ISIM: The Simulator for The Impulse Adaptable Memory System

Lixin Zhang

UUCS-99-017

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

September 18, 1998

Abstract

This document describes ISIM, the simulator for the Impulse Adaptable Memory System. Impulse adds two new features to a conventional memory system. First, it supports a configurable, extra level of address remapping at the memory controller. Second, it supports prefetching at the memory controller. Consequently, two new units, a remapping controller and a memory controller cache, are added to a traditional memory system to support the new Impulse features. ISIM is based on Paint, a PA-RISC instruction set interpreter. ISIM extends Paint with a detailed Impulse memory system model which includes a primary data cache, a secondary data cache, a system bus, an Impulse memory controller, and a renovated DRAM backend.

Note that this document focuses on the Impulse extensions only. The reader should consult the Paint technical report [2] for an overview of the Paint simulation environment and terminology.

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

This document describes the memory system modeled in ISIM — the simulator developed for the Impulse Adaptable Memory System Project. The goal of the Impulse project is to build an adaptable memory controller that can significantly increase the efficiency of the system memory bus and cache. The Impulse memory system adds two important features to a traditional memory system. First, Impulse supports application-specific optimizations through configurable physical address remapping. By remapping physical addresses, applications can control how their data is accessed and cached, thereby improving the cache and system memory bus utilization. Second, Impulse supports prefetching at the memory controller. By prefetching data at the memory controller, the Impulse memory system can hide the DRAM access latency. This document requires readers to know the basics of the Impulse project. For an overview of Impulse and its terminology, please read the appropriate Impulse document [1].

The PAINT [2] simulator has been extended to support the Impulse project. Paint interprets the PA-RISC 1.1 instruction set. It models a variation of a 120MHz, single-issue, HP PA-RISC 1.1 processor running a BSD-based micro-kernel. Paint supports multiprogramming and models both kernel and user code. Impulse extensions include a virtually indexed, physically tagged first-level cache with an optional assist cache and an optional victim cache; a physically indexed, physically tagged second-level cache; an HP Runway bus and the Impulse memory system derived from the HP Kitty Hawk memory system. Note that this document focuses on the Impulse extensions only. The reader should consult the Paint technical report [2] for an overview of the Paint simulation environment and terminology.

The rest of this document is organized as follows. Section 2 briefly explains how the Impulse memory system fits in the system and how it communicates with other components in the system. Section 3 presents the top-level architecture of the Impulse memory system and describes the functionality of each major component. The details of these components are presented in successive sections. Section 4 talks about the master memory controller (MMC) and the performance model. Section 5 details the design of the Impulse remapping controller. Section 6 describes MC-based prefetching and the memory controller cache. Section 7.2 elaborates how the memory controller TLB works.

The Impulse project is still in an early stage. Many questions need to be answered in the future. In each section, the second to last subsection lists the problems, missing parts, and future work that are related to that section. One goal of this document is to assist the reader in running ISIM and reading the source code of ISIM. The last subsection of each section describes the relevant source files and the configurable parameters related to the component described in that section.

2 System Overview

Figure 1 shows an example of the computer systems simulated by ISIM. This example contains two CPUs, a dual I/O adapter, and a master memory controller (MMC) that are all connected together through a system memory bus. The MMC controls a DRAM Backend that contains a DRAM Scheduler and some DRAM chips.

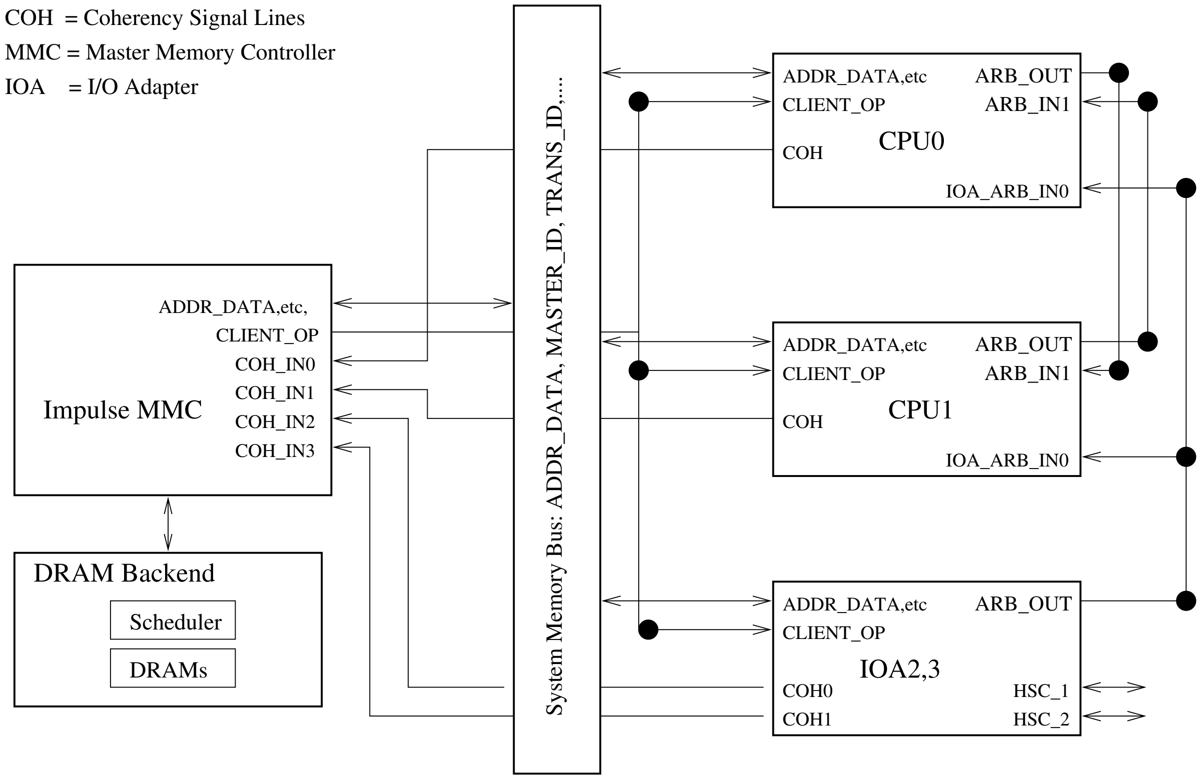


Figure 1: System Block Diagram

The system memory bus is a split-transaction, time-multiplexed address and data bus. It includes an address and data bus, master ID bus, transaction ID bus, and control signal bus. The CLIENT_OP bus is used for flow control and memory arbitration. The ARB_OUT/ARB_IN bus is used for memory arbitration. The COH bus sends coherency report from each CPU or I/O adapter to the MMC.

The MMC acts as the host of the system memory bus. Each CPU or I/O adapter connected to the system memory bus is called a *bus module*. This system supports multiple outstanding split-transactions per bus module and uses an efficient distributed, pipelined arbitration scheme, a predictive flow control mechanism, and a snoopy coherency protocol.

The master ID and transaction ID signals support multiple outstanding split-transactions by uniquely tagging all transactions. The MMC implements predicative flow control by specifying what types of transactions can be started at any time and driving the special CLIENT_OP bus accordingly. The distributed, pipelined arbitration is implemented using dedicated unidirectional wires (ARB_OUT and ARB_IN) from each bus module to other bus modules. Coherency is maintained by having all bus modules report coherency status on the dedicated unidirectional wires (COH) to the MMC. The MMC calculates the coherency response and sends coherency result through the CLIENT_OP bus along with the return data.

The width and frequency of the system memory bus are configurable. The number of bus modules that the system memory bus can support completely depends on the number of the master ID signals. The dedicated wires for flow control, coherency checks, and memory arbitration also limit the total number of bus modules that a system can have.

2.1 Split-Transaction

All read transactions are split-transactions. A read transaction is initiated by transmitting the encoded header, which includes the address, the issuer's master ID, and a unique transaction ID, to all other bus modules. The issuing bus module then relinquishes control of the system memory bus, allowing other bus modules to issue their transactions. When the data is available, the bus module supplying the data, typically the MMC, arbitrates for the system bus, then transmits the data along with the master ID and unique transaction ID so that the original issuer can match the data with a particular request.

Write transactions are not split, since the issuer has the data that it wants to write. The single-cycle transaction header is followed immediately by the data. From the CPU's perspective, a write transaction is retired right after the written data has been sent. But from the MMC's perspective, it is not retired until the data has been written into physical memory. By appropriately driving the CLIENT_OP bus, the MMC ensures that the same transaction ID will not be reused before a write is retired at MMC.

The maximum number of outstanding transactions each bus module can support is limited by the transaction ID signals in the system memory bus. For example, if there are only six transaction ID signals, the system can have at most 64 transactions in progress at one time.

2.2 Arbitration

The system memory bus uses a distributed, pipelined two-state arbitration scheme in which the determination of the arbitration winner is distributed among all bus modules on the system bus.

Each bus module drives a unique arbitration request signal and receives other bus modules' arbitration signals. On the first cycle, all interested parties assert their arbitration signals and the MMC drives the CLIENT_OP control signals indicating flow control information such as whether all bus modules will be preempted by a memory data return. During the second cycle, all bus modules evaluate the information received and make a unanimous decision about who has gained ownership of the system bus. On the third cycle, the bus module that won arbitration drives the system bus.

The latency between arbitration and bus access is two cycles. Once a bus module wins arbitration, it may optionally assert a special *long transaction* signal to extend bus ownership for a limited number of cycles for certain transactions. To maximize bus utilization, arbitration is pipelined: while arbitration can be asserted at any cycle, it is only effective for the selection of the next bus owner two cycles before the current bus owner releases the system bus.

Arbitration priority is designed to maintain fairness while delivering optimal performance. The highest arbitration priority is always given to the current bus owner through use of the long transaction signal, so that the current owner can finish whatever transaction it started. Since the data return is the completion of a previous split read request, it is likely that the requester is stalled waiting for the data and the data return will allow the requester to continue processing. Therefore, the second highest priority is given to the MMC for sending out data returns, using the CLIENT_OP bus to take control of the system bus. The third highest priority goes to the I/O adapter, which requests the system bus relatively infrequently, but needs low latency when it does. The lowest priority is assigned to the processors, which use a round-robin algorithm to take turns using the bus.

The arbitration protocol is implemented in such a way that higher-priority bus modules do not have to look at the arbitration request signals of lower-priority bus modules, thus saving pins and reducing costs. A side effect is that the arbitration of low-priority bus modules involves fewer modules than that of high-priority ones when the system bus is idle. This simplifies arbitration of the processors, which are the main consumers of the system bus, and does not hinder the MMC since it can predict when it will need the system bus for data return and can start arbitrating sufficiently early to account for the delay in arbitration.

2.3 Predictive Flow Control

Any live transaction on the system memory bus is never aborted or retired, so each bus module has to ensure it is safe to initiate a new transaction on the system bus. The CLIENT_OP bus is used to communicate what transactions can safely be initiated. Since the system bus is heavily pipelined, there are many queues in the MMC, processors, and I/O adapters to hold transactions until they can be processed. The CLIENT_OP bus is used to communicate whether there is sufficient room in these queues to receive a particular kind of transaction. Through various means, the MMC keeps track of how much room is remaining in these queues and restricts new transactions when a particular queue is critically full. For the purpose of flow control, a queue is considered "critically full" if the number of remaining slots in the queue is less than the number of transactions being started in the pipeline plus one more. "One more" is counted because CLIENT_OP cannot stop the current bus arbitration winner from issuing a transaction. Since the memory controller "predicts" when a queue needs to stop accepting new transactions to avoid overflow, this is called *predictive flow control*.

