows their total execution times. Except mentioned otherwise, all the results shown in this paper are normalized to the baseline's. The execution time contains two components: *memory cycles* and *CPU cycles*. The DRAM backend is targeted to attack the memory cycles only, so the CPU cycles will never change along with the changing of the DRAM backend. Because changing DRAM organization will not change the bank access cycles, there is no surprise to see there are only small performance changes seen in applications – **CG.pc3, CG.pc7, Spark, TMMP** – where the bank access cycles dominate the DRAM access latency. But for benchmarks – **CG.iv, ADI, Rotation** – where the bank waiting cycles are the dominant force of a DRAM access latency, increasing the number of memory banks, RD busses, and SD bus queues achieves close to or more than linear improvement on the bank waiting cycles. For example, compare configuration 1 to 6 in Figure 6, **Rotation** changes its average DRAM access latency from 2123 cycles (2067 are bank waiting cycles) to 109 cycles (52 are bank waiting cycles), which contributes 87% saving on total memory cycles and 78% saving on total execution time (that is $(1/(1\text{-}78\%)) = 4.55$ speedup!).

All benchmarks, except **ADI**, achieve similar results on both RDRAM and SDRAM. **ADI** gains huge improvement on RDRAM but very insignificant benefit on SDRAM when the DRAM organization is expanded. The reason is that SDRAM, which inherently has less banks than same capacity RDRAM, can't provide the minimum number of memory banks that ADI needs to avoid long bank waiting queues. Even in configuration 6, the average length of a bank waiting queue is 13.6 transactions for SDRAM. However, it's only 1.9 transactions for RDRAM.

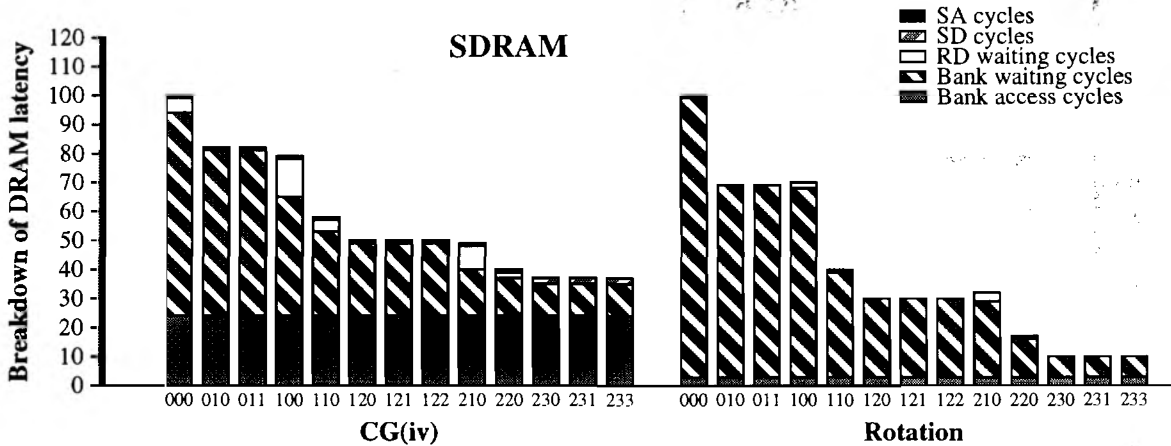


Figure 8: Breakdown of the average SDRAM access latency at different DRAM organizations. Each configuration is labeled as **ABC**, where **A** represents the number of memory banks: 0 – 8, 1 – 16, 2 – 32; **B** represents the number of RD busses: 0 – 2, 1 – 4, 2 – 8, 3 – 16; **C** represents the number of SD bus queues: 0 – 1, 1 – 2, 2 – 4, 3 – 8.

The ratio among the number of memory banks, RD busses, and SD bus queues has to be in a certain range in order to keep the DRAM backend balanced. When the ratio is beyond that range, increasing the number of one component will not increase the performance. More specifically, each RD bus or SD bus queue can serve only a certain number of memory banks. Letting it serve too few banks is wasting resource, and letting it serve too many banks will offset the benefits obtained from the increased number of memory banks. Figure 8 shows the performance of **CG.iv** and **Rotation**, two DRAM organization-sensitive benchmarks, under variant DRAM organizations based on SDRAM. The results for RDRAM are not shown here because they are very similar with SDRAM's. Two interesting facts can be found in Figure 8. First, whenever the ratio between the number of memory banks and the number of RD busses drops from higher value such as 8:1 or 4:1 to 2:1 (e.g. from 000 to 010, 100 to 120, or 210 to 230 in Figure 8), there is a big drop in the average DRAM access latency. This fact implies that the best ratio between the number of memory banks and the number of RD busses is 2:1. Since each SDRAM chip contains two banks, we also can put it in another way: in order to avoid significant contention on RD busses, one RD bus can only be used by one SDRAM chip. Second, increasing the number of SD bus queues (010 to 011, 120 to 122, and 230 to 233) does not increase performance noticeably. This fact implies that one SD bus queue can match up with at least 16 RD busses. Based on those two facts, we can easily reach a conclusion: a cost-effective, balanced DRAM backend has one RD bus for each DRAM chip and has very few number of SD bus queues.

6.2 The Impact of Slave Busses

Three possible settings about slave busses have been modeled. The results are shown in Figure 9, 10, and 11. The first setting is one SA bus and one SD bus only. The second setting has two SA busses and two SD busses. The third one uses the maximum setting – eight SA busses and eight SD busses. In the second setting, shadow access uses one SA/SD bus and non-shadow access uses the another one. In the third setting, each of the seven shadow descriptors uses one SA/SD bus and non-shadow access uses the remaining one.

