A DRAM Backend for The Impulse Memory System

Lixin Zhang

UUCS-00-002

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

December 16, 1998

Abstract

The Impulse Adaptable Memory System exposes DRAM access patterns not seen in conventional memory systems. For instance, it can generate 32 DRAM accesses each of which requests a four-byte word in 32 cycles. Conventional DRAM backends are optimized for accesses that request full cache lines. They may not be able to handle smaller accesses effectively.

In this document, we describe and evaluate a DRAM backend that reduces the average DRAM access latency by exploiting the potential parallelism of DRAM accesses in the Impulse system. We design the DRAM backend by studying each of its important design options: DRAM organization, hot row policy, dynamic re-ordering of DRAM accesses, and interleaving of DRAM banks. The experimental results obtained from the execution-driven simulator Paint [10] show that, compared to a conventional DRAM backend, the proposed backend can reduce the average DRAM access latency by up to 98%, the average memory cycles by up to 90%, and the execution time by up to 80%.

This effort was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of DARPA, AFRL, or the US Government.

Contents

1 Introduction

The Impulse memory system adds two important features to a traditional memory system. First, it supports application-specific optimizations through configurable physical address remapping. By remapping physical addresses at the memory controller, applications can control how their data is accessed and cached, thereby improving cache performance and bus utilization. Second, it can prefetch data from DRAM to an SRAM buffer in the memory controller. For accesses that hit in the SRAM buffer, Impulse effectively hides DRAM access latency from the processor.

As a result, Impulse exhibits DRAM access patterns different with what a conventional memory system does. For example, it may gather a 128-byte cache line by generating 32 four-byte DRAM accesses directed to 32 different memory locations. Since conventional DRAM backends are designed to handle accesses that fetch cache lines, they may not work well with smaller DRAM accesses. To further improve the performance of the Impulse memory system, we explore the potential of redesigning the DRAM backend for Impulse.

To handle the large number of small DRAM accesses in the Impulse memory system, a DRAM backend for Impulse must be able to exploit the inherent parallelism of those DRAM accesses. The design options that can significantly affect the efficiency of such a backend include DRAM organization, hot row policy, access scheduling, and bank interleaving. DRAM organization determines 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. Hot row policy tries to reduce the average DRAM access latency by judiciously opening/closing the hot rows of DRAMs. Access scheduling reorders DRAM accesses to explore parallelism. The interleaving of memory banks may affect the performance dramatically because it directly determines the potential parallelism that a sequence of DRAM accesses might have.

The remainder of this document is organized as follows. Section 2 provides the overview of the Impulse memory system, focusing on the master memory controller. Section 3 provides some background information about two common types of DRAMs: Synchronous DRAM and Direct Rambus DRAM. Section 4 describes the proposed DRAM backend. Section 5 describes the simulation environment and the benchmarks used in our experiments. Section 6 presents the performance results. Section 7 discusses future work and concludes this document.

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 32-bit physical addressing and only one gigabytes of installed DRAM, the other three gigabytes of physical address space are not directly backed up by DRAM and will generate errors if presented to a conventional memory controller. We call these otherwise-unused physical addresses *shadow addresses*, and they constitute a *shadow address space*. In an Impulse system, applications can reorganize their data structures in the shadow address space to explicitly control how their data is accessed and cached. When the Impulse memory controller receives a shadow address, it will translate the shadow address to a set of "real" physical addresses (a.k.a physical DRAM addresses) instead of generating an error as a conventional memory controller does. In the current Impulse design, the mapping from the shadow address space to the real physical address space can be in any power-of-two granularity from word-size to page size.

Data items whose virtual addresses are not contiguous can be mapped to contiguous shadow addresses, so that sparse data items in virtual memory can be compacted into dense cache lines in shadow memory before being transferred to the processor. To map data items in these compacted cache lines back to physical memory, Impulse must recover their offsets within the virtual layout of the original data structures. We call these offsets *pseudo-virtual addresses*. Pseudo-virtual memory mirrors real virtual memory and is necessary to map data structures larger than a page. The memory controller translates *pseudo-virtual addresses* to physical DRAM addresses in page-level. 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 – integrated at the memory controller stores data prefetched from DRAM. For this document, we assume a simple next-line sequential prefetch scheme for MC-based prefetching: when an access misses in the MCache, fetch the requested cache line and prefetch the next one; when an access hits in the MCache, prefetch the next one. For normal data, prefetching is useful for reducing the memory latency of sequentially-accessed data. For shadow data, prefetching enables the controller to hide the cost of remapping shadow addresses and issuing multiple DRAM accesses.

The shadow address space is managed by the operating system in a way similar to real physical address space. The operating system guarantees the shadow address space image of any remapped shadow region to be contiguous even it spans multiple pages. This guarantee not only simplifies the translation hardware at the memory controller, but also allows the CPU to use superpage TLB entries to translate remapped data. The operating system provides an interface for applications to specify optimizations for their particular data structures and configure 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 to configure the Impulse memory controller. To keep the memory controller simple and fast, Impulse restricts remapping in two ways. First, any data item being remapped must be a power of two in size. Second, an application that uses remapping must ensure data consistency through appropriate flushing of the caches.

2.1 Hardware Organization

Figure 1 shows the block diagram of the Impulse memory system, which includes the following components:

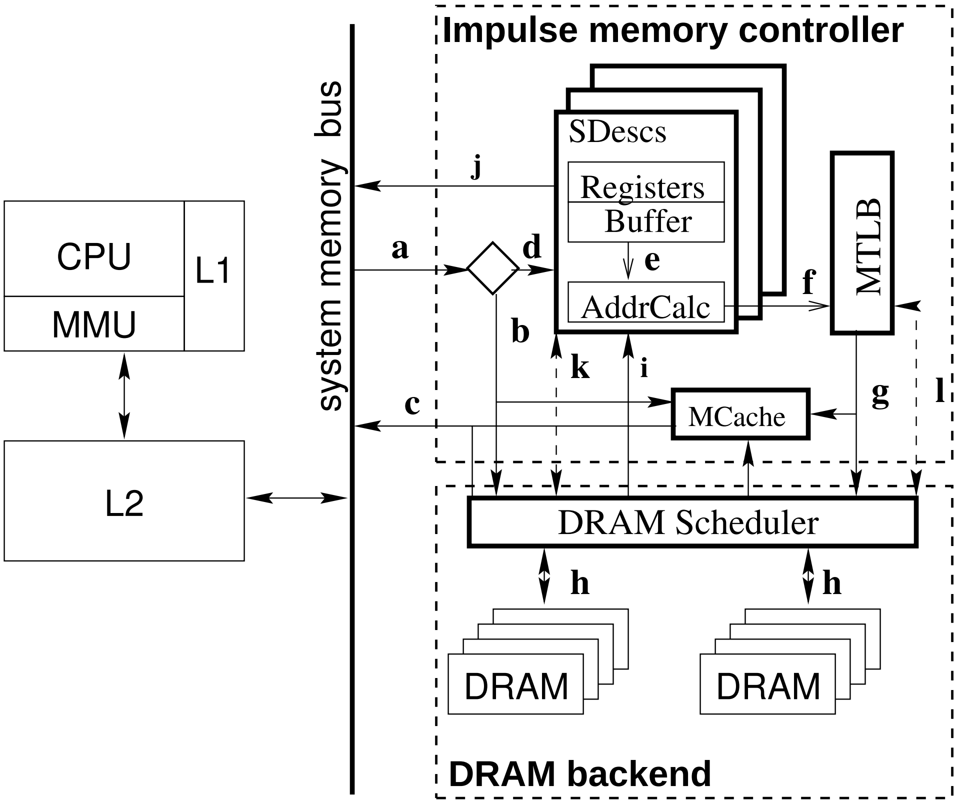


Figure 1: The Impulse memory architecture. The arrows indicate how data flows within an Impulse memory system.

- a small number of *control registers*, which are split into a set of *Shadow Descriptors (SDescs)* and store configuration information for remapped shadow regions,
- a simple *ALU unit (AddrCalc)*, which translates 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 holds data prefetched from DRAM;
- a *DRAM Scheduler*, which contains circuitry that orders and issues DRAM accesses;
- DRAM chips, which constitute 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 translates the shadow address into a set of pseudo-virtual addresses using the remapping information stored in the matching shadow descriptor (**e**). These pseudo-virtual addresses are translated into real physical addresses by the MTLB (**f**). The real physical addresses pass to the DRAM scheduler (**g**). The DRAM scheduler orders and issues the DRAM accesses (**h**) and sends data back to the memory controller (**i**). Finally, when a full cache line has been gathered, the MMC sends it to the system memory bus (**j**).

2.2 Remapping Algorithms

Currently, the address translation at the Impulse memory controller can take four forms, depending on how the MMC is used to access a particular data structure: *direct remapping*, *strided remapping*, *transpose remapping*, or *remapping through an indirection vector*.

