CACHE FILTERING ALGORITHM FOR LEAST FREQUENTLY USED DATA WITH ACCURATE MEMORY SIMULATION

By Kiu Kwan Leung

B.Eng. Ryerson University, 2007

Toronto, Ontario, Canada

A thesis

presented to Ryerson University

in partial fulfillment of the

requirement for the degree of

Master of Applied Science

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2010

Author's Declaration

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other Institutions or individuals for the purpose of scholarly research.

Author's signature:

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Author's signature:

Abstract

Cache Filtering Algorithm For Least Frequently
Used Data with Accurate Memory Simulation

Master of Applied Science

Department of Electrical and Computer Engineering

Ryerson University

We propose a cache filtering algorithm to improve processor performance using a small buffer inside the processor and an algorithm to filter least frequently used accesses from L1 and L2 caches. The algorithm uses simple DRAM fast-page accessing mode to identify accesses that are not previously accessed or not frequently used and keep them out of the cache system and store them in small buffer.

We have also added a realistic page interleaved DDR3 memory simulation model to the SimpleScalar simulator. This model supports any processor and memory clock speeds, different sets of memory latencies, various configurations of memory banks and channels.

Results show that the filtering algorithm could improve performance of some applications compared to the same system that does not use the filtering algorithm.

Acknowledgement

I would like to express my deep gratitude to my supervisor, Dr. Nagi Mekhiel for offering me such a chance to work with him as his master student. It is always his brilliant ideas and directions that saved me out of my hurdles throughout the research process and it is always his patience and tolerance that made this research work such an enjoyable learning experience.

I must also thank my parents for their constant support throughout my life. I am thankful for their encouragement and their teachings, which had shaped me up to what I am over the years and allowed me to have such a strong sense that I have a nice place called home. I truly believe that without them, I will not be able to receive such a wonderful learning experience at Ryerson University.

Thirdly, I appreciate the assistance offered by my former supervisor, Dr. Cungang Yang, throughout my graduate studies. His has always been one of my best guides to direct me through the program and taught me how to be a good graduate assistance. I must also thank Professor Ken Clowes for his great advices. As one of the most skilled programmers across the department, he nicely explained the meanings of the advanced syntax used by the author of the SimpleScalar simulator, which enabled me to understand simulator and become a SimpleScalar developer quickly.

I am grateful for the financial assistance that I received through my supervisor and the Electrical and Computer Engineering department at Ryerson University. It is the only reason why I can stay focus and enjoy this research process.

Finally, I sincerely appreciate the Electrical and Computer Engineering department's lab support team for their kind assistance. Without their technical knowledge about the UNIX operating system, I will never finish the research at ease.

Dedication

To my supervisor: Professor Nagi Mekhiel

To my family: father and mother

To my former supervisor: Professor Cungang Yang

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the years computer engineers and architects have been researching and developing new techniques to enhance computer performance. These technologies include increasing operating frequency of processor and memory, pipelining and developing superscalar architecture to exploit instruction-level parallelism(ILP), introducing multi-level cache system to hide the overgrowing memory latency, developing simultaneous multithreading (SMT) and multicore processors to harness the power of thread-level parallelism(TLP), etc.[1, 12 and 14]

However, studies have revealed the fact that today's computer performance is not only governed by the amount of work performed per clock cycle of the processor, but also the amount of data bandwidth the memory subsystem can offer, which is inversely proportional to the latency of the memory system. According to [1 and 3], processor performance has been improving at a rate of 52% annually starting from the year 1986. However, the evolution of memory performance has only been maintaining at a steady pace of 7% per year since 1980. The industry noticed the issue of the growing performance difference between processors and memory and tried to solve this problem through various techniques.

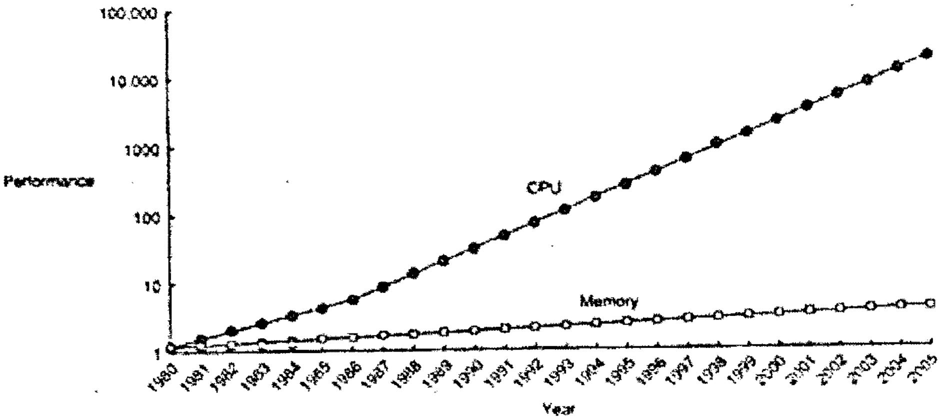


Figure 1.1: The performance ratio comparing to 1980s baseline computer systems – the growing performance gap between processor and memory [1]