Surprisingly, none of those benchmarks wastes significant amount of time waiting for slave

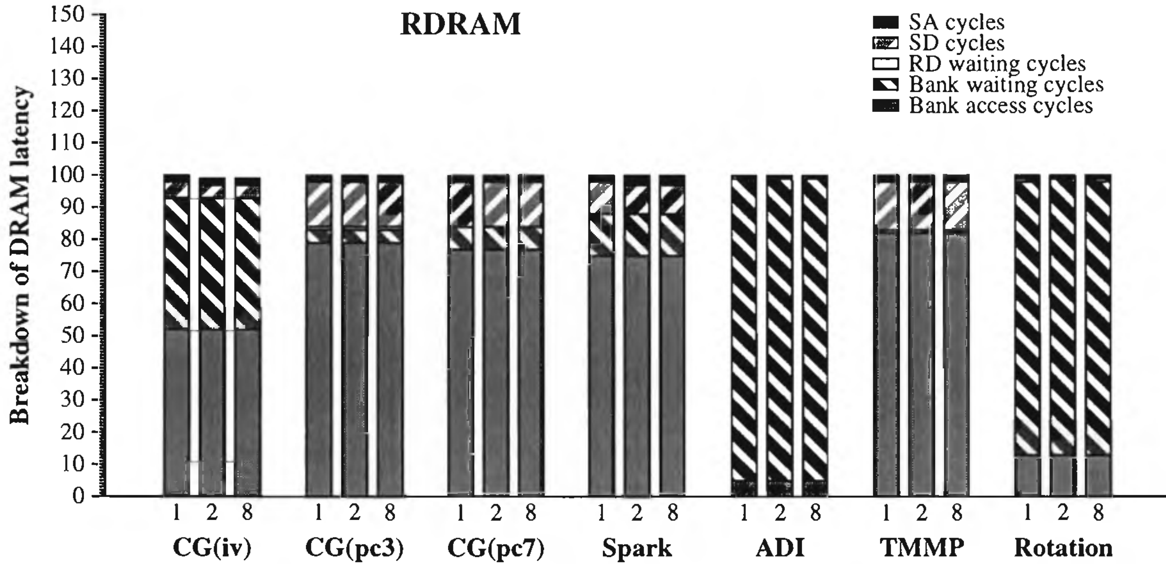


Figure 9: Breakdown of the average RDRAM access latency for different number of slave busses: **1** – 1 SA bus, 1 SD bus; **2** – 2 SA busses, 2 SD busses; **8** – 8 SA busses, 8 SD busses.

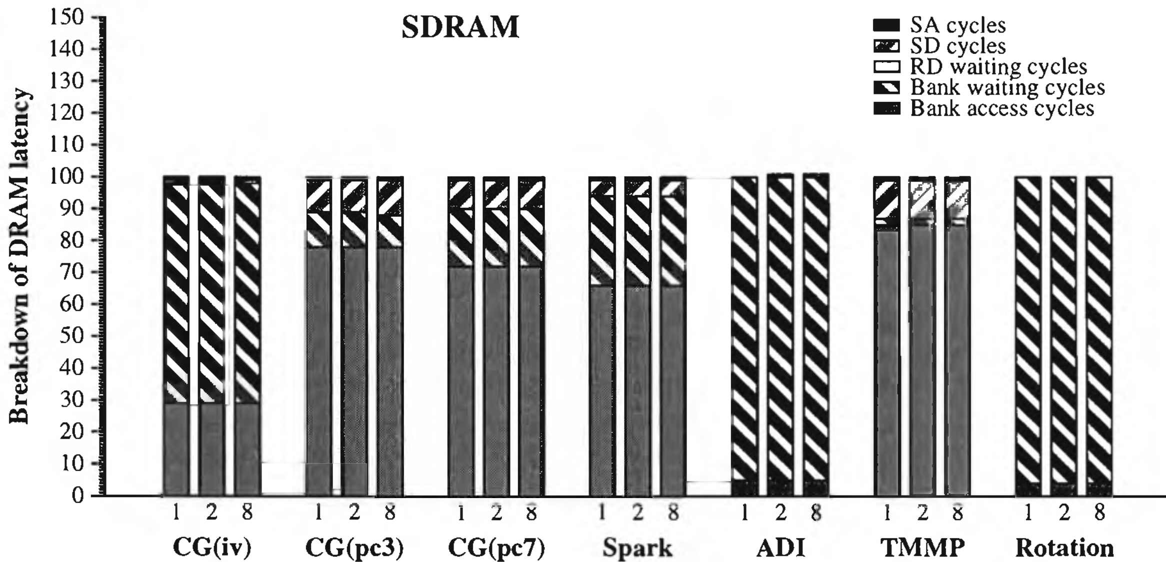


Figure 10: Breakdown of the average SDRAM access latency for different number of slave busses: **1** – 1 SA bus, 1 SD bus; **2** – 2 SA busses, 2 SD busses; **8** – 8 SA busses, 8 SD busses.

	CG.iv	CG.pc3	CG.pc7	Spark	ADI	TMMP	Rotation
cycles(RDRAM)	1.1/4.5	1.0/2.6	1.0/2.6	1.1/4.0	2.8/5.3	1.0/1.6	1.3/2.8
cycles(SDRAM)	1.1/2.6	1.0/2.6	1.0/2.6	1.0/3.7	2.7/4.1	1.0/1.6	1.9/2.2

Table 4: The average cycles of each SA/SD wait in the baseline execution.

busses. Table 4 lists the average waiting cycles of an SA or SD wait in the baseline execution of each benchmark. Compared to the average DRAM access latency, the average SA/SD waiting cycles are negligible. Though the average waiting time may go up to 5.3 cycles (**ADI** with RDRAM), it