- *Direct mapping* maps one contiguous cache line in shadow memory to one contiguous cache line in real physical memory. The pseudo-virtual address for the shadow address *saddr* is (*saddr* $-$ *ssaddr*), where *ssaddr* is the starting address (assigned by the OS) of the data structure's *shadow address space image*. Examples of using this mapping include recoloring physical pages without copying [2] and constructing superpages from non-contiguous physical pages without copying [11].
- *Strided mapping* creates dense cache lines from data items that are not contiguous but stridedly distributed in virtual memory. The MMC maps a cache line addressed by the shadow address *saddr* to multiple pseudo-virtual addresses: (*stride* $\times$ *(saddr* $-$ *ssaddr)* / *size_of_data_item* + *stride* $\times$ *i*), where *i* ranges from 0 to (*cache_line_size* / *size_of_data_item* -1). This mapping can be used to create tiles of a dense matrix without copying or to compact strided array elements [2].
- *Transpose mapping* creates the transpose of a two-dimensional matrix by mapping the element $[j][i]$ of the transposed matrix to the element $[i][j]$ of the original matrix. This mapping can be used wherever a matrix is accessed in a major different with what it is stored [14].
- *Remapping through an indirection vector* packs dense cache lines from array elements according to an indirection vector. To remap the shadow address *saddr*, the MMC first computes its offset in shadow memory as *soffset* = *(saddr* $-$ *ssaddr)* / *size_of_array_element*, then uses the indirection vector *vector* to map the cache line addressed by the shadow address *saddr* to several pseudo-virtual addresses (*vector*$[soffset+i]$), where *i* ranges from 0 to (*cache_line_size* / *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 to use it to optimize the sparse matrix-vector product algorithm [2].

In direct mapping, each shadow address generates exactly one DRAM access. In other three mappings, each shadow address generates (*cache_line_size* / *size_of_data_item*) DRAM accesses if (*cache_line_size* $>$ *size_of_data_item*), or one DRAM access if (*cache_line_size* $\leq$ *size_of_data_item*).

3 DRAM Basics

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 is 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 is 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. In addition, DRAM has to be refreshed about 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, which allow a read or a write command to 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 the hot row, so it does not need a RAS signal. **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. The write transaction introduces two restrictions. First, a delay ($tDPL$) must be satisfied from the start of the last write cycle to the start of the precharge command. 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 SDRAM [7]. Their meanings and typical values in SDRAM clock cycles are presented in Table 1, which assumes a 147MHz clock rate.

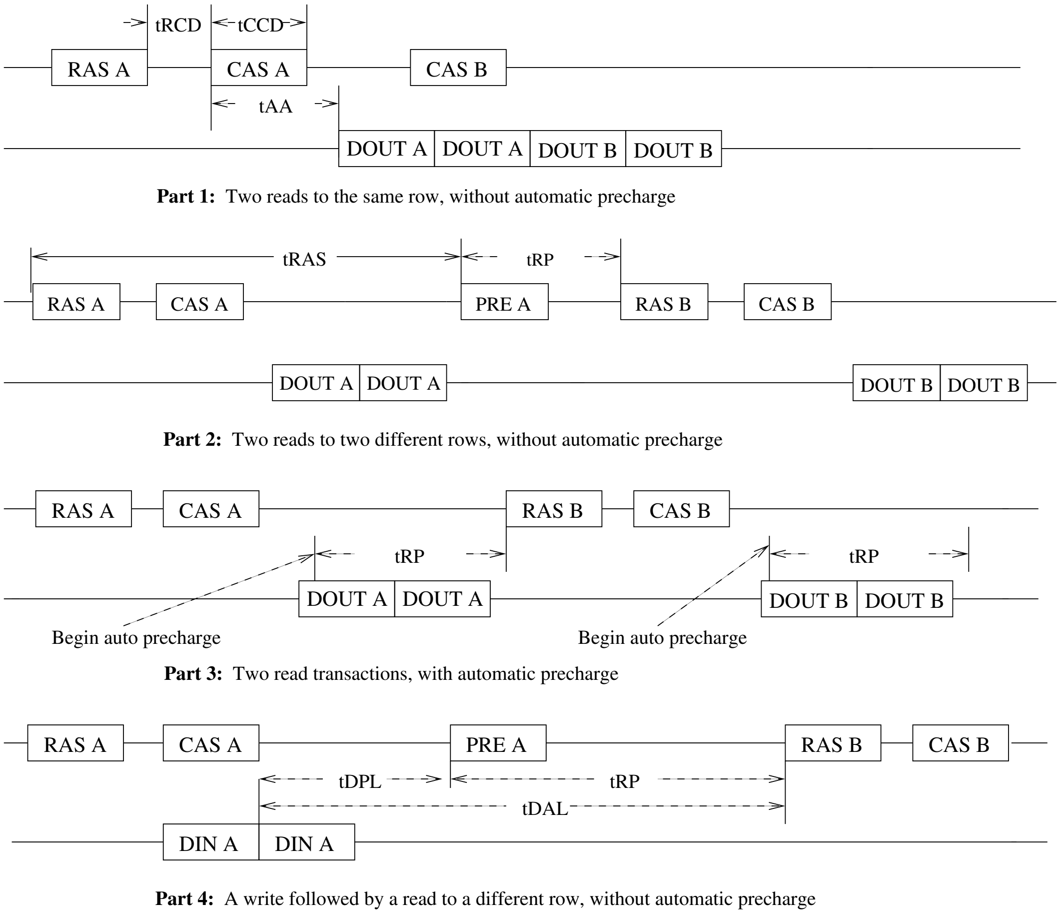


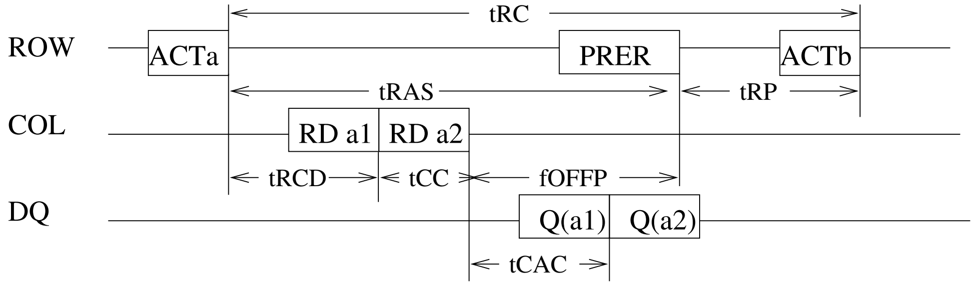
Figure 2: Examples of SDRAM transactions

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

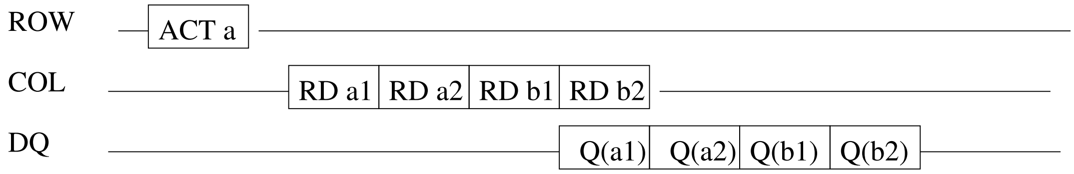
Table 1: Important timing parameters of Synchronous DRAM.

3.2 Direct Rambus DRAM

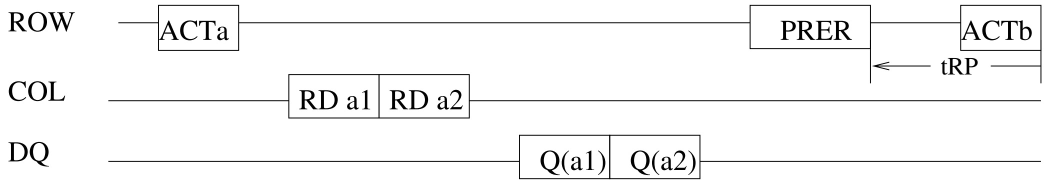
Direct Rambus DRAM is a high speed DRAM developed by Rambus, Inc [3]. RDRAM has independent pins for row address, column address, and data. Each bank can be independently opened, accessed, and precharged. Data and control information are transferred to and from RDRAM in a packet-oriented protocol. Each of the packets consists of a burst of eight bits over the corresponding signal lines of the channel.



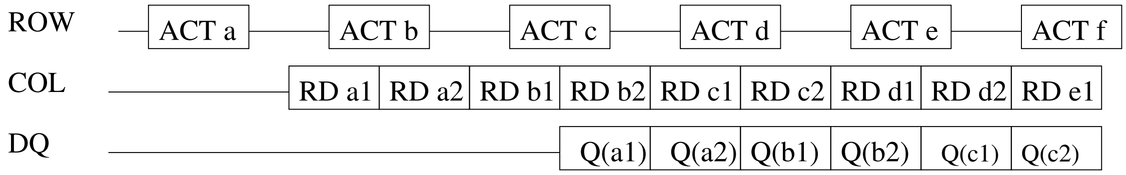
Part 1: A read transaction with precharge followed by another read.



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

Figure 3 shows some RDRAM transactions that all access the same chip. **Part 1** of Figure 3 shows a read transaction with a precharge command, followed by another transaction to the same bank. **Part 2** of Figure 3 shows the overlapping of two read transactions directed to the same row. **Part 3** of Figure 3 shows a read transaction without a precharge command followed by a transaction to a different row. In this case, the hot row must be explicitly precharged before the second transaction starts. **Part 4** 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 and their typical values in clock cycles are presented in Table 2, which assumes a 400MHz clock rate [6].

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.