The primary benefit of predictive flow control is greatly reduced complexity, since bus modules no longer have to provide the capability of retrying an aborted transaction. This also improves bandwidth since each transaction is issued on the system bus exactly once. A secondary benefit of predictive flow control is faster completion of transactions that must be issued and received in order, particularly writes to I/O devices. If a transaction is allowed to abort, the second serially dependent transaction cannot be issued until the first transaction is guaranteed to complete. Normally, a transaction cannot be guaranteed to complete until after the receiving bus module has had enough time to look at the transaction and check the state of its queues for room, which is at least several cycles into the transaction. With predictive flow control, the issuing bus module knows when it wins arbitration that will make the first transaction be issued successfully and so it can immediately start arbitrating for the second transaction.

2.4 Coherency

A snoopy protocol maintains cache coherency among processors and I/O bus modules with a minimum amount of bus traffic. It minimizes the processor complexity required to support snoopy multiprocessing at the expense of the MMC complexity.

Whenever a coherent transaction is issued on the system bus, each processor or I/O adapter (acting as a third party) performs a snoop, or coherency check, using the physical address (and virtual index if only virtually indexed first level cache is used in the processors). Each bus module then sends its coherency check status directly to the MMC on dedicated COH signal lines. A coherency status of COH_OK, means either the cache line is absent or the cache line has been invalidated. A coherency status of COH_SHR means that the cache line is either already shared or has been changed to be shared after this coherency check. A coherency status of COH_CPY means that the third party has a modified copy of the cache line and will send the cache line directly to the requester.

After the MMC receives coherency status reports from every bus module, it will return memory data to the requester if the coherency status reports consist of only COH_OK or COH_SHR. If any bus module signals COH_SHR, the MMC will inform the requester to mark the line shared on the CLIENT_OP bus during the data return. If any bus module signals COH_CPY, however, the MMC will discard the memory data and wait for the third party to send the modified cache line directly to the requester in a cache-to-cache write transaction. The memory controller will also write the modified data into memory so that the requester can mark the line clean instead of dirty, freeing the requester from a subsequent write-back transaction if the line has to be cast out.

This coherency protocol supports multiple outstanding coherency checks and allows each bus module to signal coherency status at its own rate rather than at a fixed latency. Each bus module maintains a queue of coherent transactions received from the system bus to be processed in first-in-first-out order at a time convenient for the bus module. A read transaction in the MMC usually starts accessing memory before its coherency check completes (explained in section 4). As long as the coherency response is signaled before data is available from the MMC, delaying the coherency check will not increase memory latency. This flexibility allows the CPUs to implement a simple algorithm to schedule their coherency checks to minimize conflicts with the instruction pipeline for cache access.

2.5 Thoughts And Problems

What memory system will the Impulse memory system be based if we move to RSIM? Can Impulse technology be used in distributed share-memory architectures? How will things be different in a system with directory-based coherency protocol?

2.6 Source Codes And Configurable Parameters

Source codes

caches/*.[ch]:	L1 cache and L2 cache
bus/bus.[ch]:	the system bus
mmc/mmc_inter.c:	interface between the system bus and the MMC

Configurable parameters related to the CPU caches

L1C_assist_cache:	size of assist cache
L1C_victim_cache	size of victim cache
L1C_size:	size of L1 cache
L1C_line_size:	cache line size of L1 cache
L1C_associativity:	associativity of L1 cache
L1C_write_allocate:	whether or not use write-allocate for L1 cache
L2C_size:	size of L2 cache
L2C_line_size:	cache line size of L2 cache
L2C_associativity:	associativity of L2 cache
Cache_collect_stat:	whether or not collect statistics related to caches
Cache_prefetch_on:	whether or not enable L1 cache prefetching
Cache_debug:	debugging mode

Configurable parameters related to the system bus

Systembus_width:	bandwidth of the system bus
Systembus_frequency:	clock rate of the system bus
Systembus_dumpstats:	collect statistics related to the system bus
Systembus_debug:	debugging mode
Systembus_trace:	tracing mode

3 Memory System Overview

The Impulse memory system is constructed from five major components: the Master Memory Controller (MMC), DRAM Dispatcher, Slave Memory Controllers (SMC), Data Accumulate/Mux chip, 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 2. An Impulse memory system contains exactly one MMC and one DRAM dispatcher, but can have multiple SMCs, multiple RA busses, and multiple plug in memory modules. Figure 2 shows a simple configuration that has two SA busses, two SD busses, two MD busses, two RA busses, four RD busses, two Accumulate/Mux chips, four SMCs, and eight memory banks. Note that the DRAM dispatcher and SMCs do not have to be in different chips. Figure 2 just shows them in a way easy to understand. Whether or not to implement the DRAM scheduler in a single chip is currently an open question.

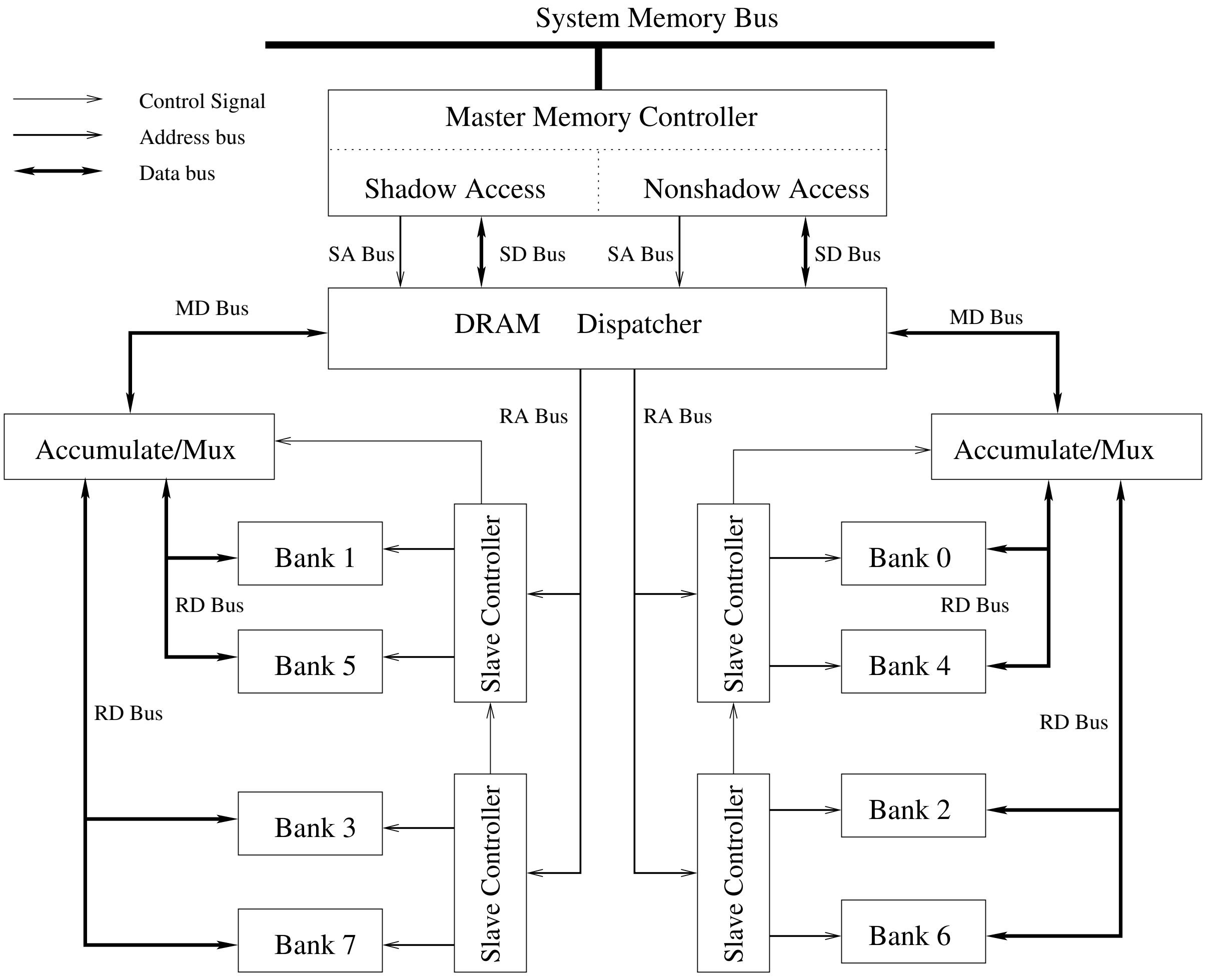


Figure 2: Impulse Memory System Block Diagram

3.1 Master Memory Controller and Slave Busses

The Master Memory Controller 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.

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 the simulator, the number of SA busses or SD busses can vary from one to the number of shadow descriptors plus one. If there is only one SA bus or SD bus, 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 rests will be used by a subset of the shadow descriptors. If the maximum configuration (one plus the number of shadow descriptors) is used, each shadow descriptor will exclusively use one and non-shadow accesses will use the remaining one. Note that the SA busses and SD busses are configured independently so the number of SA busses does not have to be the same as the number of SD busses.

The contention on SA busses is resolved by the MMC. The contention on SD busses is resolved by the DRAM dispatcher. When several memory accesses need the same SA bus or SD bus at the same time, the MMC or DRAM dispatcher picks a winner according to a round-robin algorithm and queues up the others. When a waiting queue becomes critically full, the sender (either MMC or SMC) will be stopped until there is enough room.

The width of the SA bus is always the same as the number of bits in a physical address. However, the width of the SD bus is configurable in ISIM, as is the clock rate of the SA bus or SD bus.

3.2 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 the MMC and memory banks via SA busses, RAM Data busses (RD bus), and Accumulate/Mux chips.

The RA bus is used to send memory accesses to SMCs. The Mux Data bus (MD bus) is used to pass data between the DRAM dispatcher and memory banks. If there is more than one SA bus, contention on the 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 the contention by picking a winner according to a round-robin algorithm and queuing up the others. If a waiting queue becomes critically full, the DRAM dispatcher will stop the sender (either MMC or SMC) from sending more requests.

The number of RA busses or MD busses is configurable. Each RA bus/MD bus must be used by the same number of SMCs. So the number of SMCs must be a multiple of both the number of RA busses and the number of MD busses. Note that SA busses and MD busses are set independently, so the number of SA busses is not necessarily the same as that of MD busses. The frequency of the RA bus, the width and frequency of the MD bus are configurable parameters too. Normally, the number of RA busses equals the number of SA busses; and the number of MD busses equals the number of RD busses. When there is only one SA bus, one SD bus, one RA bus, and one MD bus, the DRAM dispatcher does not need exist at all because the SA bus directly connects to the RA bus and the SD bus directly connects to the MD bus. One goal of future work is to find out how many SA/SD/RA/MD busses are needed to provide satisfactory performance.

3.3 Data Accumulate/Mux Chips

The Accumulate/Mux chips handle accumulating, multiplexing, and de-multiplexing between the MD bus and the independent RD bus. Specifically, it has two uses: first, to buffer transactions coming from the RD busses so that the RD busses can be freed up for other transactions; second, to solve contention on SD busses by queuing all other contenders except the winner. Each Accumulate/Mux chip serves an exclusive subset of all RD busses and has a fixed-size queue to buffer incoming data in first-in-first-out order. When the queue becomes critically full, the Accumulate/Mux chip will inform the DRAM dispatcher or SMCs to stop sending more requests.