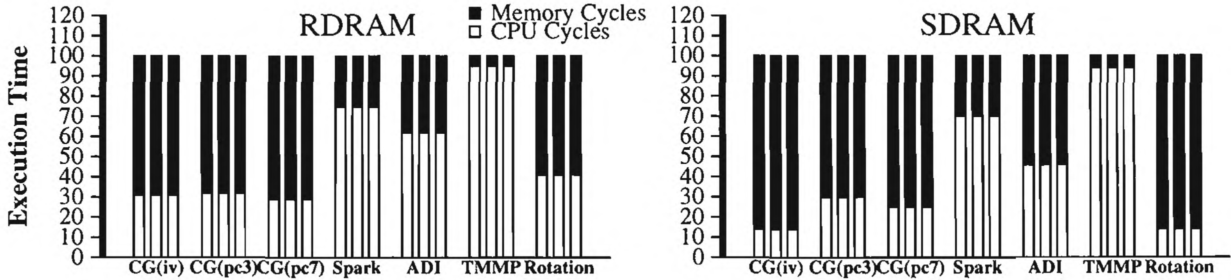


Figure 11: Execution times for different number of slave busses

still does not hurt performance because the DRAM backend gets extremely busy and the average DRAM access latency goes up to hundreds or even thousands of cycles whenever there is noticeable contention like this on slave busses. Comparing to hundreds of cycles, 5.3 cycles are too tiny to make difference. If we look back to Figure 5 and 6, we also can see that the SA cycles and the SD cycles are the same for almost any configurations. This implies that slave busses doesn't get any busier relative to other components of the DRAM backend even when the DRAM backend can deliver the data several times faster. Based on those facts mentioned above, we can conclude that one SA bus and one SD bus is enough for the Impulse DRAM backend. This conclusion also converges with the one obtained in last section – that one SD bus queue is enough.

6.3 The Effect of Hot-row Policy

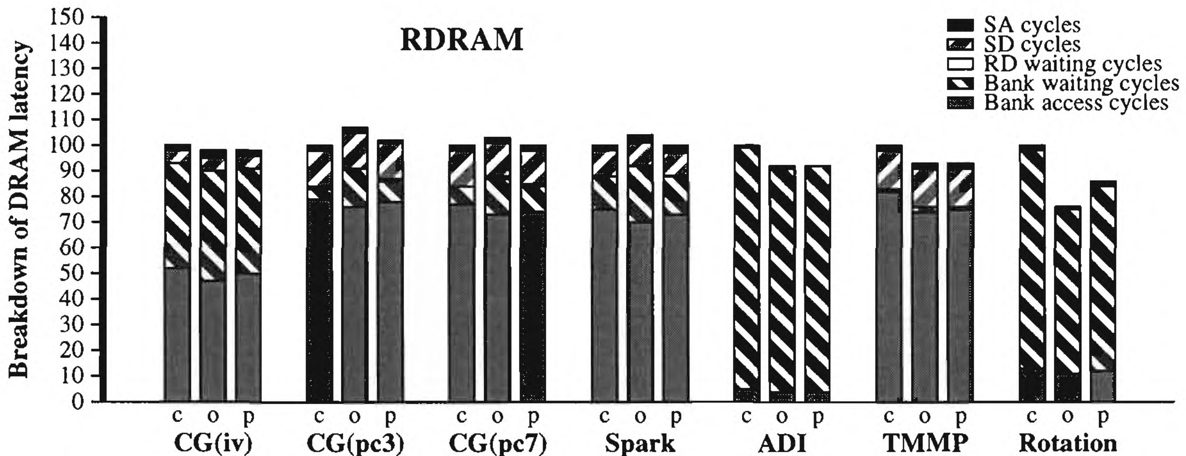


Figure 12: Breakdown of the average RDRAM access latency for 3 different hot row policies: **c** – close-page; **o** – open-page; **p** – use-predictor.

Figure 12 and 13 display the results of three different hot-row algorithms: close-page policy (**c**), open-page policy (**o**), and use-predictor policy(**p**). In the use-predictor policy, the predictor has 4-bit history. The next access is predicted to be a "hit" if there are at least three hits in the history, and a "miss" otherwise. The main goal of a hot row policy is to reduce the *bank access cycles*. It also has a nice side-effect that, when the *bank access cycles* of an access is reduced, following accesses in the waiting queue can be issued earlier and consequently those accesses' *bank waiting cycles* will be decreased. Table 5 displays the hot row hit and miss ratio under the open-page policy and use-predictor policy. The hit ratio equals to (total hot row hits / total accesses) and the miss

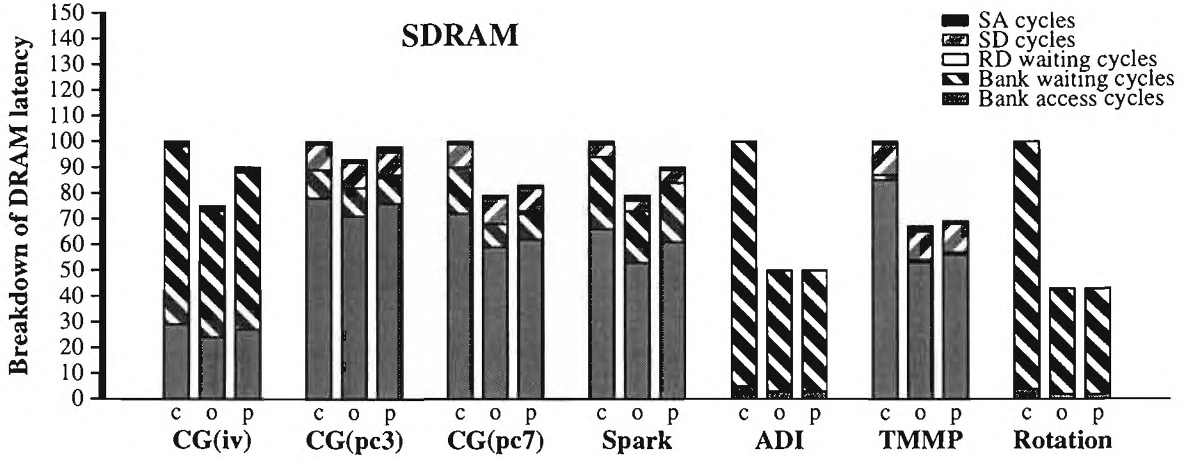


Figure 13: Breakdown of the average SDRAM access latency for 3 different hot row policies: **c** – close-page; **o** – open-page; **p** – use-predictor.