4 Design

The proposed Impulse DRAM backend[1] contains three major components: the DRAM Dispatcher, Slave Memory Controllers (SMCs), and plug-in memory modules — DRAM chips. The DRAM dispatcher, SMCs, and RAM Address busses (RA busses) connecting them constitute the DRAM scheduler shown in Figure 1. A DRAM backend contains one DRAM dispatcher, but can have multiple SMCs, multiple RA busses, and multiple plug-in memory modules. Figure 4 shows a configuration that has four SMCs, four DRAM chips, eight banks, and two RA busses. Note that the DRAM dispatcher and SMCs do not 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 [13] (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 a shadow access or a normal access. The MMC sends a DRAM request to the backend via Slave Address busses (SA busses) and passes data from or to the DRAM backend via Slave Data busses (SD busses). During our experiments, we vary the number of SA busses/SD busses from one to one plus the number of shadow descriptors. If there is only one SA or SD bus, normal accesses and shadow accesses will share it. If there are two SA or SD busses, normal accesses will use one exclusively and shadow accesses will use the other one exclusively. If there are more than two SA or SD busses, one will be exclusively used by normal accesses and each of the rests will be used by a subset of 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 our experiments is to find out how many SA/SD busses are needed to avoid heavy contention on them. The experimental results will show that more than one SA or SD bus does not give significant benefit over single SA or SD bus.

[1]Since the proposed memory system was modeled based on the HP Kitt-Hawk memory system [4], this document uses the terminology of Kitt-Hawk memory system.

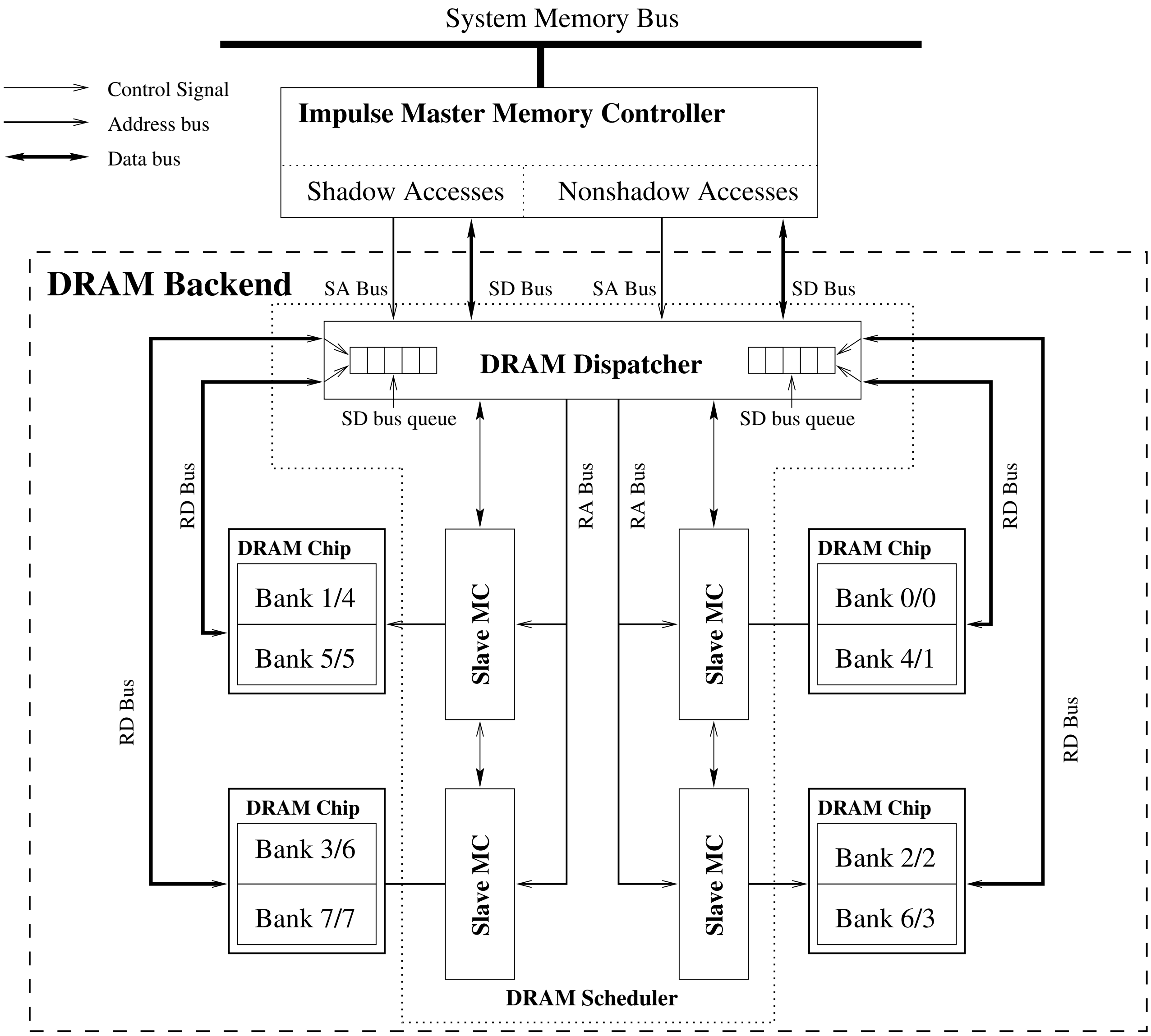


Figure 4: Impulse DRAM Backend Block Diagram

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 these contentions by picking up winners according to a designated algorithm and queuing the others. If a waiting queues becomes critically full, the DRAM dispatcher stops the MMC from sending more requests. All waiting queues work in First-Come-First-Serve (FSFC) order.

4.2 Slave Memory Controller

Each slave memory controller controls one RD bus and DRAM chips using the RD bus. The SMC has independent control signals for each DRAM chip. Each chip has multiple *memory bank*s. Each memory bank normally has its own page buffer and can be accessed independently from all other banks[2]. 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.

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

4.3 Others

This section describes algorithms implemented in the proposed 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 transactions 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

The collection of hot rows can be regarded as a cache. Proper management is necessary to make this "cache" profitable. The Impulse DRAM backend allows hot rows to remain active after being accessed. The benefit of leaving a row open is eliminating the RAS signals for accesses that hit the row. However, a DRAM access has to pay the penalty of closing the row if it misses the row. We test three hot row policies: *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 speculate whether the next access will hit or miss an open row.

[2]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.

The *use-predictor* policy was initially designed by R.C. Schumann [9]. 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 requires 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. Applications 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 experiments, the precharge policy register is set "open" when there are more hits than misses in the history and "close" otherwise. 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 perform speculation. If the next transaction accesses the same row as the current transaction, the row is left open. Otherwise, the row is closed.

4.3.2 Access reordering

The Impulse MMC can several types of DRAM accesses 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 the following four types.

- *Direct accesses* are normal accesses directly coming from the system memory bus (arrow **b** in Figure 1). As in conventional systems, each direct access requests a cache line from the DRAM backend.
- *Indirection vector accesses* fetch the indirection vector for *remapping through an indirection vector* (**k**). Each indirection vector access is for a cache line and the return data is sent to the relevant shadow descriptor.
- *MTLB accesses* are generated by the MTLB to fetch page table entries from DRAM 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 accesses* are generated by the Impulse memory controller to fetch remapped data (**g**). The size of each shadow access varies with application-specific mappings. The return data of shadow accesses is sent to the remapping controller for further processing.

For convenience, we use *non-shadow accesses* to represent direct accesses, indirection vector accesses, and MTLB accesses. Normally, most of DRAM accesses are either direct accesses or shadow accesses, with a few being MTLB accesses and indirection vector accesses. Intuitively, different types should be treated differently in order to reduce the average memory latency. For example, an indirection vector access is depended upon 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 is 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 indirection vector accesses and MTLB accesses. After having taken consider these facts, we propose a reordering algorithm with the following rules.

1. No reordering shall violate data consistency — read after write, write after read, and write after write.
2. Once an access is used to predict 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 is guaranteed to access the relevant memory bank right after the preceding one.
3. Non-prefetching accesses have higher priority than prefetching accesses.
4. MTLB accesses and indirection vector accesses have higher priority than others.
5. Since it is hard to determine whether direct accesses or shadow accesses should be given higher priority over each other, we study two opposite choices: giving direct accesses higher priority or giving shadow accesses higher priority.
6. The priority of an access increases as its waiting time increases. This rule guarantees no access would stay in a waiting queue "forever". We make this rule optional so that we can find out if it is useful and if some accesses will starve without this rule.
7. If there is a conflict among different rules, always use the first rule of the conflicting ones according to the sequence above.

4.3.3 Interleaving

The interleaving scheme determines the mapping from physical DRAM addresses to memory banks. Interleaving can be either page-level if the mapping granularity is the size of the page buffer or cache-line-level if the mapping granularity is the size of an L2 cache line. Page-level interleaving 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. We consider two interleaving schemes, as shown by Figure 4: *modulo-interleaving*, which maps consecutive physical addresses to different DRAM chips; and *sequential-interleaving*, which maps consecutive physical addresses to banks inside the same chip.

5 Experimental Framework

5.1 Simulation Environment

We extended the executive-driven simulator Paint [10, 12] 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 unified I/D TLB is single-cycle, and fully associative, uses a not-recently-used replacement policy, and has 120 entries.

The simulated Impulse memory controller is derived from the HP memory controller used in servers and high-end workstations [5]. We model seven shadow descriptors, each of which is associated with four 128-byte lines. The controller can prefetch the corresponding shadow data into these fully associative lines. A 4Kbyte 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, and uses two 128-byte buffers to prefetch consecutive lines of page table entries.

In the simulated system, the CPU runs at the clock rate of 400MHz. The SDRAM works at 147MHz. Each SDRAM chip contains two banks, each of which has a 16Kbyte page-buffer. The RDRAM works at 400MHz. Each RDRAM chip contains 8 banks, each of which has an 8Kbyte page-buffer. The DRAM width is 16 bytes, so are the RD busses and SD busses. An SA bus can transfer a request per cycle.

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

We use the following benchmarks to test the performance of the proposed Impulse DRAM backend.