3.4 Slave Memory Controller

Each Slave Memory Controller controls one RD bus and several DRAM chips sharing the RD bus. The SMC has independent control signals for each DRAM chip. The basic unit of memory 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 be shared between two adjacent banks, which introduces a 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 its DRAM type. Typically, each SDRAM chip contains two to four banks and each RDRAM chip contains eight to 16 banks.

The following logic/functionality is included in the SMC:

- keeping track of each memory bank's page buffer and deciding whether or not to leave page buffer open after an access;
- controlling an independent waiting queue for each memory bank and scheduling transactions in the waiting queue with the intention of reducing average memory latency;

- managing the interleaving of memory banks;
- controlling the data Accumulate/Mux chip;
- controlling DRAM timing and DRAM refresh.

When an access is broadcasted on a RA bus, all SMCs on the RA bus will see it, but only one SMC will respond. The interleaving scheme determines which SMC responds to a specified physical address. The number of SMCs and the number of memory banks that each SMC manages are configurable in ISIM. So are the capacity, frequency, width, page buffer size, and minimum access size of each memory bank.

3.5 Plug-in Memory Modules

Two types of DRAM, namely Synchronous DRAM and Direct Rambus DRAM, are simulated in ISIM. Both types of DRAM were simulated based on the current IBM products [6] [7].

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 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. DRAM also has to be refreshed hundreds of times each second in order to retain data in its memory cells.

3.5.1 Synchronous DRAM

The synchronous DRAM synchronizes all input and output signals to a system clock and therefore makes 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 3 shows the interleaving of some SDRAM transactions that access the same bank. Part 1 of Figure 3 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 3 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

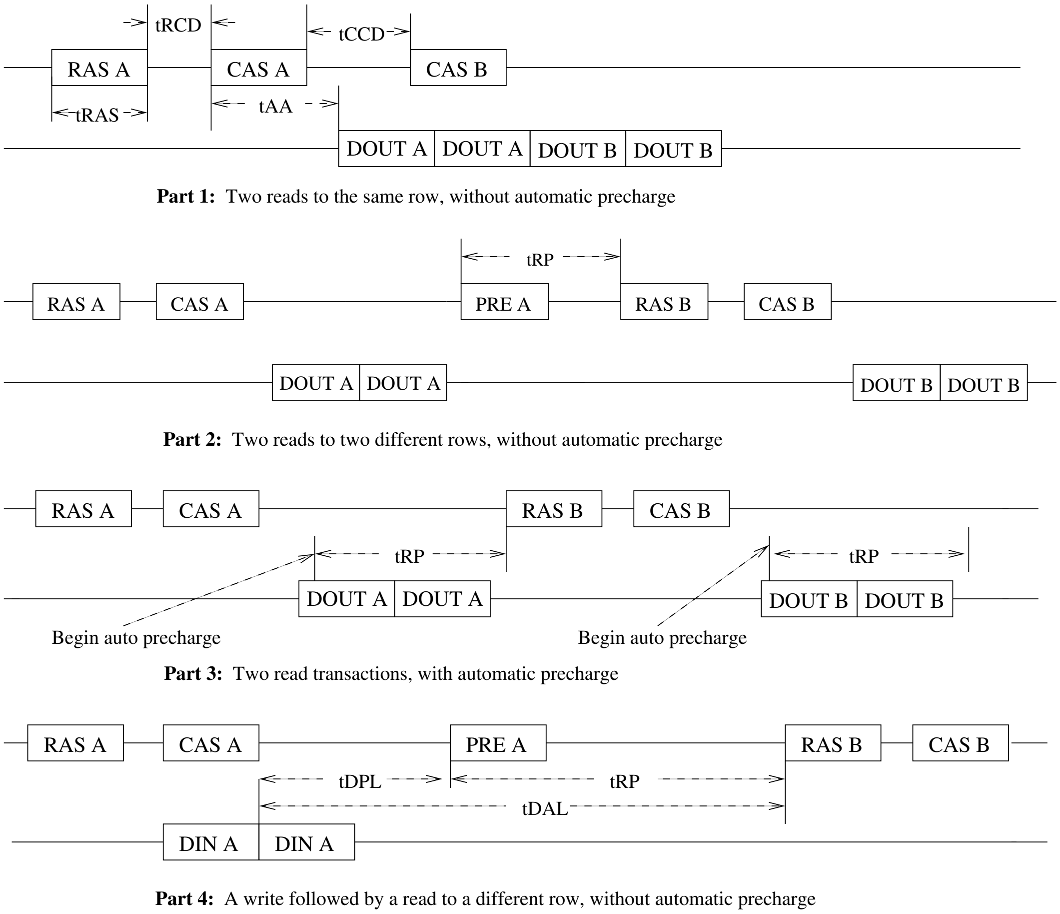


Figure 3: Examples of the sequences of SDRAM transactions

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 3 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 the same regardless of whether they access the same row or not. Part 4 of Figure 3 displays a write transaction followed by a read transaction that accesses a new row. An explicit precharge command must be inserted before the second transaction. 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 3 also shows the key timing parameters of the SDRAM. Their meanings and typical values in SDRAM clock cycles are described in Table 1.

Symbol	Meaning	Value
$tRAS$	minimum bank active time	7
$tRCD$	RAS to CAS delay time	3
tAA	CAS latency time	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.

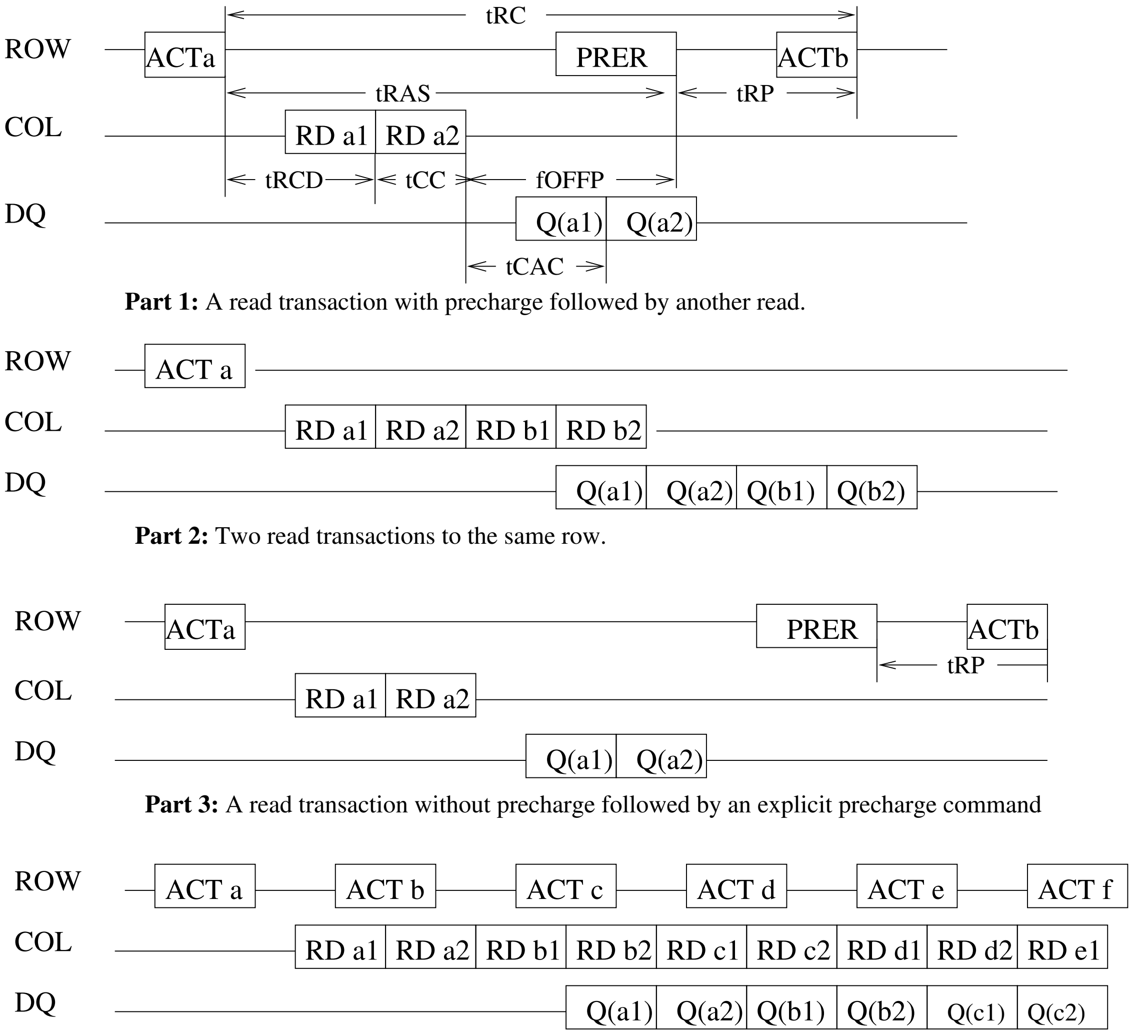
3.5.2 Direct Rambus DRAM

The Direct Rambus DRAM is a high speed DRAM developed by Rambus, Inc. 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 are transferred to and from the 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 4 shows the interleaving of some RDRAM transactions that all access the same chip. Part 1 of Figure 4 shows a read transaction with a precharge command, followed by another transaction to the same bank. Part 2 of Figure 4 shows the effective overlapping between two read transactions directed to the same row. Part 3 of Figure 4 shows a read transaction without precharge command followed by a transaction to a different row. Just as with SDRAM, the hot row has to be explicitly precharged before the second transaction. Part 4 of Figure 4 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 RDRAM clock cycles are presented in Table 2.

Symbol	Meaning	Value
tRC	the minimum delay between two successive ACT commands	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.



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 4: Examples of RDRAM operations

3.6 Thoughts And Problems

Partitioning of Functionality

Whether or not to implement the DRAM scheduler in one single chip is left to be solved in the future. Integrating everything into one chip probably will simplify hardware design, but it will lack scalability because the maximum number of RD busses that a DRAM Scheduler can support will be fixed.

Interleaving

How to interleave the memory banks has not been studied. The interleaving of memory banks directly affects the inherent parallelism of DRAM accesses. The Impulse memory system exhibits quite different DRAM access patterns that are not seen in a traditional memory system. For example, it may generate many less-than-cache-size DRAM accesses, which do not exist in a traditional memory system. So the traditional interleaving schemes may not work well with Impulse. Searching for the right interleaving scheme is an important part of future work.

Contention on SA/SD/RA/MD Bus

How to resolve the contention on SA/SD/RA/MD bus is still undecided. Current assumption that there is a fixed-size queue for each bus may be unrealistic. One goal of future work is to find out whether or not it is worthwhile to have multiple SA/SD/RA/MD busses. If it is not, this issue will disappear automatically.