But as the performance gap between processor and memory keeps growing, modern processors will make more frequent stalls to wait for instructions and data to be transmitted from memory. Moreover, traditional solutions to solve the memory bandwidth/latency problem will also reach its physical limits so engineers must spare no efforts to reinvent more efficient way of utilizing the cache and memory systems, which is the main focus of this work.

1.1 List of Contributions and Objectives of the Research

The main objective of this thesis is to enhance the performance of Professor Mekhiel's patented work – *Cache Filtering Method and Apparatus.* The overall research contributions are summarized as follows:

- Studied modern computer programs' memory address usage pattern and established a relationship between cache pollution caused by single memory accesses and the concept of cache filtering.
- Illustrated the problem of cache pollution.
- Created a page interleaved DDR3 memory module for the SimpleScalar Simulator version 3.0d, which will be used to implement and test our cache filtering algorithm. Our memory module supports multi memory channels, multi memory banks, user configurable access latencies and allows easy extensions to other memory types (RDRAM, SDRAM, XDRRAM, etc.).
- Improved Professor Mekhiel's cache filtering algorithm by:
    - Reinventing a new set of cache miss handling procedures to replace the original cache filtering scheme's procedures.
    - Introducing a filter buffer to capture filtered data and prevent eviction of useful cache blocks.

1.2 Motivation

We began our research by studying the memory address traces of some SPEC CPU 2000 component benchmarks. The traces were obtained from Brigham Young University Trace Distribution Center [15]. They are recorded from the first instruction to the 50 millionth instruction of the benchmarks. Through studying the address traces, we can learn history of addresses requests generated by the benchmarks from their initialization stage to middle of the execution.

We wrote a java program, which takes a memory address trace as an input file and counts the followings:

- The total amount of memory access requests generated within the instruction range;
- The total amount of unique memory addresses being requested for only once (single accessed addresses);
- The total amount of unique memory addresses being requested for more than once (repeated addresses);
- The total amount of times the program made accesses request to repeated addresses (repeated access).

The counting results and the average of the results are shown in the following tables:

Table 1.1: Total amount of memory access requests generated within execution range

Apsi	equake	gcc	mcf	mgrid	
10388240	10327879	10356038	10377283	10363629	
Parser	perlmk	swim	twolf	vortex	average
9222144	10326311	9164118	10024960	10270839	10082144

Table 1.2: Total amount of single accessed addresses within execution range

Apsi	equake	gcc	mcf	mgrid	
62231	52093	112909	58889	400161	
Parser	perlmk	swim	twolf	vortex	average
86838	78304	316984	141746	344597	165475

Table 1.3: Total amount of repeated addresses within execution range

Apsi	equake	gcc	mcf	mgrid	
112459	89547	175754	83503	475660	
Parser	perlmk	swim	twolf	vortex	average
146835	113867	351524	211584	438342	219908

Table 1.4: Total amount of repeated accesses within execution range

Apsi	equake	gcc	mcf	mgrid	
10326009	10275786	10243129	10318394	9963468	
Parser	perlmk	swim	twolf	vortex	average
9135306	10248007	8847134	9883214	9926242	9916669

With the counting results available, we can calculate:

- the percentage of the average single accessed addresses;
- the percentage of the average repeated addresses;
- the percentage of the average single access;
- the percentage of the average repeated accesses.