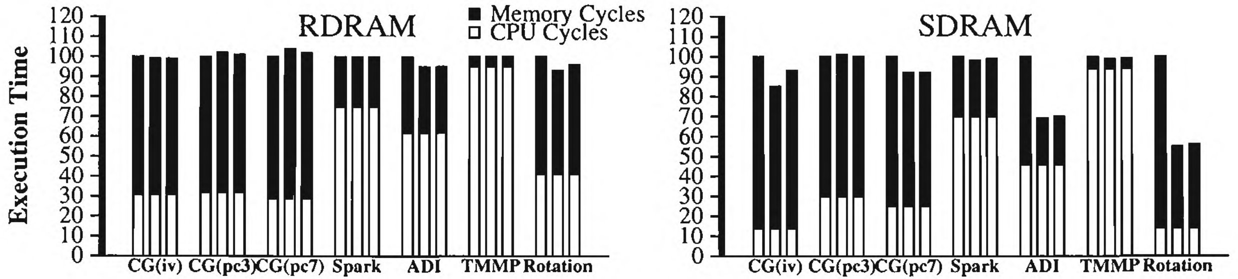


Figure 14: Execution time for 3 different hot row policies.

ratio equals to (total hot row misses / total accesses). If a transaction accesses a bank without active hot row, it's taken neither as a hot row hit nor as a hot row miss. Remember, hot row is closed during refresh operations, so the hit ratio plus miss ratio for the open-page policy does not have to be 100%.

	CG.iv	CG.pc3	CG.pc7	Spark	ADI	TMMP	Rotation
open-page(RDRAM)	37/63	20/80	37/63	34/66	71/29	68/7	40/59
predictor(RDRAM)	13/18	9/32	29/28	13/20	69/18	64/6	23/19
open-page(SDRAM)	19/81	20/80	37/63	27/73	64/36	72/10	63/36
predictor(SDRAM)	9/7	5/15	24/15	11/17	63/24	65/9	63/25

Table 5: The hot row hit/miss ratios.

It's not clear that which policy is the best because the benchmarks behave very differently under different hot row policies. In order to reach more concrete conclusion, let's first take a look at Figure 2 and 3 to quantify the benefit of hitting a row and the penalty of missing a row. For RDRAM, the hit benefit is $tRCD + tOFFP + tRP - tCC = 14$ cycles for (read, read)[3], or $14 - tBUB1 = 10$ cycles for (read, write), or $14 - tBUB2 = 6$ cycles for (write, X); and the miss

[3]Represents (type of previous access, type of current access). "X" means either read or write.

penalty is $tRP = 8$ cycles. For SDRAM, the hit benefit is $tCCD + tRCD = 3$ cycles; and the miss penalty is the minimum of tRP and the CAS count of previous transaction.

Theoretically, the impact of a hot row policy on a special run is "closely" proportional to

$$(\text{total-hits} \times \text{average-hit-benefit} - \text{total-misses} \times \text{average-miss-penalty}). \quad (1)$$

We say it is "closely" because the hit benefit and miss penalty may change when the time interval between two consecutive transactions varies. In order to achieve positive impact, equation 1 has to be larger than 0, i.e., the ratio between total hits and total misses has to be larger than (average-miss-penalty / average-hit-benefit). This ratio between total hits and total misses is probably the best leverage to measure a hot row policy's performance.

The theoretical results obtained using equation 1 match with the experimental results almost perfectly. We are not going to check the matches one by one because most of them are obvious. Instead, we are going to look at a couple of non-obvious cases. Look at the open-page policy in Table 5 and Figure 12, you can find **CG.iv** and **CG.pc7** have the same hit/miss ratio, but different average DRAM access latency. It's resulted from their different ratios between reads and writes. In **CG.iv**, more than 98.3% of DRAM accesses are reads. In **CG.pc7**, only 83% of DRAM accesses are reads. Whenever a write transaction is involved in RDRAM, the hit benefit will be decreased from 14 cycles to 10 or 6 cycles. So though **CG.pc7** has the same hot row hit/miss ratio as **CG.iv**'s, it's still outperformed by **CG.iv** because its higher percentage of write transactions introduces a smaller average hit benefit. Another weird example, occurred when SDRAM is used, is on **Rotation**, which has different miss ratios but the same performance for the open-page policy and the use-predictor policy. Most of the DRAM accesses in **Rotation** are shadow accesses requesting a eight-byte double word, so the average miss penalty (equals to MIN(tRP, CAS count of previous transaction)) is close to the average CAS count — 1. With 1-cycle miss penalty and 3-cycle hit benefit, the 11% difference on the miss ratio can not make big difference. (The actual difference on execution time is 0.14% that is too small to show.)

The use-predictor policy always stays between the close-page policy and open-page policy. Wherever the open-page policy is helpful, the use-predictor policy will be helpful too, but with smaller benefit. Wherever the open-page policy hurts the performance, the use-predictor policy will hurt the performance too, but with a much smaller degree. Though both policies may degrade the performance by up to 4%, they can improve the performance by up to 44%. In addition, they gain positive impact most of the time. The results implies that both the open-page policy and use-predictor policy are fine choices for the hot row policy in the Impulse DRAM backend. We suggest the use-predictor policy because it's stabler than the open-page policy.

6.4 The Effect of Bank-queue Reordering Algorithm

Total 6 different bank-queue reordering algorithms have been used in our experiment. The first one is no reordering, or say it's FCFS (number it as No.1). The other 5 are the alternatives of the algorithm described in section 4.3.2. No.2 is giving direct access priority over shadow access, but without priority updating – step 6 in the original algorithm. No.3 is giving shadow access priority over direct access, without priority updating. No.4 is giving shadow access and direct access the same priority, without priority updating. No.5 is the same as No.2 except it allows priority updating, i.e., increasing priority with increased waiting time. No.6 is the same as No.3 except it allows priority updating.