3.7 Source Codes And Configurable Parameters

Source codes for the DRAM backend

dram/dram_aux.c:	some utility functions
dram/dram_init.c:	initialization of the DRAM backend
dram/dram_debug.c:	debugging support
dram/dram_main.c:	main timing model of the DRAM backend
dram/dram_stat.c:	statistics collection
dram/dram.h:	definition of major data structures
dram/dram_func.h:	all functions in the DRAM backend model
dram/dram_gen_def.h:	some useful macros
dram/dram_param.h:	parameters related to the DRAM backend

Parameters about the memory system

DRAM_sa_bus_cycles:	SA bus frequency
DRAM_sa_busses:	the number of SA busses
DRAM_sd_bus_cycles:	SD bus frequency
DRAM_sd_busses:	the number of SD busses
DRAM_sd_bus_width:	SD bus width
DRAM_num_banks:	the number of memory banks
DRAM_num_smcs:	the number of slave memory controller

Parameters about the DRAM backend

DRAM_type:	DRAM type, either SDRAM or RDRAM
DRAM_row_hold_time:	how long can a hot row be active
DRAM_refresh_delay:	cycles of a refresh operation
DRAM_refresh_period:	cycles of a refresh period
DRAM_frequency:	DRAM clock rate relative to CPU clock rate
DRAM_width:	width of each DRAM bank
DRAM_mini_access:	minimum DRAM access, in bytes
DRAM_block_size:	the block size, in bytes (i.e., cache line size)
DRAM_row_size:	an active row size, in bytes
DRAM_debug_on:	trace execution, used for debugging the simulator
DRAM_dumpstats:	collect statistics related to the DRAM backend
DRAM_dumplongstats:	dump detailed statistics for each component
DRAM_trace_on:	generate trace file
DRAM_trace_file:	name of trace file
DRAM_trace_maximum:	maximum entries in trace file
DRAM_trace_sample:	sampling step size

Timing parameters of the SDRAM

SDRAM_tR:	one $\overline{RAS}$ latency
SDRAM_tR2C:	time from $\overline{RAS}$ to $\overline{CAS}$
SDRAM_tC:	one $\overline{CAS}$ latency
SDRAM_tC2D:	time from $\overline{CAS}$ to DQ
SDRAM_tD:	one DQ latency
SDRAM_tP:	one precharge latency

Timing parameters of the RDRAM

RDRAM_tRCD:	time from ACT packet to first RD packet
RDRAM_tCAC:	time from RD command to the data out
RDRAM_tCC:	time for a RD packet
RDRAM_tRAS:	the minimum time from ACT command to PRER command
RDRAM_tOFFP:	the minimum time from last RD command to PRER command
RDRAM_tRC:	the minimum time between two successive ACT commands
RDRAM_tRP:	the minimum time from PRER command to next ACT command

4 Master Memory Controller

The Master Memory Controller is the core of the Impulse memory system. It communicates with processors and I/O adapters through the system memory bus and generates DRAM accesses. This section focuses on the memory performance model and the MMC's internal architecture. Figure 5 illustrates the controller's high-level behavior. For the sake of simplicity, the memory controller cache and memory controller-based prefetching are not shown in Figure 5. They will be added in in Section 6.

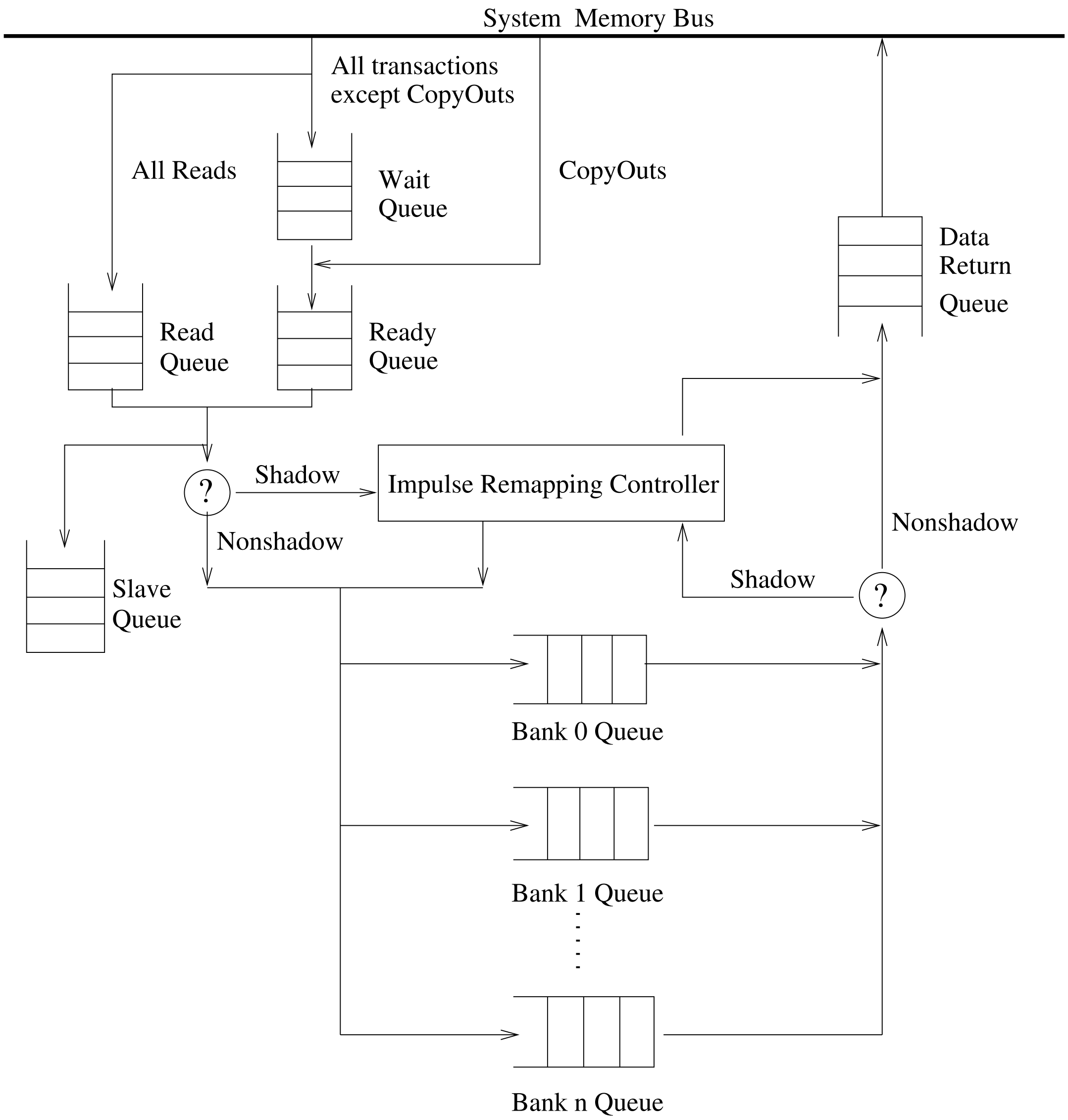


Figure 5: Memory System Performance Model

The MMC holds incoming transactions from the system memory bus in several queues and generates DRAM accesses according to certain rules. Many queues – the wait queue, read queue, ready queue, slave queue, bank queues, and data return queue – are used in the Impulse memory system. Except for the slave queue, all the queues are processed in first-in-first-out order. The bank queues are controlled by SMCs; and other queues are controlled by the MMC.

4.1 Wait Queue

All transactions except copyouts are entered in the wait queue. Transactions are processed when they reach the head of the wait queue.

When a non-coherent read transaction reaches the head of the wait queue, it is stalled until the corresponding data is returned from the main memory or the memory controller cache; its address is then checked against writes in the ready queue. If there are no conflicts (i.e., read and write address does not match), the read data is returned on the system bus; and the read is removed from the wait queue. If a conflict is detected, the read transaction will be moved into the ready queue for reissue. Coherent reads are treated identically except that they must wait for their coherency checks to complete. If a coherent read receives a copyout coherency response, its associated read data is discarded and the transaction is removed from the wait queue.

When a non-coherent write transaction reaches the head of the wait queue, it is removed from the wait queue and entered in the ready queue. Coherent write transactions are treated identically except that they must wait for their coherency checks to complete.

Copyout transactions are generated when dirty cache data – cache-to-cache copy data, replaced victim data, or data forced out due to a flush – is being written back to memory. In general, logically correct operation is determined by the order in which transactions are issued to the system bus. Copyout transactions are exceptions since in some instances a transaction that appears on the bus before a copyout transaction must be treated as if it had occurred after the copyout transaction. To guarantee transactions in the ready queue are logically correct in order, each client (CPU) must ensure that the copyout is issued to the system bus before sending the Cache Coherency Check (CCC) response for the conflicting transaction. If all transactions in the ready queue are performed in order, the results will be correct. The memory controller can therefore ensure correct behavior by placing all copyout transactions directly into the ready queue and all other transactions into the wait queue. Since a transaction does not move from the wait queue to the ready queue until after all CCC responses have been received for that transaction, any copyouts that should be logically complete before the transactions will be in the ready queue in front of it.

4.2 Read Queue

The read queue is used to reduce the memory latency for read transactions. Read transactions in the read queue are issued to the DRAM backend as quickly as possible; the transactions are not held in the read queue to wait for the CCC responses. When a read is received by the MMC, it is placed in both the read queue and the wait queue. The read placed in the read queue follows the *fast read path* and is called a *fast read*. The read placed in the wait queue follows a path that is guaranteed to produce correct results and is called a *logically ordered read*. The fast read is performed with higher priority and will return incorrect data when conflicts happen. The MMC detects and resolves the conflict cases to ensure correct results.

Given that logically ordered reads will always yield the correct results, conflict cases only occur in the window of time between when a fast read is issued and when the corresponding logically ordered read is issued. Two steps are performed to detect conflicts.

First, at the time that a read exits from the wait queue, the ready queue is checked for any writes to the same address. If no conflicting writes are in the ready queue, the result from the fast read will be correct. If conflicting writes are found in the ready queue, the results from the fast read may not be correct and the read will be placed in the ready queue to wait for reissue. In general, the fast read queue has priority over the ready queue, so transactions may stay in the ready queue for many cycles. When a conflict occurs, the ready queue is temporarily given priority over the fast read queue until the logically ordered read has exited the ready queue and been issued to the DRAM backend.

Second, whenever a write is issued to the DRAM backend, the wait queue is checked for any conflicting reads, which may or may not have been issued to the DRAM backend. This check is necessary because a conflicting write may be issued to the DRAM backend after the fast read is issued to the DRAM backend but before the conflict check is performed. (Note that once a write has been issued to the DRAM backend it is logically complete and is removed from the ready queue.) If such a conflict is found, the MMC will use the results of the logically ordered read instead of the fast read. This may result in false conflict cases, but the returned data is always guaranteed to be correct.

4.3 Ready Queue

The ready queue holds both copyout transactions taken directly from the system memory bus and transactions moved from the wait queue. Transactions in the ready queue are ready to be issued to the DRAM backend. Usually, most transactions in the ready queue are writes. Because the latency of completing writes is not as critical to system performance as the latency of completing reads, the read queue is given priority over the ready queue in most cases. Figure 6 gives the algorithm used to issue transactions from the ready queue or the read queue to the DRAM backend.

```
/*
 * Must drain the ready queue until we hit the read when
 * conflicts have been detected.
 */
if (drainreadyq) {
   next_transaction = FIFO_pop(ready_queue);
   if (next_transaction is READ)
      drainreadyq = 0;
}
/*
 * Look for possible ready queue overflow
 */
else if (lengthof(ready_queue) > READYQ_OFLOW ||
         empty(read_queue) && notempty(ready_queue)) {
   next_transaction = FIFO_pop(ready_queue);
}
/*
 * Otherwise, issue the head of the read queue.
 */
else
   next_transaction = FIFO_pop(read_queue);

send_to_DRAM_backend(next_transaction);
```

Figure 6: Code segment to issue transactions to the DRAM backend.

4.4 Slave Queue

All transactions that come from the read queue and the ready queue are placed in the slave queue. The primary purpose of the slave queue is to coordinate the use of memory system resources such as the data busses and memory banks. A transaction in the slave queue is removed when it is removed from its bank queue if the transaction is a normal access, or when all of its spawned transactions are removed from their bank queues if the transaction is a shadow access. Since transactions in different bank queues may be processed at different speeds, the slave queue will not be processed in first-in-first-out order.