- **CG** from NPB2.3 [1] 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 $A \times P$, where A is the sparse matrix and P is the vector. Two Impulse optimizations can be applied to this benchmark independently: remapping through an indirection vector (call it **CG.iv** in the following discussion) and direct mapping. Remapping 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 (call it **CG.pc3**); another one remaps all the seven major data structures in CG (call it **CG.pc7**).

- **Spark98** [8] 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 remapping through an 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 restriction that the size of a 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 $C = A \times B$, where A, B, and C are dense matrices. Impulse remaps a tile of each matrix into a contiguous space in shadow memory using *strided mapping*. Impulse divides 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 L2 cache during the computation of the relevant row of the C matrix.

- **Rotation** rotates an image by performing three one-dimensional shears. The second shear operation walks along the column of the image matrices (assuming the image arrays are stored along the x axis), which gives poor memory performance for large images. Impulse can transpose matrices without copying it, so column walks are replaced by row walks. 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 the original column walks are replaced by the row walks over the shadow matrices. The algorithm involves three matrices. All of them are remapped using *transpose remapping* during the second phase.

Remapping improves all of these benchmarks significantly. Where the benefits come from and how much improvement each benchmark has is beyond the scope of this document. Instead, this document focuses on how the DRAM backend impacts the performance. These benchmarks have different ways to use remappings. 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 the benchmark. "Gathering Factor" indicates the number of DRAM accesses needed to compact a shadow cache line.

5.3 Methodology

The design options that we care about include the interleaving scheme, the 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. Each option

Benchmarks	Problem Size	Remapping	Descriptors	Gathering Factor
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.

can have multiple values. It is infeasible to use the full factorial design, so we pick up a practicable subset of these options at a time to study their relative impacts.

In our experiments, we first set up a baseline and then vary one or a few options at a time. The baseline uses cache-line modulo-interleaving, no access reordering, 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

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

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

The data transferring cycles on SA/SD/RD busses are inevitable and cannot be reduced. The bank access cycles are inevitable too, but they can be reduced by an appropriate hot row policy. The waiting cycles on SA/SD/RD bus or DRAM bank are extra overhead and should be avoided as much as possible. Reducing the waiting cycles is the main goal of the proposed DRAM backend.

6.1 The Impacts of DRAM Organization

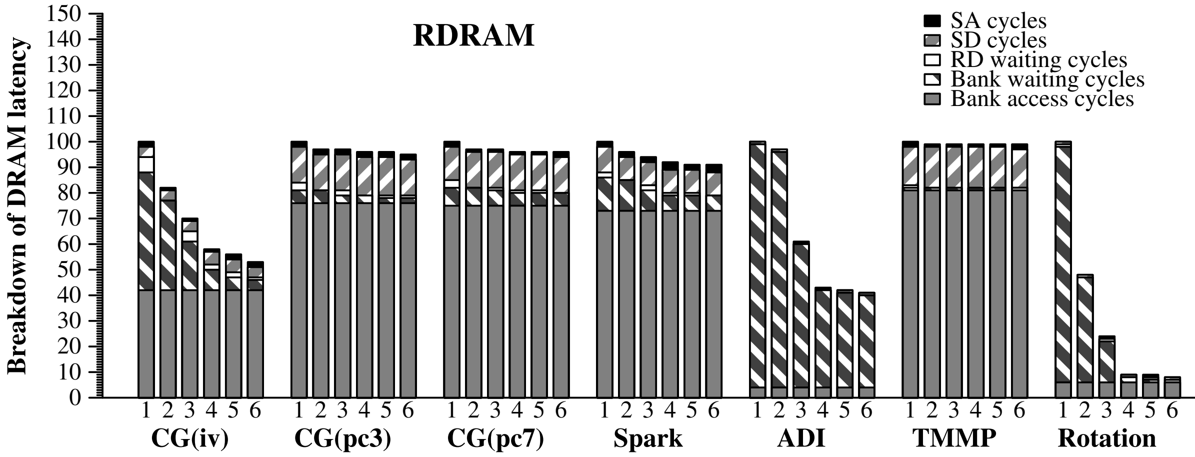


Figure 5: Breakdown of the average RDRAM access latency for various DRAM organizations: organization **1** – 32/2/1 (the number of memory banks/RD busses/SD bus queues); **2** – 32/4/2; **3** – 64/4/2; **4** – 128/8/4; **5** – 256/8/4; **6** – 256/16/8.

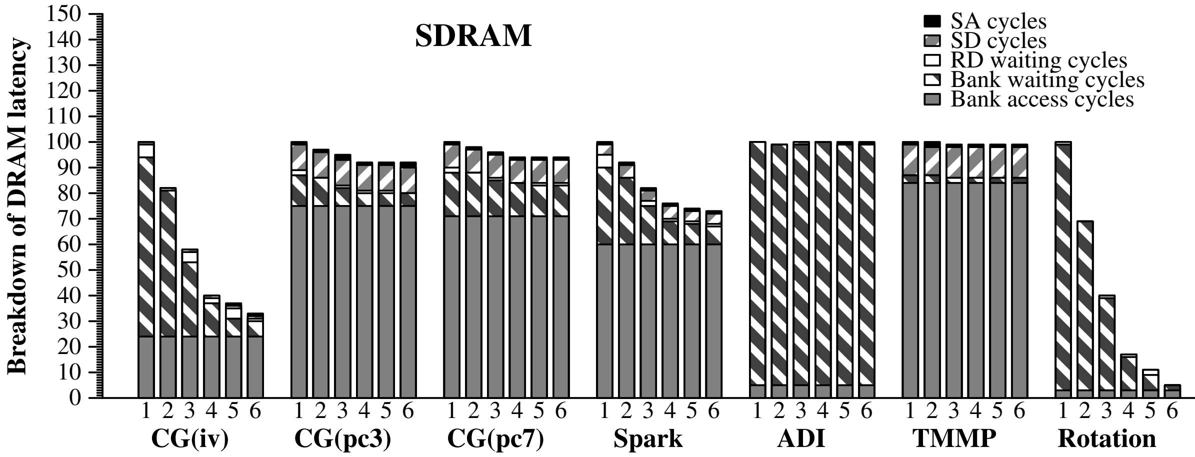


Figure 6: Breakdown of the average SDRAM access latency for various DRAM organizations: organization **1** – 8/2/1 (the number of memory banks/RD busses/SD bus queues); **2** – 8/4/2; **3** – 16/4/2; **4** – 32/8/4; **5** – 64/8/4; **6** – 64/16/8.

The number of memory banks, RD busses, and SD bus queues are tightly related to one another, so we consider them together as one compound factor — DRAM organization. They can affect the bank waiting cycles, RD waiting cycles, and SD cycles, but cannot 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 of each benchmark in six different

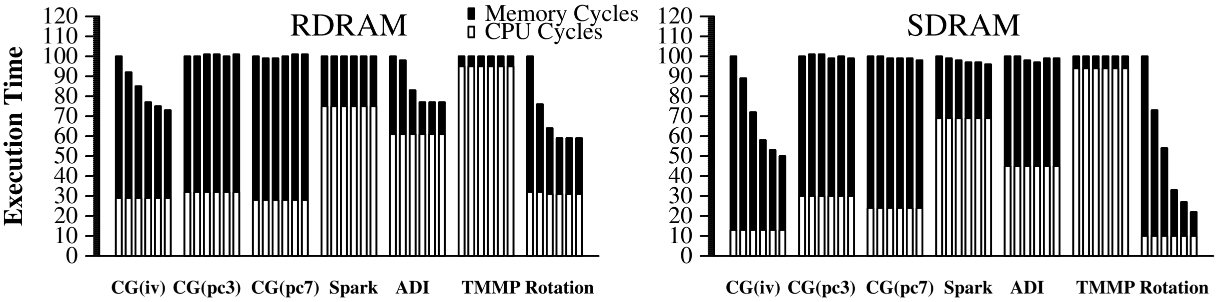


Figure 7: Execution times with various DRAM organizations.

DRAM organizations based upon RDRAM. Figure 6 shows the corresponding results for SDRAM. Figure 7 shows their execution times. All the results shown in this document are normalized to the baseline's, except stated otherwise. We split execution time into two parts: *memory cycles* and *CPU cycles*. The DRAM backend is targeted to attack the memory cycles only, so the CPU cycles remains constant when the DRAM backend changes. Because changing the DRAM organization does not change the bank access cycles, the performance of **CG.pc3**, **CG.pc7**, **Spark**, and **TMMP**, where the bank access cycles dominate the DRAM access latency, changes insignificantly. However, the performance of **CG.iv**, **ADI**, and **Rotation**, where the bank waiting cycles are the dominant factor of the DRAM access latency, changes significantly. In particular, increasing the number of memory banks, RD busses, and SD bus queues achieves nearly linear improvement in the bank waiting cycles. For example, comparing configuration **1** with **6** in Figure 6, the average DRAM access latency of **Rotation** decreases from 2123 cycles (2067 of which are bank waiting cycles) to 109 cycles (52 of which are bank waiting cycles), which results in a 87% saving in the memory cycles and a 78% saving in execution time.

All other benchmarks, except **ADI**, have similar results on RDRAM and SDRAM. When the DRAM organization expands, **ADI** gains much more improvement on RDRAM than on SDRAM. It is because SDRAM, which normally has fewer banks than same capacity RDRAM, cannot provide the minimum number of memory banks that **ADI** needs to avoid long bank waiting queues. For instance, even with configuration **6**, the average length of a bank waiting queue is 13.6 transactions for SDRAM, but only 1.9 transactions for RDRAM.