The highest priority in our simulation is 15. The priority vector, indexed by the sequence of MTLB access, indirection vector access, direct access, and shadow access followed by their prefetching versions in the same order, is: $\{15, 15, 11, 9, 7, 7, 3, 1\}$ for No.2 and No.6; $\{15, 15, 9, 11, 7, 7, 1, 3\}$

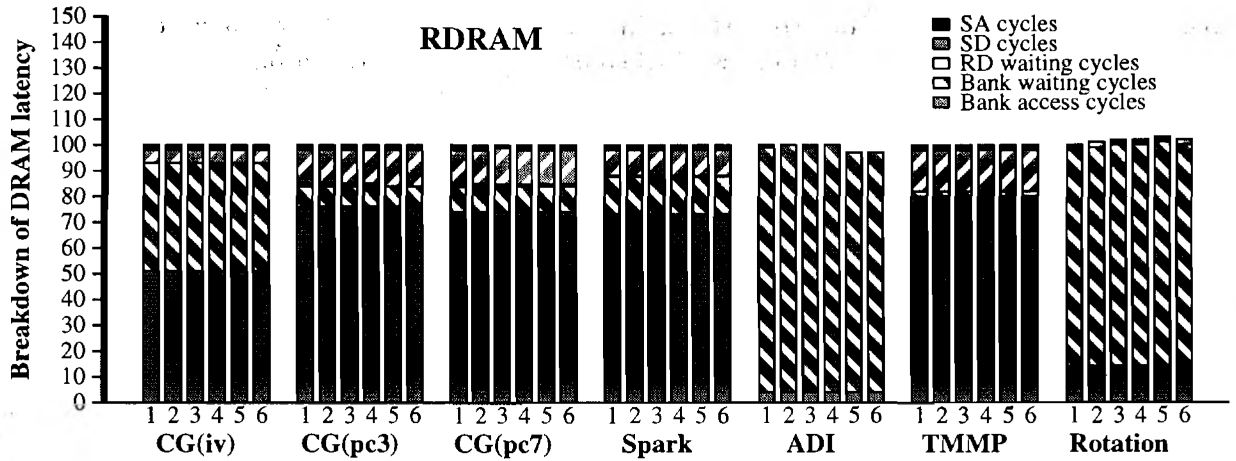


Figure 15: Breakdown of the average RDRAM access latency on different bank-queue reordering algorithms.

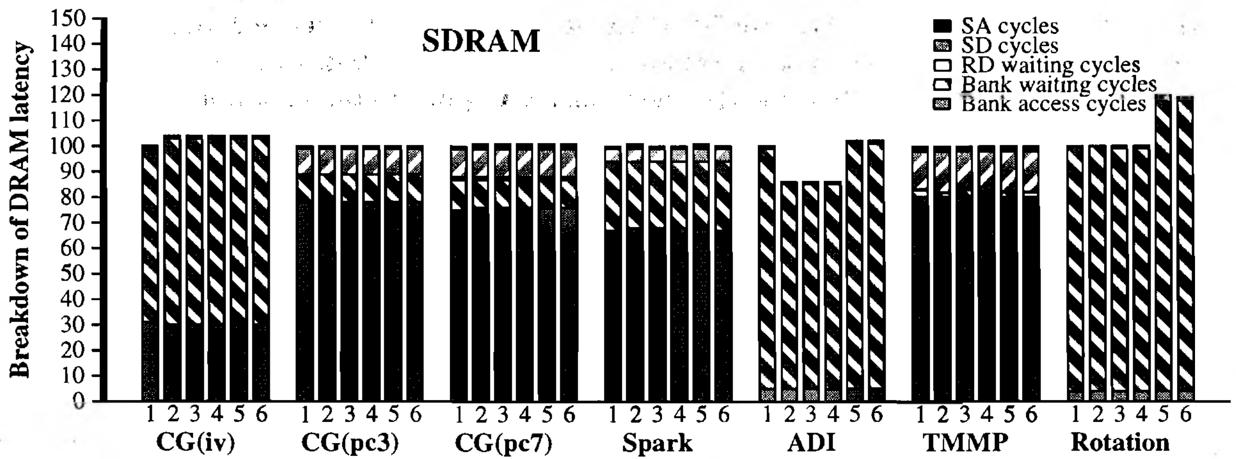


Figure 16: Breakdown of the average SDRAM access latency on different bank-queue reordering algorithms.

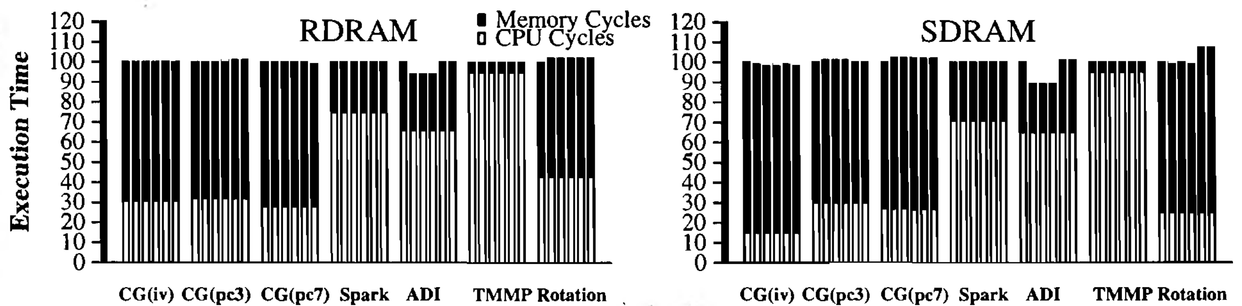


Figure 17: small Execution time on different bank-queue reordering algorithms.

for No.3 and No.5; $\{15, 15, 10, 10, 7, 7, 2, 2\}$ for No.4. The updating policy increases an access's priority by 1 whenever it's overtaken by another access. For example, in No.2, one non-prefetching shadow access with initial priority being 9 may be overtaken by accesses with higher priority for at most 6 times. Once it has given away its position for 6 times, it will have the highest priority and no accesses after it can get ahead of it.

The reordering algorithm will reduce the *bank access cycles* by using the first transaction on the waiting queue to make always-right decision about whether or not to leave a hot row open after an access. But it will never directly change other parts of a DRAM access latency. The preliminary premise to make the reordering useful is that the waiting queues reach a certain length. If the waiting queues are always short, like only 0 or 1 transaction in a queue most of the time, the reordering can not do anything because there is nothing to be scheduled. Table 6 lists the average queue length in the baseline execution of each benchmark. It shows only **ADI** and **Rotation** have long waiting queues. Consequently, they are the only benchmarks noticeably impacted by the reordering algorithm (from -7% to 11% on execution time).

	CG.iv	CG.pc3	CG.pc7	Spark	ADI	TMMP	Rotation
RDRAM	0.04	0.01	0.01	0.01	19.36	0.00	0.73
SDRAM	0.77	0.02	0.09	0.10	19.92	0.01	12.18

Table 6: The average bank queue length.