The slave queue is an obsolete feature from the original system. In the Impulse memory system, the same coordinating functionality has been moved to the DRAM scheduler and SMCs. The only use of the slave queue is to control the total number of outstanding transactions sent to the DRAM backend, which actually can be accomplished by one single counter-register.

4.5 Bank Queues

Each transaction issued to the DRAM backend affects only one memory bank. Determining which memory bank is addressed by a particular transaction is a function of the number and size of memory banks and the interleaving scheme. A bank queue is placed in a relevant SMC. Currently, we are designing an algorithm for reordering the bank queues to reduce the average memory access latency.

4.6 Data Return Queue

The data return queue is used to hold data that has been returned from the DRAM backend, but has not yet been sourced to the system bus. Time spent in this queue is added directly to the memory latency perceived by the processor, so this queue should be empty most of the time. Data return has highest priority on the system bus, which means data usually does not need to be placed in the data return queue. However, there are three circumstances under which data is placed into the data return queue. The first case is when a multi-cycle transaction has started on the system bus. The data has to wait in the data return queue until the multi-cycle transaction completes. The second case is when this transaction's CCC responses have not been received. The transaction has to stay in the wait queue and the data is placed in the data return queue until its CCC completes. The third case is when the remapping controller and the DRAM backend transmit data on the system bus at the same time.

4.7 Impulse Remapping Controller

Each shadow transaction from the read queue or the ready queue has to be processed by the Impulse remapping controller before accessing the DRAMs. The Impulse remapping controller translates each shadow address to one or more physical addresses and pushes a transaction into the bank queue for each generated physical address. When the data associated with the shadow transaction is returned from the DRAM backend, it also has to go back to the remapping controller for further processing. Details about the Impulse remapping controller are presented in Section 5.

4.8 Flow Control

In a queue-based system, some set of algorithms or protocol rules are required to ensure that none of the queues overflows. The size of each queue in the Impulse memory system is fixed. So is the number of total write data registers in the MMC.

When the MMC runs out of write data registers, it denies all other devices to access the system bus. The ready queue has (1 + the number of write data registers) entries. Since the ready queue can hold at most one read with the rest being writes, it can never overflow. When both the wait queue and the ready queue are full, the MMC denies all other bus modules to access the system bus. When the wait queue is full but the ready queue is not, the MMC will disallow any transactions except copyouts. The read queue has the same size as the wait queue and every transaction in the read queue must also have a copy in the wait queue, so the read queue can never overflow. When the slave queue is full, the MMC will stop issuing transactions to it. Flow control for the data return queue is accomplished by never issuing a read transaction to the DRAM backend until there is an empty slot in the data return queue.

4.9 Thoughts And Problems

Bank Queue Reordering

No bank queue reordering algorithm has yet been designed. Intuitively, a good reordering algorithm should have the following features: giving non-prefetching accesses priority over prefetching accesses, determining whether or not to leave a row open after each access, putting accesses to the same row close to one another, guaranteeing no access will wait in a queue for a very long time, and ensuring the logically correct order while performing reordering.

Flow Control of the Bank Queue

Flow control on the bank queues is missing in the current design. In the original design, each bank queue had the same size as the slave queue. The flow control on the slave queue guaranteed no overflow in any bank queue. However, in Impulse, each transaction placed in the slave queue may send multiple transactions to the bank queues. The experiments showed that a bank queue with the same size as the slave queue would easily overflow in Impulse. Possible solutions for this problem are:

- increasing the depth of memory bank, i.e., the size of the bank queue (this is what we are using now);
- putting the remapping controller before the slave queue, and placing transactions generated by the remapping controller into the slave queue and stopping the remapping controller when the slave queue overflows;
- creating a connection between the remapping controller and the SMCs so that when a shadow descriptor generates a transaction whose designated bank queue is full, it will be stopped from generating new transactions until there is more room.

Remapping Controller Placement

When to start shadow address translation, i.e., where to put the Impulse remapping controller, may affect performance significantly. The remapping controller is placed between the ready queue and the bank queues in current design. One alternative is to put it in parallel with the wait queue. So the address translation can be started when a transaction is placed in the wait queue, i.e., when the transaction is coming off the system bus. Each design has its advantages and disadvantages.

The Impulse remapping controller could be placed in parallel with the wait queue.

- Pros
  - Remapping is started as early as possible, which means remapping overhead is reduced as much as possible.
  - Real physical addresses participate in coherency checks. This is critical if data may be addressed by both shadow address and non-shadow address at the same time and the data is allowed to be written.
- Cons
  - Such a placement may generate huge number of transactions. The sizes of all queues have to be increased.
  - Many more coherency checks are needed.

- The address remapping may take many cycles. A transaction whose remapping takes many cycles will stall transactions behind it in the wait queue.
- The distance between the remapping controller and the DRAM chips is logically long, which means passing data between them is not efficient.
- The performance model has to be modified significantly.

The Impulse remapping controller could be placed after the ready/read queue and before the bank queues.

- Pros
  - This organization is easy to integrate into the original system. Very little change is needed in the performance model.
  - It will not stall other transactions unnecessarily.
  - Communication between the remapping controller and memory banks is efficient.
- Cons
  - Shadow physical addresses, not real physical addresses, participate in coherency check.
  - The address remapping is started relatively late.

4.10 Source Codes And Configurable Parameters

Source codes

mmc/mmc_aux.c:	some auxiliary functions
mmc/mmc_init.c:	initialization of memory controller
mmc/mmc_debug.c:	debugging support
mmc/mmc_main.c:	performance model
mmc/mmc_stat.c:	statistics collection
mmc/mmc_global.h:	definition of global variables
mmc/mmc_param.h:	definition of parameters
mmc/mmc.h:	definition of major data structures
mmc/mmc_func.h:	definition of functions
mmc/mmc_gen_def.h:	some useful macros
mmc/mmc_stat.h:	data structures to record statistics

Almost all parameters related to the performance model are hardcoded in current implementation. Parameters about the remapping controller can be found in section 5.

5 Impulse Remapping Controller

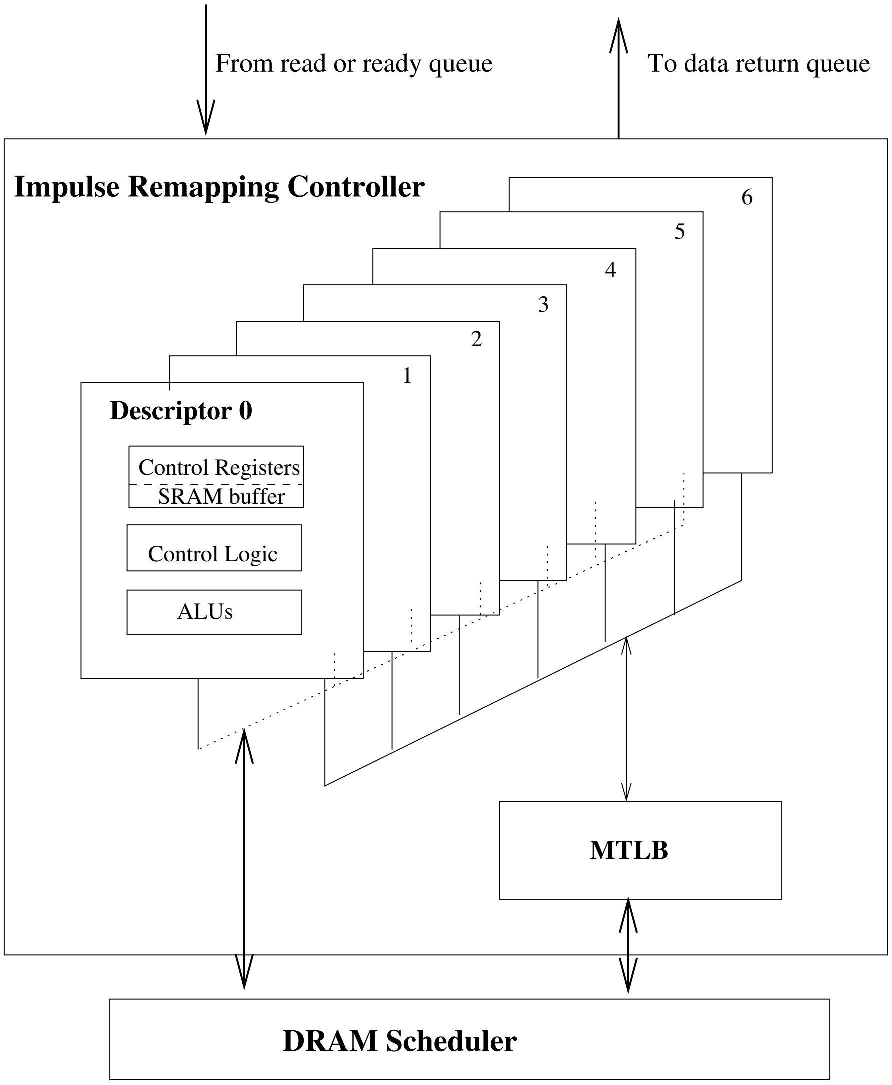


Figure 7: Impulse Remapping Controller Block Diagram

Figure 7 shows the internal structure of the Impulse remapping controller. The Impulse remapping controller contains several shadow descriptors (seven in current design) and a memory controller TLB. Each shadow descriptor includes some *control registers*, a small SRAM cache, a simple ALU unit, and the assembly logic. The control registers store remapping information such as remapping type and shadow address space region. The small cache holds remapped data prefetched from DRAM. The simple ALU unit translates shadow addresses to pseudo-virtual addresses, which are translated into real physical addresses by the MTLB. The assembly logic gathers sparse data retrieved from DRAM into dense cache lines. The functionality of shadow descriptors ALUs has yet to be completely specified. For now, we assume only integer operations, all of which complete in a single memory cycle.

The control registers are memory-mapped and have to be set with appropriate values before being used to perform remapping. Only the operating system is allowed to configure the remapping controller. All the shadow descriptors use the same memory addresses for their control registers. When the operating system tries to configure a shadow descriptor, it will first send the index number of the selected descriptor to the remapping controller, and then write information to the control registers in that shadow descriptor.

Only one read or write transaction is allowed to access a shadow descriptor at one time. The remapping controller has a queue for each shadow descriptor to hold the waiting transactions in first-in-first-out order. The size of each queue equals the size of the slave queue.

Currently, the Impulse remapping controller supports the following remapping algorithms: strided scatter/gather; scatter/gather through an indirection vector; no-copy page coloring; and no-copy superpage formation. Each different remapping algorithm uses different information and procedure to remap data. The following subsections describe these remapping algorithms in detail.

5.1 Strided Scatter/Gather

Non-unit stride accesses are common in real applications. Examples include accessing matrices in column-major order when the matrices are stored in row-major order, accessing the same field of every record in a database, and accessing tiles of a dense matrix. Applications with non-unit stride accesses may load cache lines in which most of the data goes unused by the CPU. This causes problems like cache pollution, low bus utilization, and low cache hit ratios. Impulse can create cache-friendly aliased data structures in shadow memory such that these dense structures contain only the strided data accessed by the CPU. The following example illustrates creating and using such an alias vector for a program that calculates the sum of all employees' salaries.

```
struct record {
  ...
  float  salary;
  ...
} all_employees[TOTAL_EMPLOYEES];

foo() {
  float sum = 0;
  float *salaries = Impulse_remap(sizeof(struct record),
                                  sizeof(float),
                                  OFFSET(struct record, salary),
                                  TOTAL_EMPLOYEES, ...);
  for (int i = 0; i < TOTAL_EMPLOYEES; i++)
```

```
        sum += salaries[i];
    }
```

Impulse_remap() is a system call to set up the remapping controller. It first chooses a shadow descriptor and then writes necessary information into it. In our example, the information that the remapping controller needs includes:

saddr_start:	starting shadow address – the physical address of *salaries[0]*;
stride_size:	stride size – *sizeof(struct record)*;
object_size:	object size – *sizeof(float)*;
object_count:	number of objects – *TOTAL_EMPLOYEES*;
object_offset:	offset of required object in the stride – (*&(all_employees->salary)* $-$ *all_employees*);
ptable_ptr:	a dense, flat page table to map the original data structure *all_employees* from virtual pages to physical pages.