Intuitively, 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. Specifically, each RD bus or SD bus queue can serve only a certain number of memory banks. Letting it serve fewer banks wastes its capacity, and letting it serve more banks makes it become the bottleneck. Figure 8 shows the performance of **CG.iv** and **Rotation** under various DRAM organizations based on SDRAM. We can make two observations from Figure 8. First, whenever the ratio between the number of memory banks and the number of RD busses drops from a higher value to 2:1, there is a big drop in the average DRAM access latency. This observation 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 can rephrase 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

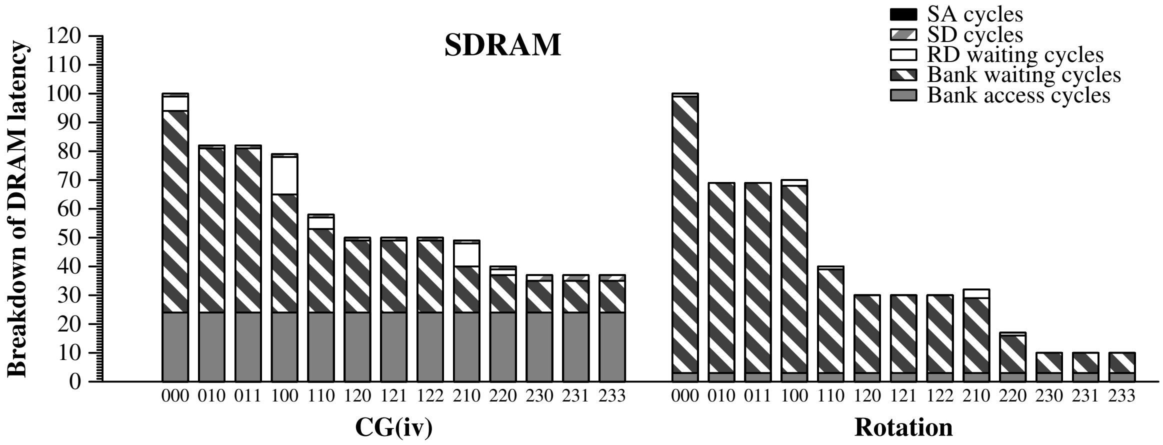


Figure 8: Breakdown of the average SDRAM access latency of **CG** and **Rotation**. Each bar is labeled as **ABC**, where **A** represents the number of memory banks: 0 – 8, 1 – 16, 2 – 32; **B** represents the number of RD busses: 0 – 2, 1 – 4, 2 – 8, 3 – 16; **C** represents the number of SD bus queues: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

bus queues does not increase performance noticeably. The observations indicate that one SD bus queue can match up with at least 16 RD busses.

Conclusion: a cost-effective, balanced DRAM backend should have one RD bus for each DRAM chip and one SD bus queue for every 16 RD busses.

6.2 The Impacts of Slave Busses

We consider three alternative configurations about slave busses: one SA bus and one SD bus; two SA busses and two SD busses; and eight SA busses and eight SD busses. The corresponding results are shown in Figure 9, 10, and 11.

	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 baselines.

None of those benchmarks wastes significant amount of time waiting for slave 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. If we look at Figure 5 and 6, we also can see that the SA cycles and the SD cycles are almost identical for any configurations. These results indicate that slave bus is normally not busier than other components of the DRAM backend.

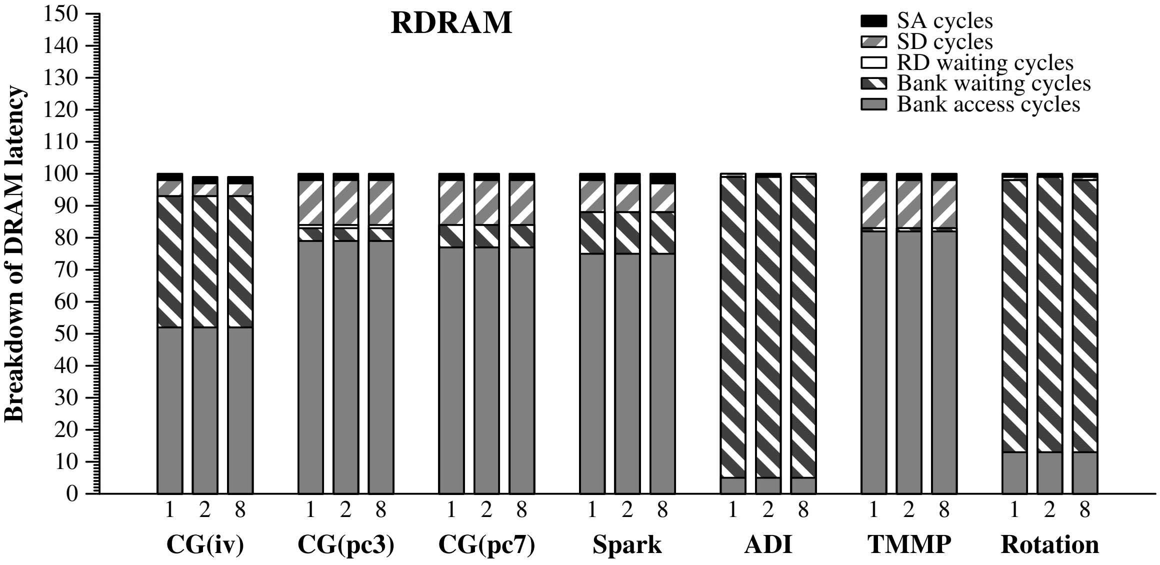


Figure 9: Breakdown of the average RDRAM access latency when the number of slave busses varies: **1** – 1 SA bus, 1 SD bus; **2** – 2 SA busses, 2 SD busses; **8** – 8 SA busses, 8 SD busses.

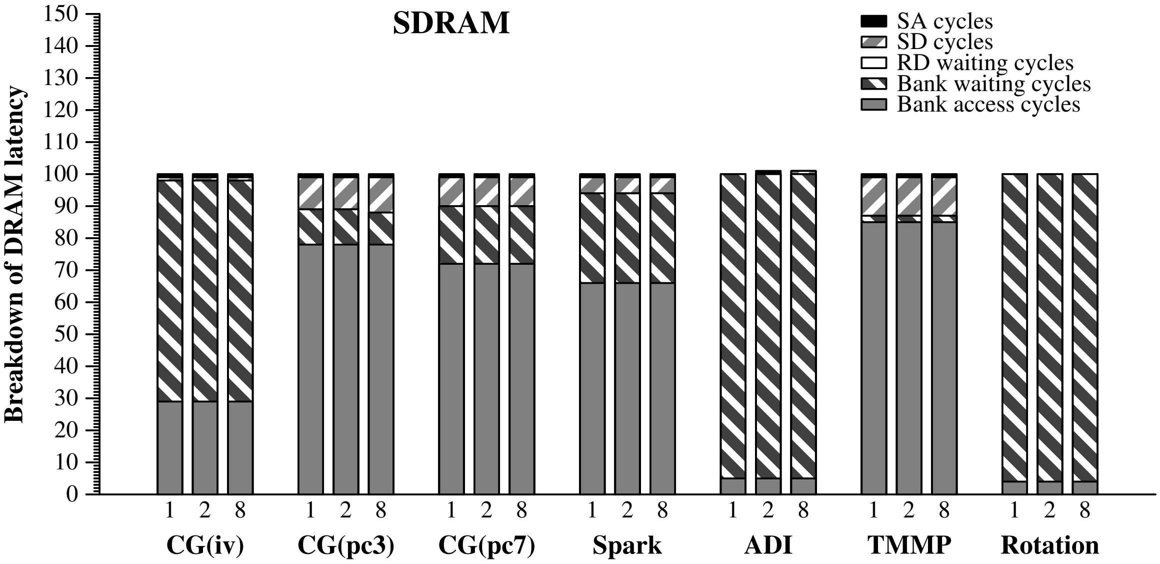


Figure 10: Breakdown of the average SDRAM access latency when the number of slave busses varies: **1** – 1 SA bus, 1 SD bus; **2** – 2 SA busses, 2 SD busses; **8** – 8 SA busses, 8 SD busses.

Conclusion: one SA bus and one SD bus are enough for the proposed Impulse DRAM backend.