The performance numbers about reordering algorithm are in Figure 12, 13, and 14. The first fact to be noticed from the results is that No.2, No.3, and No.4 always perform closely. Taking a closer look at DRAM access patterns, we found either most of DRAM accesses are direct accesses or most of DRAM accesses are shadow accesses at a short period of time in the DRAM backend, which results in every few exchanges between direct accesses and shadow accesses. This means we can just give direct access and shadow access the same priority to simplify the reordering algorithm. Another fact displayed by the results is that the updating rule is not helping. It does give a tiny benefit for **ADI** on RDRAM (3% on average DRAM access latency), but it significantly slows **Rotation** down (-19% on average DRAM access latency). The reason it slows **Rotation** down is that when the updating rule decreases the average DRAM access latency of a prefetching shadow access from 1350 cycles to 1173 cycles, it increases the average DRAM access cycles of a non-prefetching shadow access from 338 cycles to 596 cycles. This fact suggests that the updating rule has to be either dropped or modified in the final design. A natural modification will be no priority updating on prefetching accesses. That is part of future work.

6.5 The Effect of Interleaving Schemes

All of the four interleaving schemes mentioned in section 4.3.3 are compared here. Figure 18 and 19 show the DRAM access latency breakdown. Figure 20 shows their execution times. How well an interleaving scheme can perform heavily depends on applications' access patterns. There is no optimum scheme working for all benchmarks. Generally, for applications sequentially accessing data, the cache-line-level interleaving is better; for applications page-stridedly accessing data, the page-level interleaving is better. Also modulo-interleaving is always better than sequential-interleaving. Sequential-interleaving is bad because it conflicts with the spatial locality exhibited by most applications. It hurts performance by directing consecutive accesses for data with good spatial locality to the same chip, which limits the inherent parallelism of DRAM accesses. Spatial locality also requires the cache-line-level interleaving to ensure consecutive requests go to different memory banks. That's why the page-level interleaving can not work well with applications with good spatial locality. For example, **Spark** tends to put non-zero elements in a row close to one another, so the four DRAM accesses generated by a gather operation at the memory controller are likely directed to the same page. The page-level interleaving makes all four DRAM accesses go to the same bank and be served serially instead of in parallel, therefore dramatically increasing

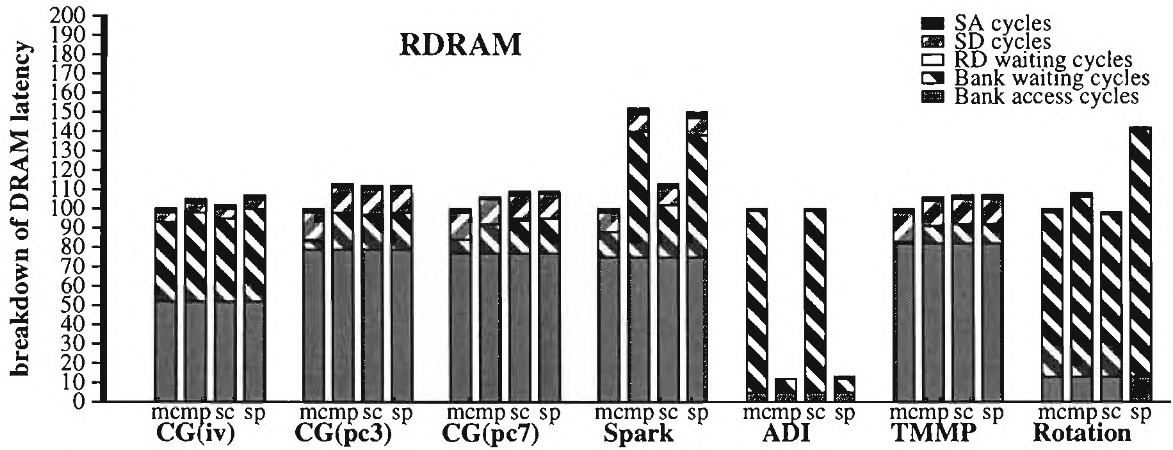


Figure 18: Breakdown of the average RDRAM access latency on different interleaving schemes: **mp** – modulo, page-level; **mc** – modulo, cache-line-level; **sp** – sequential, page level; **sc** – sequential, cache-line-level.

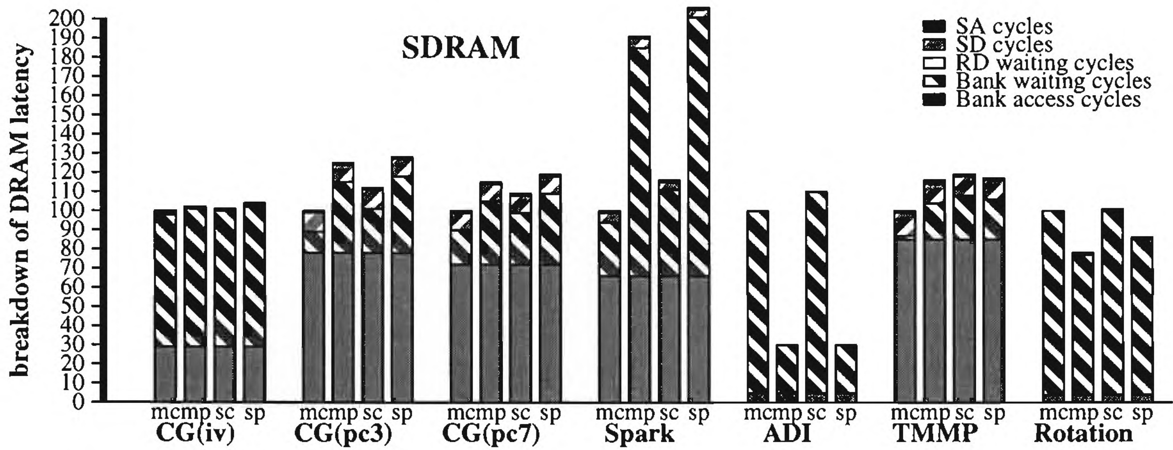


Figure 19: Breakdown of the average SDRAM access latency on different interleaving schemes: **mp** – modulo, page-level; **mc** – modulo, cache-line-level; **sp** – sequential, page level; **sc** – sequential, cache-line-level.