When the remapping controller receives a shadow access *saddr*, it performs as follows.

0. Calculate the number of DRAM accesses needed to gather a cache line:

   if object_size $<$ cache_line_size, *count* = cache_line_size / object_size;

   if object_size $\geq$ cache_line_size, *count* = 1.

   This step is actually done when the mapping is initialized. Once the mapping is set, it will not change until reset.

1. Calculate the offset in the shadow address space:

   soffset = *saddr* $-$ saddr_start.

2. Calculate the index of the stride that contains the requested object:

   index = *soffset* / object_size.

   If object_size $>$ cache_line_size, calculate the offset of the requested cache line in the object:

   coffset = *soffset* % object_size.

3. Calculate the offset in the pseudo-virtual address space:

   voffset = *index* $\times$ stride_size + object_offset (+ *coffset*, if object_size $>$ cache_line_size);

4. Look up the physical address *paddr* in the MTLB. If the lookup misses in the MTLB, the MTLB will access physical memory for the relevant page table entry.

5. Access physical memory at physical address *paddr* for
   object_size bytes, if object_size $\leq$ cache_line_size;
   cache_line_size bytes, if object_size $>$ cache_line_size.

6. if (- -*count* > 0), calculate the next item's offset in pseudo-virtual address space:
   voffset += stride_size;
   goto step 4;

Steps 4-6 are fully pipelined, so a new physical address can be generated each cycle. This pipeline is stalled only when an MTLB miss occurs in step 4. After all the memory accesses complete, the relevant shadow descriptor will create a dense cache line and send it back to the requester through the system bus.

5.2 Scatter/Gather Through An Indirection Vector

This type of remapping is used by applications that access their major data structures through indirection vectors. The following is a simple example of using this remapping.

```
foo() {
  int   iv_array[IVSIZE];
  double major_data[SIZE];

#ifndef Use_Impulse_Optimization
  /* Non-impulse version */
  for (i = 0; i < IVSIZE; i++)
      ... = ... major_data[iv_array[i]] ... ;
#else
  /* Impulse version */
  double * alias_array = Impulse_remap (...);
  for (i = 0; i < IVSIZE; i++)
       ... = ... alias_array[i] ...;
#endif
}
```

The following information is needed for performing a scatter/gather mapping through an indirection vector.

saddr_start:	starting shadow address
stride_size:	stride size
object_size:	object size
object_count:	number of objects
object_offset:	offset of required object in the stride
iv_elem_size:	element size of the indirection vector
iv_paddr_start:	starting physical address of the indirection vector
ptable_ptr:	dense, flat page table mapping original data structure
iv_buffer:	a cache line to store elements of the indirection vector

The shadow descriptor takes the following actions to gather a cache line of data addressed by the shadow address *saddr*.

0. Calculate how many DRAM accesses are needed to gather a cache line:

   if object_size $<$ cache_line_size, *count* = cache_line_size / object_size;

   if object_size $\geq$ cache_line_size, *count* = 1.

   This step is actually done when the mapping is initialized. Once the mapping is set, it will not change until explicitly reset.

1. Calculate the offset in the shadow address space:

   soffset = *saddr* $-$ saddr_start.

2. Calculate the index of the requested object:

   index = *soffset* / object_size.

   If object_size $>$ cache_line_size, calculate the offset of the requested cache line in the object

   coffset = *soffset* % object_size.

3. Calculate the physical address for the associated element of the indirection vector:

   iv_paddr = iv_paddr_start + *index* $\times$ iv_elem_size;

4. If *iv_paddr* is not in the IV buffer, the shadow descriptor fetches a whole cache line containing the required indirection vector element from physical memory into the IV buffer. The shadow descriptor then interprets the element to get its value *rindex*, which is the index into the original array.

5. Calculate the offset in pseudo-virtual address space:

   rindex $\times$ stride_size + object_offset (+ *coffset*, if object_size $>$ cache_line_size).

6. Look up the physical address *paddr* in the MTLB. If the lookup misses in MTLB, the MTLB will access physical memory for the relevant page table entry.

7. Access physical memory at physical address *paddr* for
   object_size bytes, if object_size $\leq$ cache_line_size;
   cache_line_size bytes, if object_size $>$ cache_line_size.

8. if (- -*count* $>$ 0), calculate the physical address of the next indirection vector element:
   iv_paddr += iv_elem_size;
   goto step 4;

When an application sets up scatter/gather remapping through an indirection vector via the operating system, the operating system moves the indirection vector into a contiguous physical space, so the physical addresses of indirection vector elements can be used directly by the remapping controller.

As with the strided remapping, steps 4 to 6 are pipelined to generate one physical address each cycle. MTLB misses stall the pipeline.

5.3 No-copy Page Coloring

No-copy page coloring is designed to optimize data layout in physically indexed caches. The example in Figure 8 explains how no-copy page coloring works. This example maps data structure **A** to the third quadrant of a physically indexed L2 cache. The operating system first allocates a shadow address space four times of the size of L2 cache and then creates a page table in the CPU to map each quarter of **A** to an appropriate region in the allocated shadow address space, as shown in Figure 8. Assuming the allocated shadow address space is L2-cache-size-aligned, all the grey boxes in the shadow address space are mapped into the same portion of the L2 cache. Note that the white spaces in shadow address space are wasted in this design. Since shadow address space is not directly backed up by real physical memory, wasting shadow address space will not waste any real physical memory.

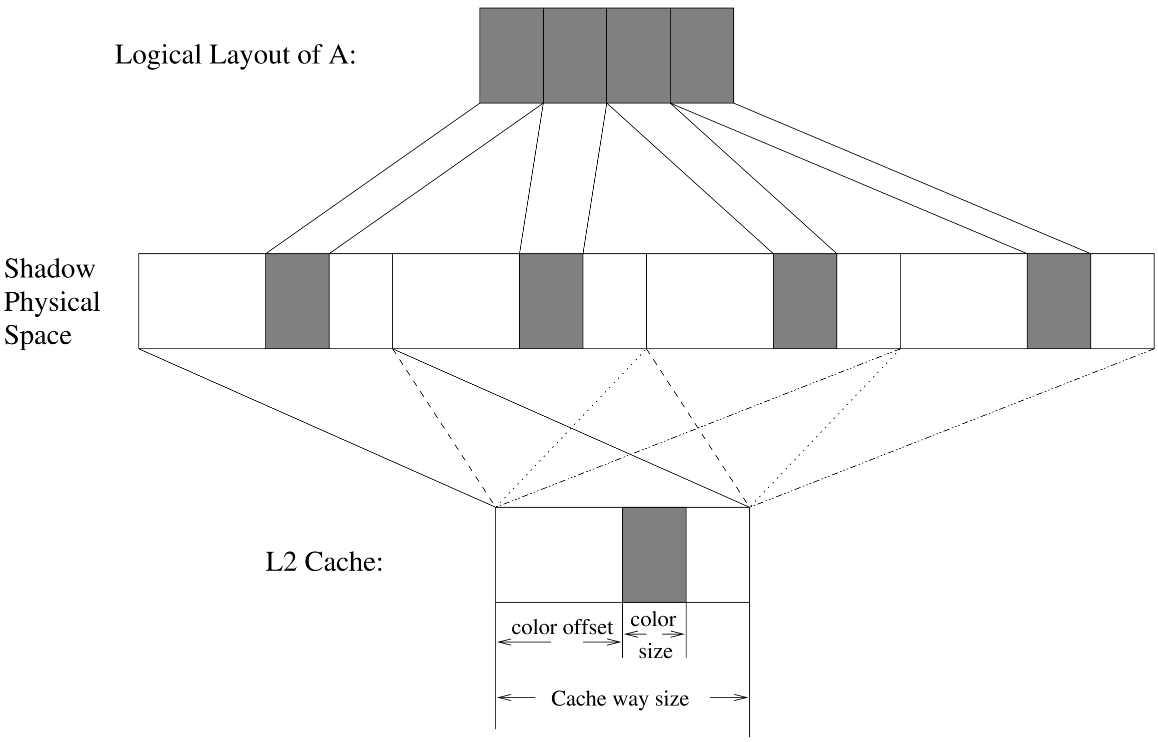


Figure 8: Map A into the third quadrant of L2 cache

The following information is needed to perform the no-copy page-coloring remapping.

saddr_start:	starting shadow address
cache_size:	effective cache way size (size / associativity)
color_size:	size of the region to which shadow data is mapped
color_offset:	offset in the region to which shadow data is mapped
ptable_ptr:	a page table mapping original data structure

When a shadow descriptor receives a shadow access *saddr*, it takes the following actions.

1. Calculate the offset in the shadow address space:

   soffset = *saddr* - saddr_start.

2. Calculate the index of the color where the requested object lies:

   index = *soffset* / cache_size;

   and the offset of the requested data in this color:

   coffset = *soffset* % cache_size - color_offset.

3. Calculate the offset in pseudo-virtual space:

   voffset = *index* $\times$ color_size + *coffset*.

4. Look up the physical address *paddr* in the MTLB. If the lookup misses in MTLB, the MTLB will access physical memory for the relevant page table entry.

5. Access physical memory at physical address *paddr* for

   cache_line_size bytes.

5.4 No-copy Superpage Formation

This type of remapping creates superpages from disjoint physical pages without copying[1]. It can be used by applications which suffer from poor TLB performance. This remapping needs only the starting shadow address and the page table. Since the offset in shadow memory is the same as the offset in pseudo-virtual memory, the physical address generation is simply two steps: calculate the offset in the shadow address space and perform the MTLB lookup.

5.5 Thoughts And Problems

ALU

There are many divisions and multiplications in the remapping algorithms described above. If a division is really needed, the latency of a division operation surely will not be the assumed one cycle. If the *object_size* (for scatter/gather) or *cache_line_size* (for page coloring) is a power of two, a division can be replaced by a logic right shift operation. Likewise, if the *stride_size* (for scatter/gather) or the *color_size* is a power of two, a multiplication can be replaced by a logic left shift operation. In real applications, the object size is usually a power of two but the stride size is often not. We will have to determine whether or not Impulse should support division. If no division unit is used, the compiler must find a way to pad each stride or each object to be a power of two. If both the stride size and object size are powers of two, all the arithmetic operations are additions, or subtractions, or logic operations. Figure 9 shows how easily the remapping can be done for the most complicated remapping — scatter/gather through an indirection vector — when the size of each data item is a power of two.