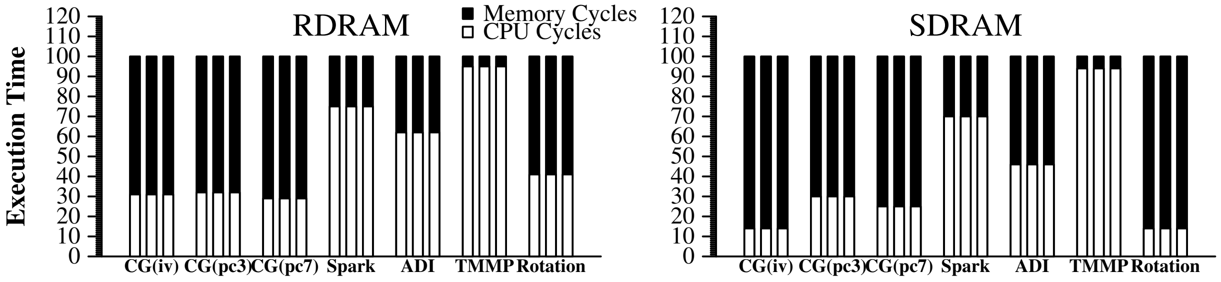


Figure 11: Execution times with various numbers of slave busses

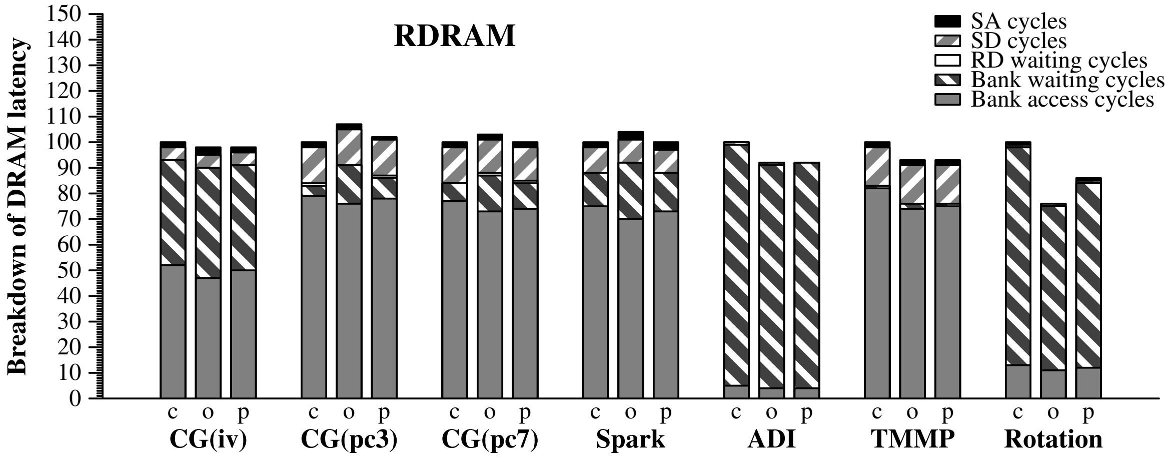


Figure 12: Breakdown of the average RDRAM access latency for three hot row policies: **c** – close-page; **o** – open-page; **p** – use-predictor.

6.3 The Impacts of Hot Row Policy

Figure 12 and 13 display the results of three different hot-row policies: 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 direct effect of a hot row policy is to reduce the bank access cycles. The indirect effect is to reduce the bank waiting cycles. Table 5 displays the hot row hit and miss ratios under the open-page policy and use-predictor policy. The hit ratio is computed as (total hot row hits / total accesses). The miss ratio is computed as (total hot row misses / total accesses). If a transaction accesses a bank without an active hot row, it is taken neither as a hit nor as a miss. In addition, hot row has to be closed during refresh operations, so the hit ratio plus miss ratio for the open-page policy may not equal 100%.

To better understand the tradeoffs involved, let's first look back 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 saving $tRCD + tOFFP +$

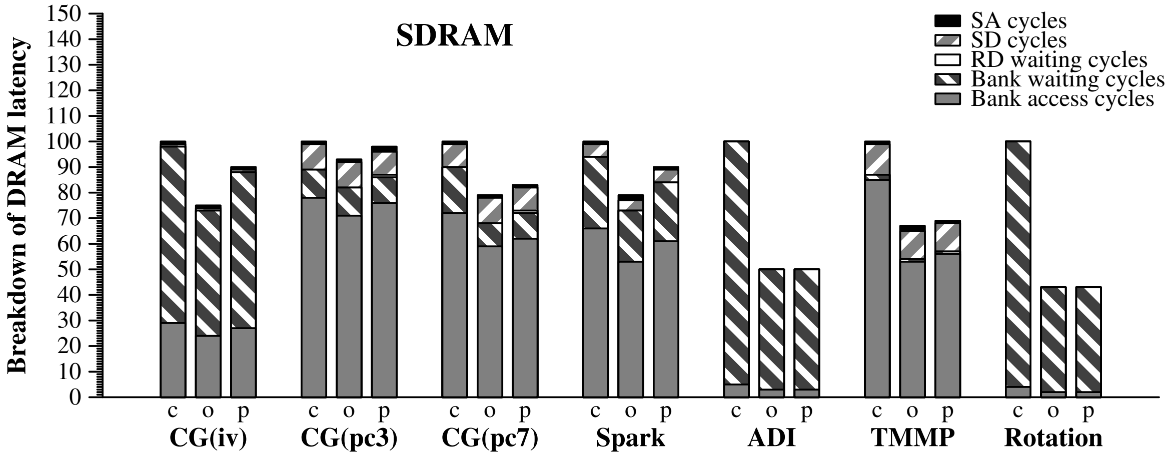


Figure 13: Breakdown of the average SDRAM access latency for three hot row policies: **c** – close-page; **o** – open-page; **p** – use-predictor.

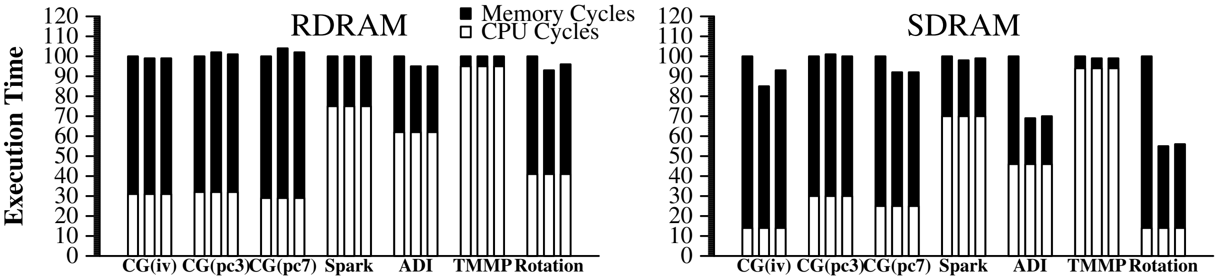


Figure 14: Execution times with three hot row policies.

	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.

$tRP - tCC = 14$ cycles for (read, read)[3], or $14 - tBUB1 = 10$ cycles for (read, write), or $14 - tBUB2 = 6$ cycles for (write, X); and the miss penalty is adding $tRP = 8$ 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.

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

The impact of a hot row policy on a special run can be approximated using the following expression:

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

In order to achieve positive impact, Equation 1 must be greater than 0, i.e., the ratio between total hits and total misses must be greater than (average-miss-penalty / average-hit-benefit). Therefore, we use the ratio between total hits and total misses as the leverage to measure a hot row policy's performance.

The theoretical results obtained using Equation 1 match with the experimental results almost perfectly, with only a couple of exceptions. For the open-page policy, **CG.iv** and **CG.pc7** have the same hit/miss ratios, but different average DRAM access latency. It is because of 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. In RDRAM, whenever a write transaction is involved, the hit benefit decreases from 14 cycles to 10 or 6 cycles. Though **CG.pc7** has the same hot row hit/miss ratios as **CG.iv**, 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 and use-predictor policy. Most of the DRAM accesses in **Rotation** are shadow accesses requesting eight-byte double words, so the average miss penalty is close to the average CAS count — one. With one-cycle miss penalty and three-cycle hit benefit, the 11% difference in the miss ratio cannot make a big difference.

The use-predictor policy always has performance between the close-page policy's and open-page policy's. Wherever the open-page policy is helpful, the use-predictor policy is also helpful, but at a smaller degree. Wherever the open-page policy hurts the performance, the use-predictor policy also hurts the performance, but at 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 has positive gains in most test benchmarks.

Conclusion: both the open-page policy and use-predictor policy are acceptable choices. We suggest the use-predictor policy because it is stabler than the open-page policy.

6.4 The Impacts of Access Reordering

We have considered six different access reordering algorithms. The first one is no reordering (number it as No.1). The other five are the alternatives of the algorithm described in section 4.3.2. No.2 gives direct access higher priority over shadow access and has no priority updating. No.3 gives shadow access higher priority over direct access and has no priority updating. No.4 gives shadow access and direct access the same priority and has no priority updating. No.5 is No.2 plus priority updating, which increases priority along with increased waiting time. No.6 is No.3 plus priority updating.

The highest priority in the simulated model is 15. The priority vector, in 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; $\{15, 15, 9, 11, 7, 7, 1, 3\}$ 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 is overtaken by another access. For example, in No.2, one non-prefetching shadow access, which starts with priority 9, may be overtaken by accesses with higher priority for at most six times. Once it has given away its position for six times, it will have the highest priority and no accesses after it can get ahead of it.

Reordering algorithms try to reduce the *bank access cycles* by using the first transaction on the waiting queue to make correct predictions about whether or not to leave a hot row open after an access.

The basic premise to use reordering is that a waiting queue must contain more than one transaction. If a waiting queue contains none or one transaction most of the time, reordering cannot do anything because there is nothing to be reordered. Table 6 lists the average queue length in the baseline execution of each benchmark. It shows that only **ADI** and **Rotation** have long waiting queues. Consequently, they are the only benchmarks noticeably impacted by the reordering algorithm.

	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.