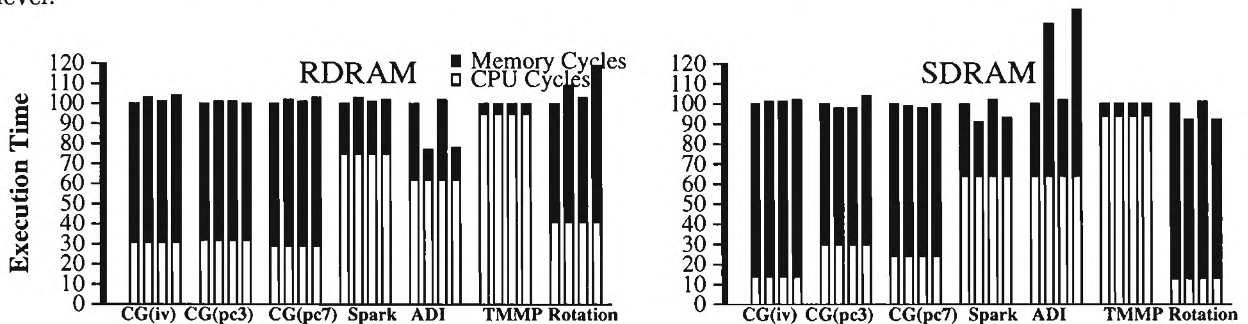


Figure 20: Execution time on different interleaving schemes.

the DRAM access latency. The experimental results show that, compared to the cache-line-level interleaving, the page-level interleaving increases Spark's average DRAM access latency by 52% on

RDRAM and 91% on SDRAM.

When **ADI** walks along a column of a 1024x1024 double array, it generates access sequence x, $x + 8K$, $x + 2 \times 8K$, ..., $x + 1023 \times 8K$. Note that in our model, the cache line is 128-byte, the number of banks is 8 for SDRAM and 32 for RDRAM, and the page size is 16K for RDRAM and 8K for RDRAM. If the interleaving is in cache-line-level, all the accesses will go to the same bank. If the interleaving is in page-level, the ith access $(x + i \times 8K)$ will go to bank $(i \% 32)$ if RDRAM is used, or bank $((i/2) \% 8)$ if SDRAM is used. Specifically speaking, all accesses go to the same bank in the cache-line-level interleaving while they are uniformly distributed among all banks in the page-level interleaving. That clearly proves why the page-level interleaving performs a lot better than the cache-line-level interleaving for **ADI** – about 30% saving on execution time.

Rotation operates on a 1024x1024 gray-scale image. Walking along a column of the image generates access sequence x, $x + 1K$, $x + 2 \times 1K$, ..., $x + 1023 \times 1K$. When SDRAM is used, all those accesses go to the same bank if the cache-line-level interleaving is used, or the first 16 accesses go to bank 0, the next 16 go to bank 1, ..., and so on if the page-level interleaving is used. That explains why the page-level interleaving is better than the cache-line-level interleaving for **Rotation** when SDRAM is used. When RDRAM is used, the ith access goes to bank $(i \% 4)$ in the cache-line-level interleaving, or the first 8 accesses go to bank 0, the next 8 go to bank 1, ..., and so on in the page-level interleaving. Though the page-level interleaving better distributes accesses among banks, the cache-line-level achieves better performance because it puts consecutive accesses into different banks, which is much better than putting consecutive accesses into the same bank like what the page-level interleaving does.

6.6 Putting It All Together

Based on the experimental results and the analyses presented above, we suggest our DRAM backend with following configuration: as many memory banks as possible, one RD bus for each DRAM chip, one SA bus, one SD bus, one SD bus queue, use-predictor hot row policy, No.4 accesses reordering algorithm, and cache-line-level modulo-interleaving. Since the cache-line-level interleaving may significantly slow down applications page-stridedly accessing its major data structures (such as ADI), the software (compiler or OS) pads each stride in those applications to an appropriate size (such as 4096 → (4096+128) in ADI) to avoid severe unbalanced loading in memory banks.

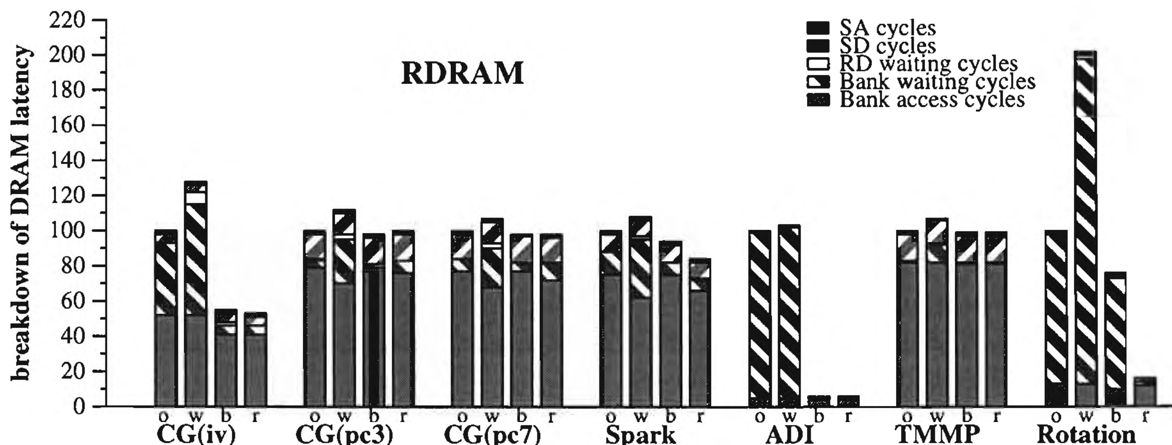


Figure 21: The average RDRAM access latency on four different DRAM backends: **o** – original baseline; **w** – worst; **b** – best; **r** – recommended.