Average total amount of unique memory addresses being requested within the execution range (total unique addresses):

Total unique addresses = average single accessed addresses + average repeated addresses

= 165475 + 219908 = 385383

Percentage of average single accessed addresses (average single accessed addresses %):

Average single accessed addresses % = (average single accessed address / total unique addresses) x 100%

= (165475 / 385383) x 100% = 42.9%

Percentage of average repeated addresses (average repeated addresses %):

Average repeated addresses% = 100% - average single accessed addresses %

= 100% - 42.9% = 57.1%

Average total amount of accesses to single accessed addresses (Average single accesses):

Average single access = average single accessed addresses = 165475

Percentage of average single access (average single access %):

Average single access % = *(average single access / average total memory access) x 100%*

= *(165475 / 10082144) x 100% = 1.6%*

Percentage of average repeated access (average repeated access %):

Average repeated access % = *100% - average single access %*

= *100% - 1.6% = 98.4%*

The calculations are indicating the 57.1% of repeated addresses are occupying 98.4% of the total memory accesses as they are being used for twice or more. The remaining 42.9% of memory addresses, contributing to only 1.6% of the total accesses, are being used for only once and they will be discarded by the benchmarks.

We believe that allowing the 1.6% of single accesses to enter the cache system will ultimately lead to degraded system performance because:

- they will cause cache pollution by replacing the frequently used data originally resides in the cache (to be discussed later);
- they will waste the valuable memory bandwidth to transmit them back and forth from the main memory to the cache and vice versa as they will be accessed only once and will soon be replaced by other cache blocks;
- they will also increase the memory row/page miss rate as those accesses will break the normal memory accessing pattern, causing the memory to open another rarely used page and to close the frequently used one.

1.2.1 Cache pollution caused by rarely used data blocks

As cache is designed to provide high speed access to the processor, it will service any kind of access by loading the requested data and keeping the data in its cache blocks, in hope that the program will reuse the data in the future. The only problem with this approach is that it will cause a situation known as cache pollution, which is very expensive in terms of memory bandwidth and processor time as it destroys the cache's data locality (the word locality implies that when data is needed, it is already located at the cache). To explain the situation, we have included the following example:

A computer has a small 2 way associative cache with 2 cache blocks and only 1 index, which implements the least recently used (LRU) cache replacement scheme, is executing a program. The program being executed makes frequent use of data located at addresses X and Y so the cache is fully occupied by those data.

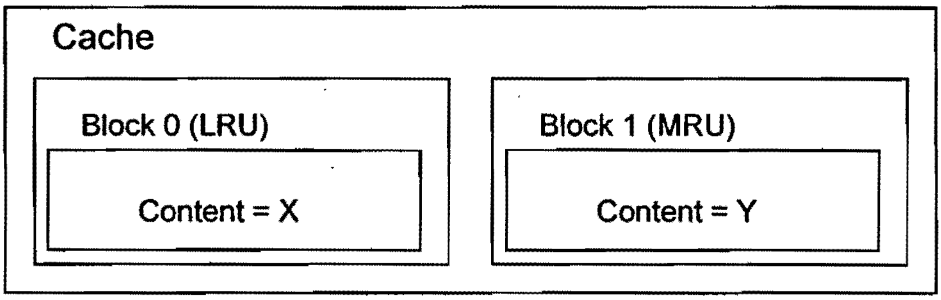


Figure 1.2: Initial content of cache

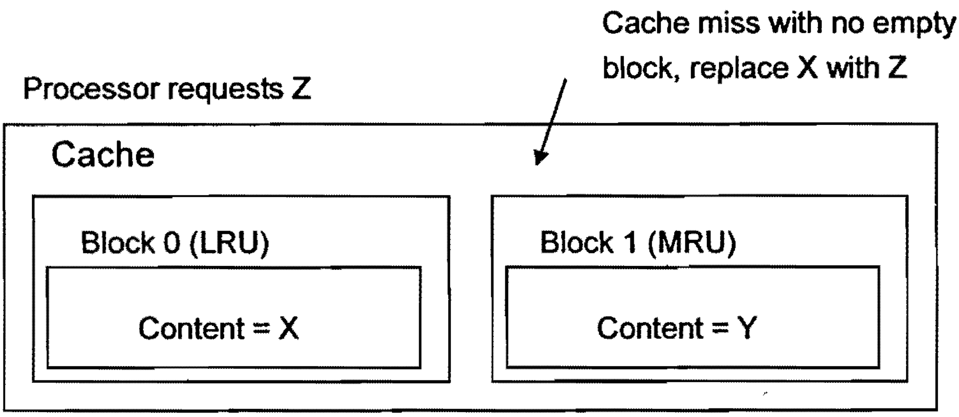


Figure 1.3: Cache miss forces cache to replace Block 0

Suddenly, the program makes request for a data from address Z, which will only be used once (or rarely used) throughout the program execution. Suppose the program had just accessed address Y, then the cache would have no space for the request of Z. It must replace cache block 0, which stores data at address X, by writing the content back to main memory and load data at address Z into the cache block.