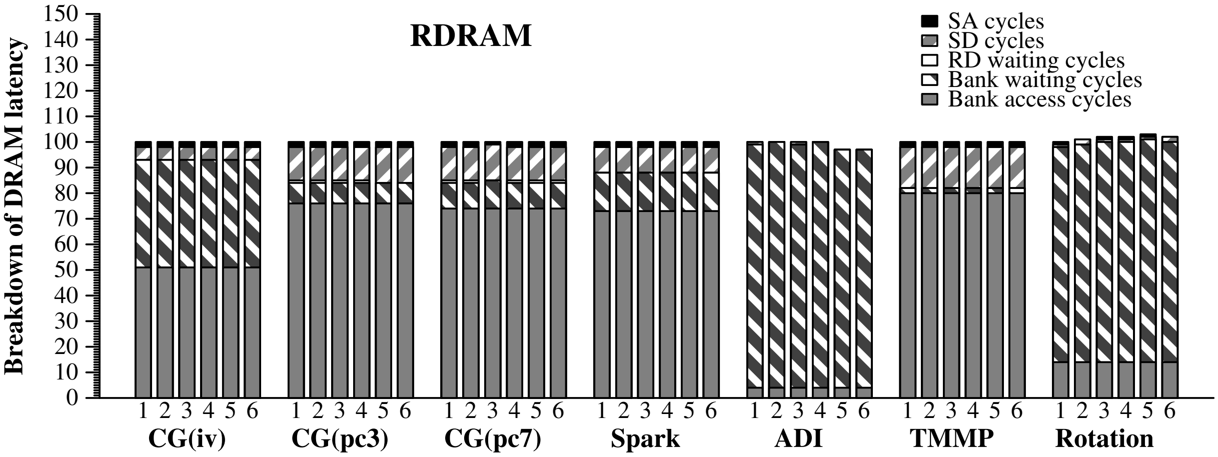


Figure 15: Breakdown of the average RDRAM access latency for various reordering algorithms.

Figure 12, 13, and 14 show the performance numbers for various reordering algorithms. They show that No.2, No.3, and No.4 perform closely on all benchmarks. Taking a closer look at DRAM access patterns of these benchmarks, we found either most of DRAM accesses are direct accesses or most of DRAM accesses are shadow accesses at a short period of time, which results in every few reorders between direct accesses and shadow accesses. These results indicate that we can just give direct access and shadow access the same priority to simplify the reordering algorithm. These figures also show that the updating rule does not help very much. Although it gives a 3% reduction in the average DRAM access latency for **ADI** on RDRAM, it contributes a 19% increase in the average DRAM access latency for **Rotation**. For **Rotation**, the updating rule decreases the average DRAM access latency of a prefetching shadow access from 1350 cycles to 1173

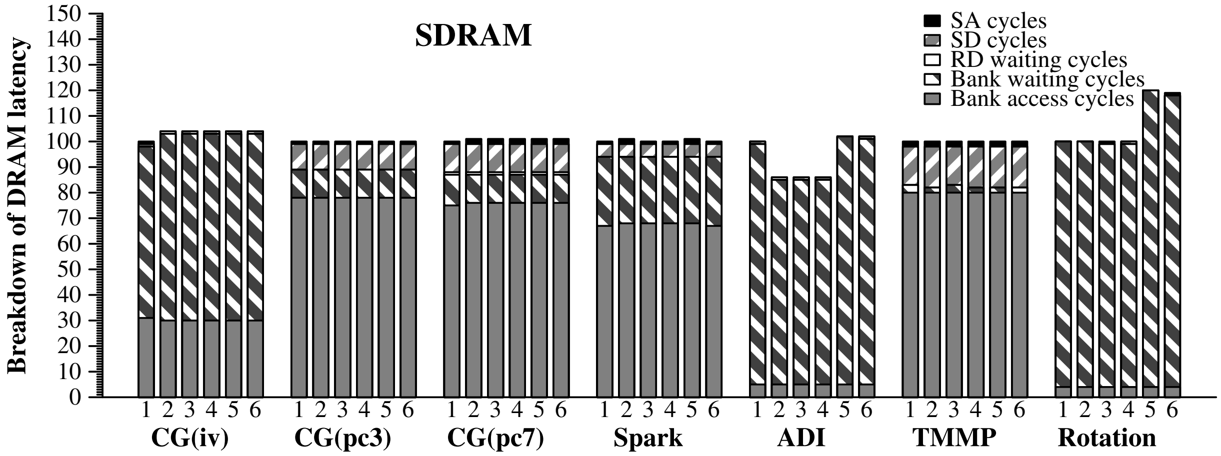


Figure 16: Breakdown of the average SDRAM access latency for various reordering algorithms.

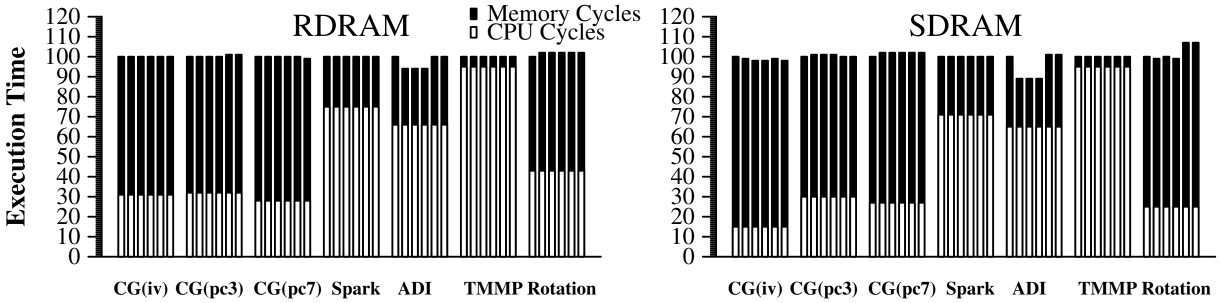


Figure 17: Execution times with various reordering algorithms.

cycles, but it increases the average DRAM access cycles of a non-prefetching shadow access from 338 cycles to 596 cycles.

Conclusion: the best choice is No.4, which gives direct accesses and shadow accesses the same priority and does not update priority dynamically.

6.5 The Impacts of Interleaving

We consider four interleaving schemes, as described in section 4.3.3. Figure 18 and 19 show the breakdown of the average DRAM access latency. Figure 20 shows the execution times. How well an interleaving scheme can perform heavily depends on applications' access patterns. There is no optimal scheme working for all benchmarks. For example, for applications that perform sequential accesses, cache-line-level interleaving is better; for applications that perform strided accesses, page-level interleaving may be better. However, modulo-interleaving is always better than sequential-interleaving. Sequential-interleaving is bad because it cannot exploit spatial data locality exhibited by most applications. It hurts performance by di-

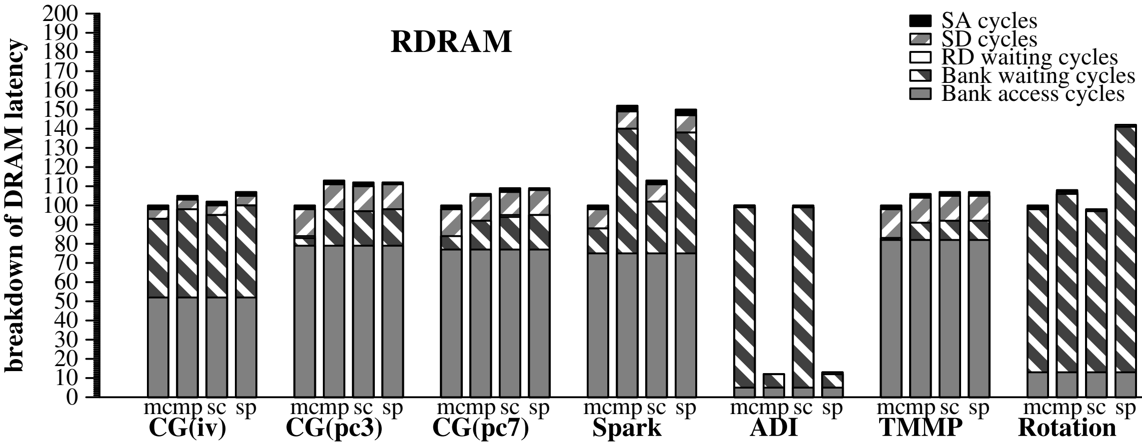


Figure 18: Breakdown of the average RDRAM access latency for various interleaving schemes: **mp** – modulo, page-level; **mc** – modulo, cache-line-level; **sp** – sequential, page level; **sc** – sequential, cache-line-level.

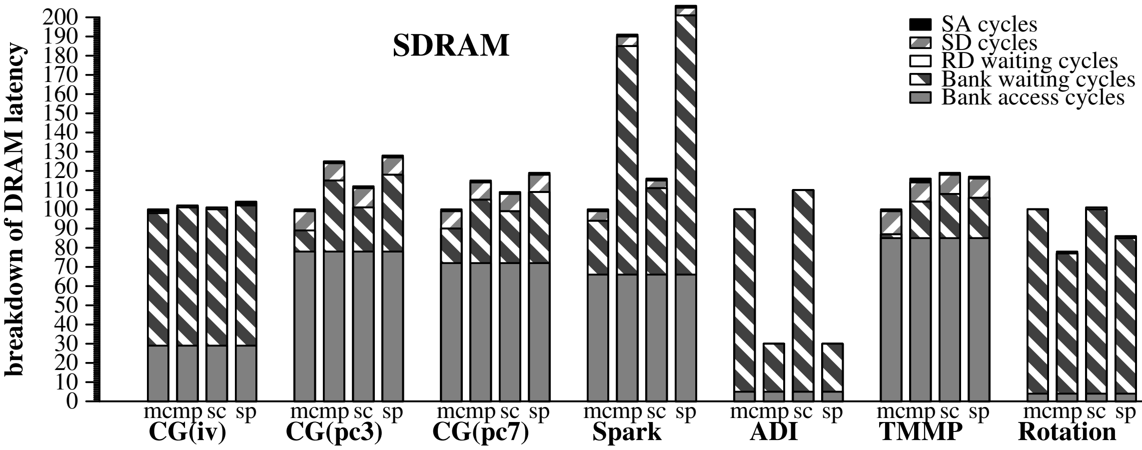


Figure 19: Breakdown of the average SDRAM access latency for various interleaving schemes: **mp** – modulo, page-level; **mc** – modulo, cache-line-level; **sp** – sequential, page level; **sc** – sequential, cache-line-level.

recting consecutive accesses to the same chip, which limits the inherent parallelism of DRAM accesses. Spatial locality also requires cache-line-level interleaving to ensure that consecutive requests go to different memory banks. That is why page-level interleaving cannot 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 are likely directed to the same page. Page-level interleaving makes the four DRAM accesses go to the same bank and be served serially instead of in parallel. The results show that, compared to cache-line-level interleaving, page-level interleaving increases the average DRAM access latency of **Spark** by 52% on RDRAM and 91% on SDRAM.