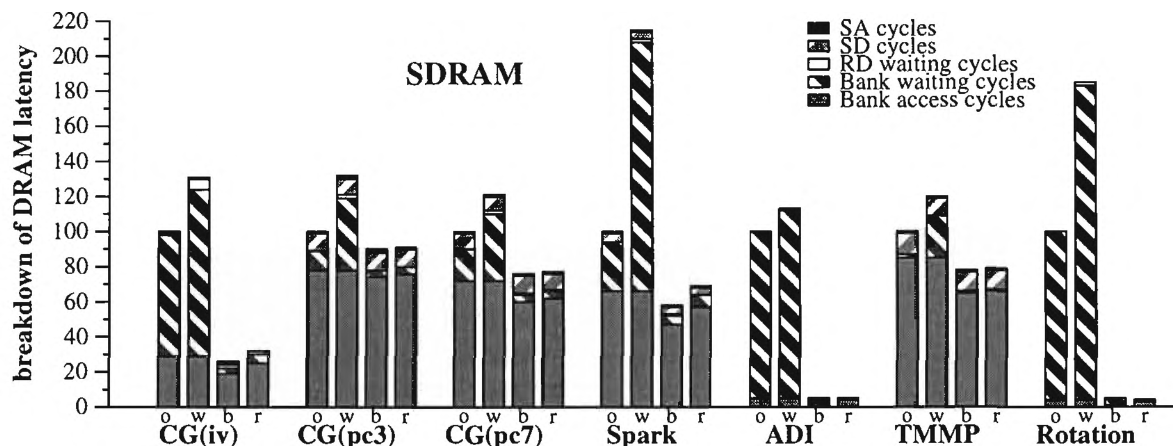


Figure 22: The average SDRAM access latency on four different DRAM backends: **o** – original baseline; **w** – worst; **b** – best; **r** – recommended.

Figure 21 and 22 present the performance numbers of four different DRAM backends: the original baseline (**o**); the configuration combined by the worst setting of each factor (**w**); the configuration combined by the best setting of each factor (**b**); the recommended configuration (**r**). These figures let us visualize how bad or how well a DRAM backend can be. The recommended configuration works pretty well in all benchmarks. It always works closely to or even better than the combination of the "bests". The interaction among different factors makes the configuration with the best value of each individual factor not always work best. Compare the recommended one to the baseline, the saving on the DRAM access latency ranges from 1% to 94% with a mean of 35% for RDRAM and from 9% to 96% with a mean of 49% for SDRAM. The modification on ADI works excellently too. It makes ADI work on the cache-line-level interleaving just as well as on the page-level interleaving.

7 Conclusion and Future Work

This paper described and evaluated a DRAM backend design for the Impulse memory system. The target of this project is not to design a real DRAM backend in hardware, instead it is to find out whether or not it's worthwhile to redesign the conventional DRAM backend for Impulse and to quantify how much the DRAM backend can affect the performance. The experimental results show the DRAM backend can change the memory cycles by from -69% to 89% and the execution time by from -41% to 72%, which implies that it is indeed necessary to design a specific DRAM backend for Impulse.

Some of the proposed algorithms in this paper is too complicated to be implemented by hardware. However, if we find a certain algorithm worth to implement and it is too much for hardware to handle, we will try to design a simpler algorithm which is suitable for the hardware implementation and is able to deliver acceptable performance.

There is lots of work needed to be done before we make final decisions. In current design, there is a queue for each memory bank. One potential useful alternative is keeping a queue for each DRAM chip, then performing higher-level scheduling such as bank conflicts avoiding. Another alternative is keeping a global queue for all banks, then performing optimum global scheduling. We can also extend the priority-based algorithm to include non-priority-based rules. For example, putting all

the transactions accessing the same row together to avoid RAS signals might be very helpful and will make the interaction between reordering according to priority and reordering according to row address be a very interesting topic. As to the interleaving of memory banks, more schemes, like double-word level or combinations of modulo and sequential interleaving might be interesting to exploit for the experimental purpose. The use-predictor policy also needs further exploitation: how many bits in history are enough? what's the best value for precharge policy register? Also, the paper doesn't specifically compare the RDRAM with the SDRAM, though there are enough data to make comparisons. Another very important feature missed by this paper is the interaction among all the factors of the DRAM backend. All the questions here are left to be answered in the future.

References

[1] Kitt Hawk Memory System, External Reference Specification, Revision B, May 1995.

[2] IBM Advanced 64Mb Direct Rambus DRAM, November 1997.

[3] IBM Advanced 256Mb Synchronous DRAM – Die Revision A, August 1998.

[4] D. Bailey, E. Barszca, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[5] J. B. Carter, W. C. Hsieh, L. B. Stoller, M. R. Swanson, L. Zhang, E. L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth IEEE Symposium on High Performance Computer Architecture*, pages 70–79, Orlando, Florida, January 1999.

[6] R. Crisp. Direct RAMBUS technology: The new main memory standard. *IEEE Micro*, pages 18–29, November 1997.

[7] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf. Access order and effective bandwidth for streams on a Direct Rambus memory. In *Proceedings of the Fifth IEEE Symposium on High Performance Computer Architecture*, pages 80–89, Orlando, Florida, January 1999.

[8] T. R. Hotchkiss, N. D. Marschke, and R. M. McClosky. A new memory system design for commercial and technical computing products. *Hewlett-Packard Journal*, 47(1):44–51, February 1996.

[9] D. R. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, School of Computer Science, Carnegie Mellon University, October 1997.

[10] R. Schumann. Design of the 21174 memory controller for digital personal workstations. *Digital Technical Journal*, 9(2), November 1997.

[11] L. Stoller, M. Swanson, and R. Kuramkot. Paint: PA instruction set interpreter. Technical Report UUCS-96-009, University of Utah, September 1996.

[12] M. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 204–213, Barcelona, Spain, June 1998.

[13] J. E. Veenstra and R. J. Fowler. MINT tutorial and user manual. Technical Report 452, University of Rochester, August 1994.

[14] L. Zhang. ISIM: The simulator for the Impulse adaptable memory system. Technical Report UUCS-99-017, University of Utah, September 1999.

[15] L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee. Memory system support for imaging processing. pages 98–107, Newport Beach, CA USA, October 1999.