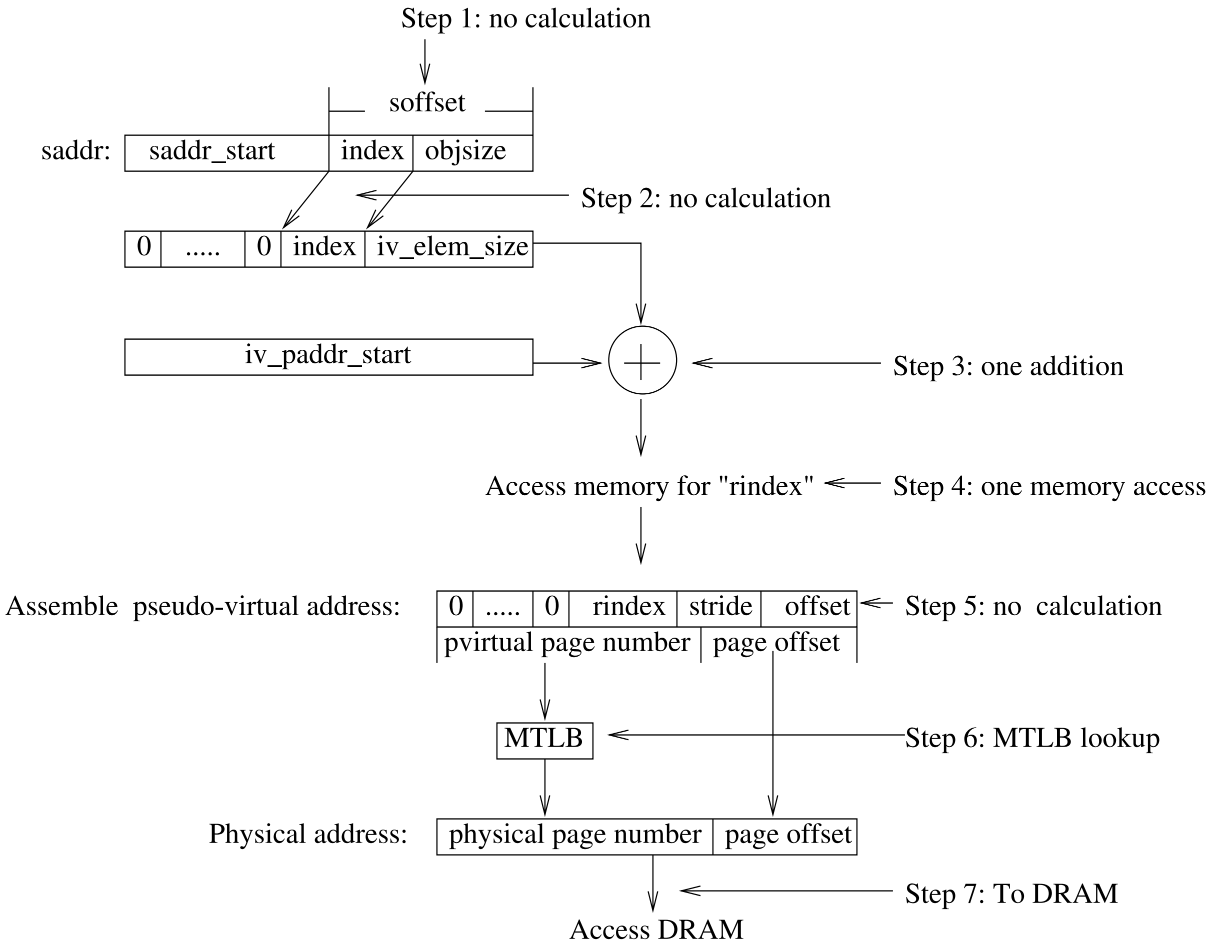


Figure 9: Address manipulations in scatter/gather through an indirection vector, assuming *stride_size*, *object_size*, and *iv_elem_size* all are powers of two.

Security/Boundary Checks

Security/boundary checks are missing. When and how to perform boundary checks is still undecided. The simplest way is to use a page table. If the page table shows that a pseudo-virtual address is out of bounds, the transaction will be aborted. But this strategy may raise problems when a cache line is gathered across page boundaries. For example, if the first half of a cache line should be gathered from page A, which is valid, and the second half should be gathered from page B, which not valid, this cache line cannot be correctly gathered because there is no way to fetch its second half. A simple solution is to put restrictions on remapping — never allowing scatter/gather to cross page boundaries. Another solution is to check every pseudo-virtual address. For scatter/gather through an indirection vector, we also have to check every access to the indirection vector. If the boundary check should be applied, we must decide whether to handle it in parallel with other operations or to make it an independent stage in the simulation model.

Page Faults

The current simulator does not have a mechanism to handle page faults generated by the MTLB. There are two obvious solutions: first, abort this transaction and let CPU load the page into main memory, then reissue this transaction; second, add an extra communication mechanism between the CPU and the MMC so that the MMC can instruct the CPU to load a required page, and the CPU can then inform the MMC when the required page has been loaded.

Number of Control Registers

Not all the remapping algorithms use the same number of control registers. If all shadow descriptors are given the same number of control registers, some control registers will not be used by some remapping algorithms. If control registers are expensive, we have to find a way to conserve control registers. One possible solution is to give each shadow descriptor different number of control registers and let it handle only one or several specified remappings, depending on how many control registers it has. Another solution is to convert "unused" control registers into data buffers.

Waiting Queue Control

The current design uses a big waiting queue for each shadow descriptor, which seems unreasonable. One alternative is to use a global queue for all shadow descriptors. If the global queue is not larger than the slave queue, no extra flow control is required. If the global queue is smaller than the slave queue, the MMC has to be stopped when the global queue is full. A global queue will not maintain first-in-first-out ordering, however. Transactions in the global queue will be started as soon as the corresponding shadow descriptor becomes available, which will happen in a dynamically determined order.

5.6 Source Codes And Configurable Parameters

Source codes

mmc/mapcontroller.c:	simulator of the Impulse remapping controller
mmc/mapcontroller.h:	header file

Parameters for the Impulse remapping controller

Shadow_Region_Mask:	the shadow address space
Num_descriptors:	number of shadow descriptors
Descriptor_start_cycles:	overhead to start a new mapping
Descriptor_add_cycles:	cycles of an addition/subtraction operation
Descriptor_div_cycles:	cycles of a division operation
Descriptor_mul_cycles:	cycles of a multiple operation
DesQueue_num:	how many queues in the remapping controller
DesQueue_len:	size of each queue

6 MC-based Prefetching and MCache

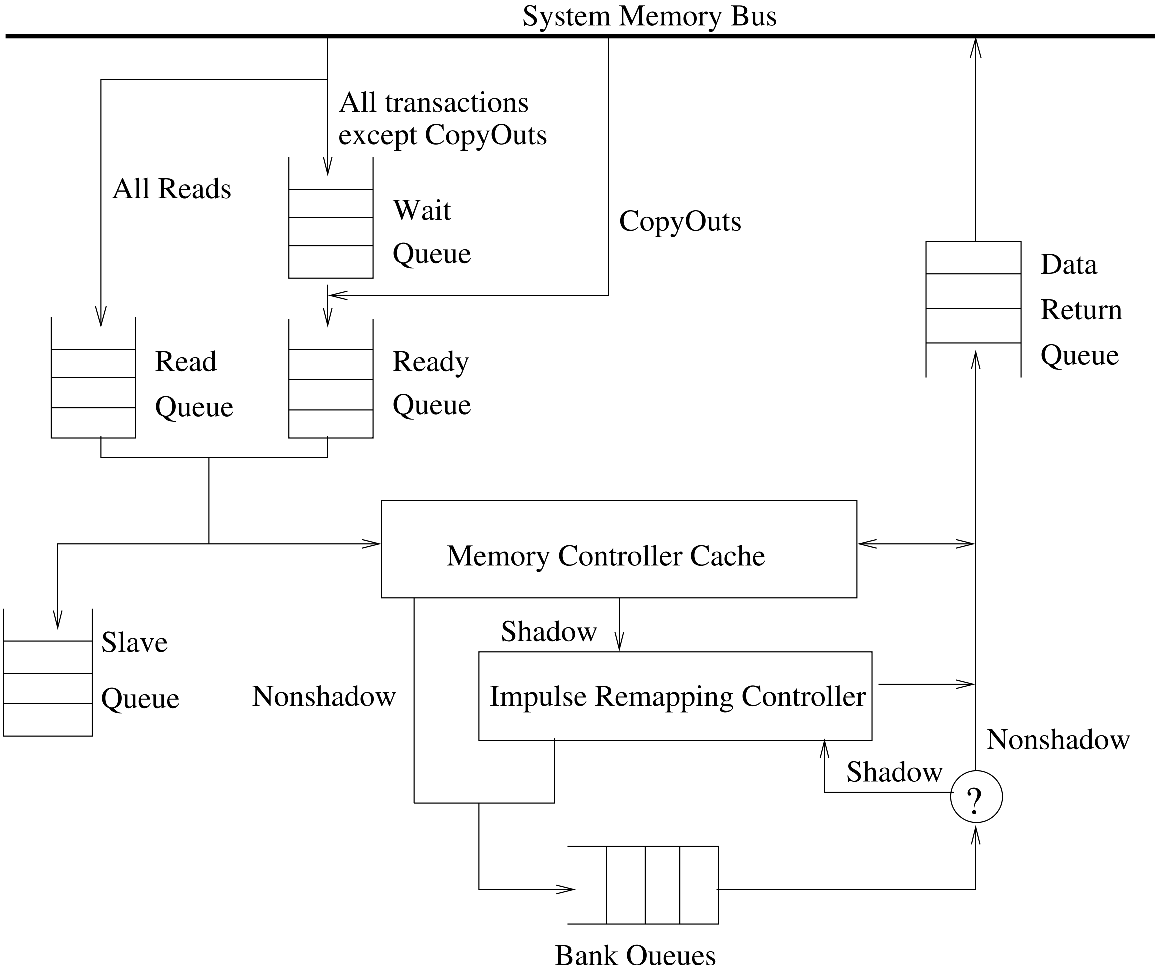


Figure 10: The memory performance model with the MCache

The second important feature of Impulse is its support for prefetching at the memory controller – MC-based prefetching. MC-based prefetching uses a small SRAM cache – MCache – in the memory controller to reduce the effective memory latency perceived by the processor. Specifically, the Impulse memory controller can speculatively load data from DRAM into the MCache. If a memory request hits in the MCache, the MMC can quickly provide the requested data without going through a full DRAM access. Figure 10 shows the MCache's position in the memory performance model. The MCache introduces one more cycle into the critical timing path of a memory access. Since the MMC is fully pipelined, it adds only one cycle into a sequence of continuous accesses. Compared to 50-plus-cycle memory latency, one extra cycle is insignificant. MC-based prefetching is more important for shadow accesses than for non-shadow accesses. Each shadow access must go through the remapping controller, which may take up to hundreds of cycles. It is crucial for the MMC to start loading shadow data as early as possible to hide the cost of remapping.

6.1 MC-based Prefetching

We face several challenges during the design of MC-based prefetching. The first challenge is to determine what kind of algorithm MC-based prefetching should use and how aggressive it should be? At the point that this document is being written, we use a simple algorithm: sequential prefetching for non-shadow data, and configurable-stride prefetching for shadow data.

The second challenge is to decide when to generate a prefetch address and when to issue a prefetch. A prefetch address may be used immediately, or be saved and then used later. A prefetch does not need to perform coherency check regardless of whether the transaction spawning the prefetch is coherent or not because the coherency check will be performed when the CPU issues a memory request for the prefetched data.

The possible times in the performance model to generate prefetch addresses are:

1. when transactions are placed in the wait queue;
2. when transactions come off the wait queue;
3. when transactions return from the DRAM backend.