When **ADI** walks along a column of a 1024x1024 double array, it generates access sequence x, $x + 8K$,

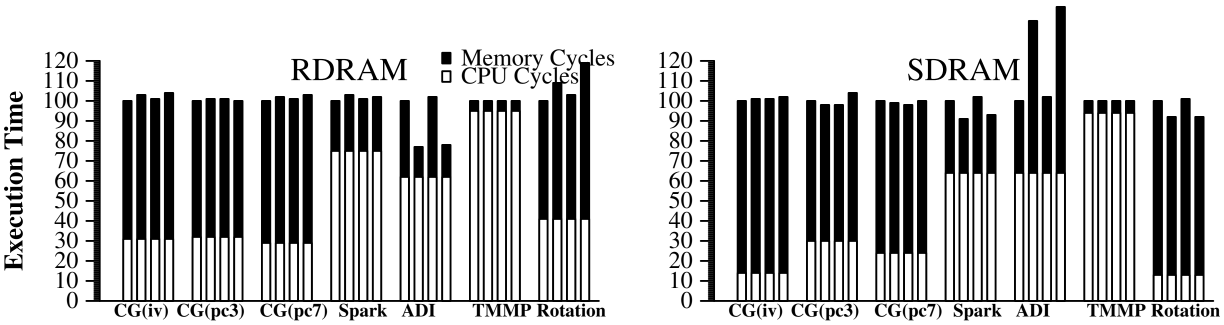


Figure 20: Execution times with various interleaving schemes.

$x + 2 \times 8K, \ldots, x + 1023 \times 8K$. If interleaving is in cache-line-level, all accesses go to the same bank. If the interleaving is in page-level, the *i*th access ($x + i \times 8K$) goes to bank (i % 32) if RDRAM is used, or bank ((i/2) % 8) if SDRAM is used. In another word, all accesses go to the same bank in cache-line-level interleaving, but they are uniformly distributed among all banks in page-level interleaving. As a result, page-level interleaving performs a lot better than cache-line-level interleaving for **ADI** – about 30% saving in execution time.

Rotation operates on a 1024x1024 gray-scale image. Walking along a column of the image generates access sequence x, $x + 1K$, $x + 2 \times 1K, \ldots, x + 1023 \times 1K$. When SDRAM is used, these accesses all go to the same bank if 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 page-level interleaving is used. Therefore, page-level interleaving is better than cache-line-level interleaving for **Rotation** when SDRAM is used. When RDRAM is used, the *i*th access goes to bank (i % 4) if cache-line-level interleaving is used, or the first eight accesses go to bank 0, the next eight go to bank 1, …, and so on if page-level interleaving is used. Though page-level interleaving better distributes accesses among banks, cache-line-level achieves better performance because it puts consecutive accesses to different banks.

Conclusion: cache-line-level modulo-interleaving is the best choice.

6.6 Putting It All Together

Based on the experimental results above, we propose a DRAM backend with this configuration: one RD bus for each DRAM chip, one SA bus, one SD bus, one SD bus queue, use-predictor hot row policy, No.4 reordering algorithm, and cache-line-level modulo-interleaving. Since cache-line-level interleaving may significantly slow down applications that access data with page-sized strides, we pad an extra line to each such stride to avoid unbalanced loading in memory banks.

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 option (**w**); the configuration combined by the best setting of each option (**b**); and the recommended configuration (**r**). The recommended configuration

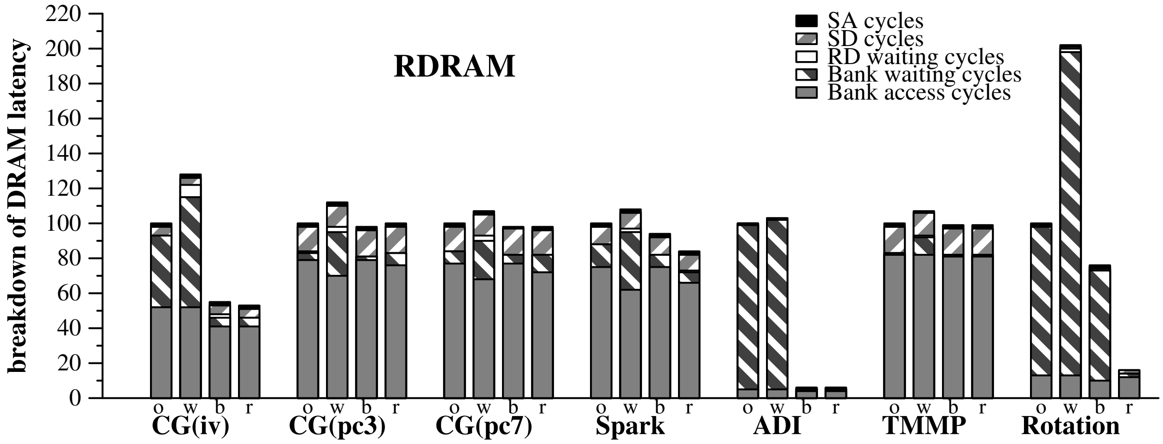


Figure 21: The average RDRAM access latency for various DRAM backends: **o** – original baseline; **w** – worst; **b** – best; **r** – recommended.

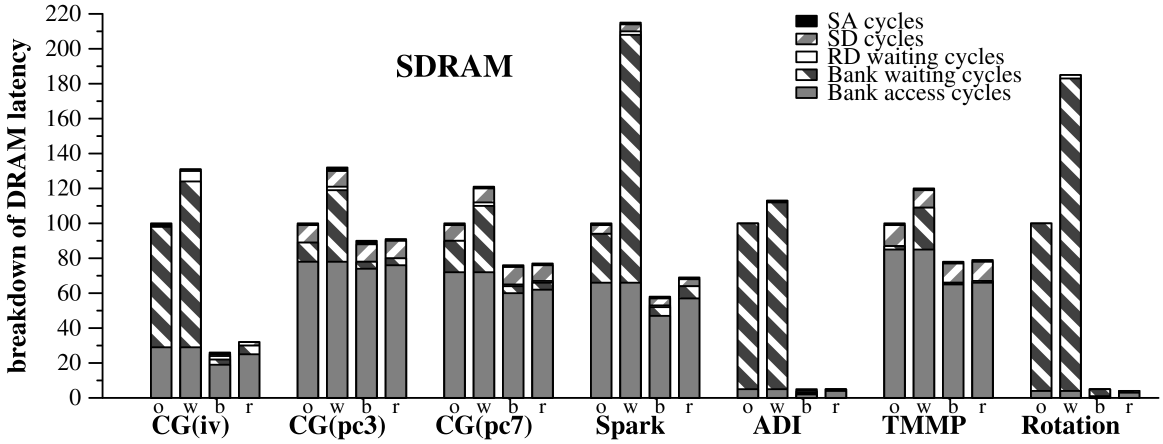


Figure 22: The average SDRAM access latency for various DRAM backends: **o** – original baseline; **w** – worst; **b** – best; **r** – recommended.

works well in all benchmarks. It performs closely to or even better than the combination of the "bests". The interaction among different factors makes the combination of the "bests" not always work best. Comparing the recommended one to the baseline, the saving in the DRAM access latency ranges from 1% to 94% with a geometric mean of 35% for RDRAM and from 9% to 96% with a geometric mean of 49% for SDRAM.

7 Conclusion and Future Work

The Impulse memory system exposes DRAM accesses not seen in conventional memory systems. In this document, we investigate whether or not it is worthwhile to redesign the conventional DRAM backend for Impulse and to quantify the degree that the DRAM backend can affect the performance. We do so by proposing a DRAM backend that can effectively exploit parallelism of DRAM accesses in the Impulse system.

We use the execution-driven simulator Paint to evaluate the performance of the proposed backend. The experimental results show that the proposed backend can reduce the average DRAM access latency by up to 98%, the average memory cycles by up to 90%, and the execution time by up to 80%. These results demonstrate that it is indeed necessary to design a specific DRAM backend for the Impulse memory system.

There are still a number of issues needed to be solved before we build a real backend. In the proposed design, there is a queue for each memory bank. One alternative is to use a queue for each DRAM chip, then perform bank-level scheduling. Another alternative is to use a global queue for all banks, then perform global scheduling. We can also extend the priority-based algorithm to include non-priority-based rules. For example, putting transactions accessing the same row together might be helpful. More interleaving schemes, such as double-word interleaving or combinations of modulo and sequential interleaving, are probable and remain to be exploited. More research on the use-predictor policy needs to be done to answer questions like how many bits in history are enough and what is the best value for the precharge policy register.

References

[1] 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.

[2] 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, FL USA, January 1999.

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

[4] Hewlett-Packard. Kitt Hawk Memory System, External Reference Specification, Revision B. Dwg No A-5180-7358-1 Rev A, May 1995.

[5] 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.

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

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

[8] 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.

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

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

[11] 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.

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

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

[14] L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee. Memory system support for imaging processing. In *Proceedings of the 1999 International Conference on Parallel Architecture and Compilation Techniques*, pages 98–107, Newport Beach, CA USA, October 1999.