Which one is better depends on applications' access patterns. For example, for applications that generate bursts of memory transactions, it may be the best to generate prefetch addresses when transactions return from the DRAM backend. For applications that generate sequential memory transactions, generating prefetch addresses when transactions are placed in the wait queue may be the best choice.

The possible times to issue prefetches are:

1. when a prefetch address is generated;
2. when both the read queue and the ready queue are empty;
3. when no outstanding transactions exist in the MMC.

If one of the last two options is selected, a prefetch will not be issued at the time when the address is generated, and the MMC will have to save the prefetch address somewhere. To issue a prefetch when its address is being generated seems like a good choice because the MMC is lightly loaded in the current uniprocessor system. When the memory system is heavily loaded, generating a prefetch address for every read transaction will make the MMC and DRAM backend too busy to complete normal transactions within reasonable time. On the another hand, issuing prefetches when both the read queue and the ready queue are empty or when the MMC is free will cause every few prefetches. These facts must be considered in the design of MC-based prefetching.

Not all the combinations of the options mentioned above are valid. For example, combining case 2 of generating prefetch addresses with case 2 of issuing prefetches does not make sense; and combining case 1 of generating prefetch addresses with case 1 of issuing prefetches allows a prefetch to go ahead of the transaction that triggered the prefetch.

Since address remapping may take many cycles, a prefetch transaction may occupy a shadow descriptor for a long time. If a non-prefetch transaction needs a shadow descriptor being used by an ongoing prefetch, it has to wait until the prefetch transaction completes its address translation. Because the cycles waiting for a shadow descriptor are directly added to the memory latency, the prefetch transaction should be either suspended or aborted. The current simulator does neither suspension nor abortion. The prefetch strategies used in the current design have worked well for the applications that we have tested. Other applications may benefit from a more sophisticated algorithm.

6.2 MCache Organization

The MCache includes a modest-size buffer to store prefetched, non-shadow data and a small buffer (two to four cache lines) for each shadow descriptor to store prefetched, shadow data. The small buffer for each shadow descriptor is fully-associative, and its design is trivial. Except where stated otherwise, the following discussion uses the MCache to represent the buffer for non-shadow data.

The MCache uses a FIFO replacement policy. The behavior of the MCache is quite different from the behavior of CPU caches. Since the most frequently used data should reside in the CPU caches, the MCache data will probably not be used frequently. Replacement policies such as LRU and NRU simply do not work well with the MCache. Our experiments show that FIFO outperforms LRU in all cases tested.

The MCache is four-way associative, physically indexed, and physically tagged. Its normal size is 4kilobytes. Its line size equals the size of an L2 cache line. The MCache uses a write-invalidate protocol, i.e., any write memory transaction invalidates matched data in the MCache. As a result, data in the MCache can never be dirty, which means that victim data can simply be discarded when a conflict occurs. Each MCache line has the following format:

used (1bit)	state (1bit)	physical tag (22bits)	data (32-128bytes)

The *used* bit indicates whether or not this line is in use. The *state* bit indicates the state of this line — either *Prefetching* or *Valid*. The *tag* is the most significant 22 bits of a physical address. *Prefetching* means that the data is being prefetched right now but have not returned from physical memory. After the prefetched data have returned, the *state* bit will be changed to *Valid*. In order to avoid generating duplicate DRAM accesses when a cache line being prefetched is also requested by a processor, an MCache line is reserved when a prefetch transaction is issued. If an access needs the same data that an ongoing prefetching transaction is fetching, it will hit in the MCache and wait for the return of the desired data. A line reserved for an ongoing prefetch transaction will not be victimized before the prefetched data return. In the case that all of four lines that a prefetch transaction can use are occupied by other ongoing prefetch transactions, the new prefetch transaction is simply discarded.

6.3 Thoughts And Problems

About MCache

Currently, non-shadow data and shadow data have independent buffers. Would it be beneficial to put buffers in all shadow descriptors together, or to coalesce all buffers in the MMC? The FIFO replacement policy outperforms both LRU and NRU in our experiments. It will be very interesting to test more policies. For example, what kind of performance would "Most Recently Used" yield?

About MC-based Prefetching

The following questions need to be answered. When should the MMC generate prefetch addresses? When should it issue prefetches? Should it, under what circumstances, abort ongoing prefetch transactions? What kind of algorithm should MC-based prefetching eventually use, and how aggressive should it be? Should the MMC generate a prefetch address for each load transaction, for only a subset of load transactions, or for more types of transactions other than just load.

6.4 Source Codes And Configurable Parameters

Source codes

mmc/mmc_cache.c:	memory controller cache
mmc/mmc_prefetch.c:	MC-based prefetching
mmc/mmc.h:	data structures used by MCache module

Parameters about MC-based prefetching and the MCache

MMC_prefetch_on:		turn on/off MC-based prefetching
	0	turn it off
	1	prefetch non-shadow data only
	2	prefetch shadow data only
	3	prefetch both non-shadow and shadow data
	1,2,3	to issue prefetch when data return from DRAMs
	4	to issue prefetch when both read and ready queue are empty
	8	to issue prefetch when the MMC is free

MMC_cache_update:	use write-update protocol for MCache
MMC_cache_replace:	replacement policy of MCache
MMC_cache_size:	size of MCache
MMC_cache_line_size:	line size of MCache
MMC_cache_associativity:	associativity of MCache
MMC_descache_size:	size of the buffer in each descriptor
MMC_descache_associativity:	associativity of the buffer in each descriptor

7 Memory Controller TLB

The MMC first translates shadow addresses to pseudo-virtual addresses, and then to physical addresses. The mapping from pseudo-virtual to physical addresses is like the one from virtual to physical addresses in the CPU/MMU. Just as the CPU uses the TLB (Translation Lookaside Buffer), the MMC uses a memory controller TLB (MTLB) to accelerate the pseudo-virtual to physical translations.

7.1 Hardware Design

When an application issues a system call to set up a remapping, the operating system creates a dense, flat page table to store the pseudo-virtual-to-physical translations of the data structure being remapped. We refer to this page table as the memory controller page table. Since the memory controller page table is dense and flat, it can be indexed by the virtual offset of the original data structure. Each 4-byte entry of the memory controller page table has the following format:

valid (1)	ref (1)	modify(1)	fault(1)	frame(20)	unused (8)

The *valid* bit indicates whether this mapping is valid. The *reference* bit indicates whether a page has been referenced. This bit is set on the first MTLB miss for the page. The *modify* bit indicates whether a page has been written. This bit is set on the first write reference for the page. The *fault* bit indicates whether the page is in the main memory. The *frame* is the physical page number. Because we assume 32-bit physical address and four-kilobytes page size, the *frame* has 20 bits. The eight *unused* bits are reserved for future expansion.

In the simulator, the MTLB has configurable size and associativity, uses a Not Recently Used (NRU) replacement policy, and has a one-cycle access latency. Each entry of the MTLB has the following format:

valid (1bit)	locked (1bit)	tag (20bits)	refcount (16bits)	PTE (4 bytes)

The *valid* bit indicates whether or not this mapping is valid. The *locked* bit indicates whether or not this entry is reserved for an ongoing MTLB-fill transaction. A *tag* is formed by a pseudo-virtual page offset and the index number of the shadow descriptor that generated this pseudo-virtual address. The *refcount* bit records the total number of references to the page. It is used to implement the NRU replacement policy. To avoid overflow in *refcount*, *refcount* is reset after a certain number of translations have accessed the MTLB.

A small buffer inside the MTLB is used to cache the page table entries loaded from physical memory. Each MTLB miss checks the buffer first before sending a fill request to DRAM. If an MTLB miss hits in the buffer, it only takes one extra cycle to load the translation into the MTLB. If it misses in the buffer, the MTLB will generate a fill request to load a cache line containing 32 translations from physical memory. The following pseudo-code describes the behavior of the MTLB when it receives a lookup request.

```
 If (lookup hits in the MTLB) {
   if (the entry's valid bit is set)
      goto hit_MTLB;
   else /* the entry's valid bit is not set */
      issue an exception;
      return;
 }
 else { /* the lookup failed in the MTLB */
    check for a match in the buffer;
    if (hit in buffer) {
       take one cycle to load the translation into the MTLB;
       goto hit_MTLB;
    }
    else { /* missed in buffer */
       launch a read transaction to load translation from memory;
       wait until read return fills the buffer;
       allocate an entry in the MTLB, and load translation into the MTLB;
       goto hit_MTLB;
    }
  }
hit_MTLB:
   /*
    * form physical address
    */
   merge physical page frame number from the MTLB entry
   with page offset portion of the pseudo-virtual address;

   /*
    * Set reference/modified bits appropriately.
    * Write it back to main memory if the translation is changed.
    */
   if (this is the first reference/write on the given page) {
      set reference/modified bit;
      write the translation back to main memory;
    }
```

The translations in the MTLB must be kept consistent with the ones in main memory. If the MTLB did not write the modified translations back to main memory immediately, fatal errors would occur. For example, if a given page translated by the MTLB is written and the MTLB does not write the translation back to physical memory before the page is evicted out of physical memory, the operating system will not know that the modified bit was set and will not write the page back to disk. Consequently the writes at the page will be lost.

In the simulator, the MTLB fill or write-back requests are non-coherent memory transactions and are put in an independent queue which has higher priority than both the read queue and the ready queue. Any MTLB access should be completed as soon as possible so that it can release the dependent transactions as early as possible, so we MTLB-initiated transactions the highest priority.

7.2 Source Codes And Configurable Parameters

Source codes

mmc/mmc_tlb.c:	MTLB simulator
mmc/mapcontroller.h:	data structures related MTLB

Parameters related to MTLB

MMCTLB_usetlb:	whether or not to use MTLB
MMCTLB_dumpstats:	turn on/off statistics collection
MMCTLB_debug:	turn on/off debugging printing
MMCTLB_numentries:	number of entries in MTLB
MMCTLB_associativity:	associativity of MTLB
MMCTLB_buffersize:	size of the buffer in MTLB
MMCTLB_resetcount:	the number of transactions needed to reset "refcount"
MMCTLB_tagcheck_cycles:	cycles of MTLB tag check

References

[1] M. Swanson, L. Stoller, J. Carter. Increasing TLB Reach Using Superpages Backed by Shadow Memory. In *25th Annual International Symposium on Computer Architecture*, July 1998.

[2] L. Stoller, M. Swanson, R. Kuramkot. Paint: PA Instruction Set Interpreter. *Technical Reports UUCS-96-009*, University of Utah, July 1997

[3] J. Veenstra, R. Folwer. MINT Tutorial and User Manual. *Technical Reports 452*, University of Rochester, Aug. 1994.

[4] T. Hotchkiss, N. Marschke, R. McClosky. A New Memory System Design for Commercial and Technical Computing Products In *HP Journal*, Vol.47 No.1, pages 44-51, Feb. 1996.

[5] W. Bryg, K. Chan, N. Fiduccia. A High-Performance, Low-Cost Multiprocessor Bus for Workstations and Midrange Servers. In *HP Journal*, Vol.47 No.1, pages 18-23, Feb. 1996.

[6] IBM Advance, 256Mb Synchronous DRAM – Die Revision A. Aug. 1998.

[7] IBM Advance, 64Mb Direct Rambus DRAM. Nov. 1997.

[8] L. Schaelicke. DRAM Backend for Impulse Memory Controller. *Unpublished report*, April 1998.

[9] Kitty Hawk Memory System, External Reference Specification, Revision B. May 1995

[10] R. Schumann. Design of the 21174 Memory Controller for DIGITAL Personal Workstations. *Digital Technical Journal*, Vol.9 No.2, Nov. 1997