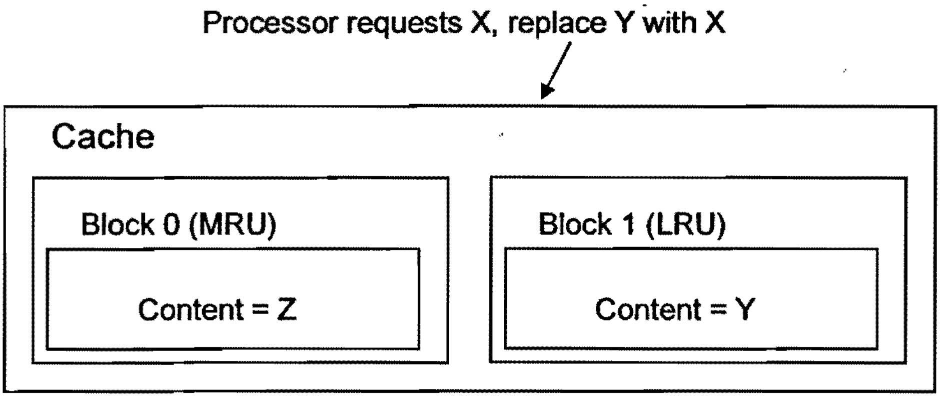


Figure 1.4: Cache content after replacing X with Z, program requests for X again causing another cache miss to replace block 1

After the program finishes using data at address Z, it resumes the regular pattern by first request for X then Y. This causes another two additional cache misses because the least recently used policy will first replace block 1 with data from address Y by data from X, then replace block 0 with data from address Z with replaced by Y.

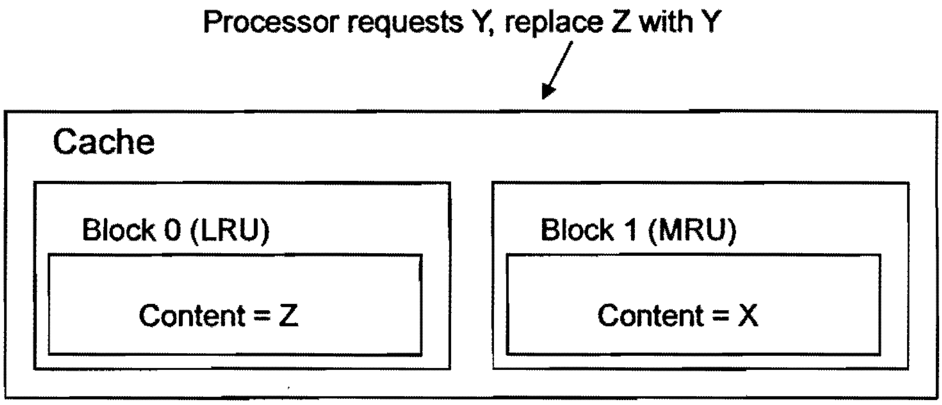


Figure 1.5: Cache content after replacing Y with X, program requests for Y again causing another cache miss to replace block 0

Note that each cache block replacement, depending on whether they have previously been modified or not, would require one to two memory accesses (writeback and loading) to service. Assuming all data accesses happened in this example were writes, address Z had caused three cache misses and six memory accesses to service such a request. Therefore, one can conclude that the access to address Z had polluted the cache's original content, consumed large amount of memory bandwidth, caused the processor to wait and yielded degraded performance.

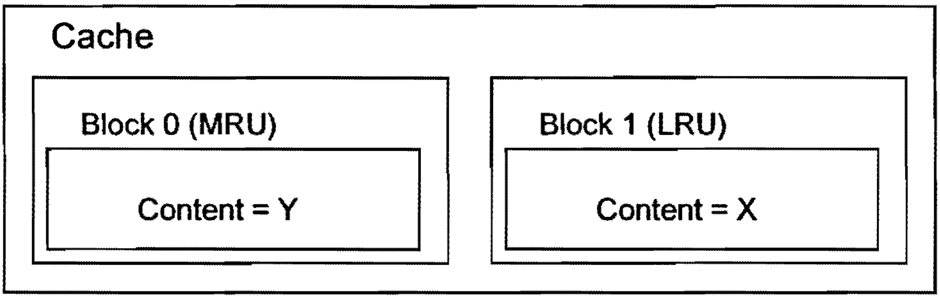


Figure 1.6: Final cache content after replacing Z with Y

1.3 The organization of this thesis

With the motivation of this research properly defined, we will organize the following chapters to:

- introduce the SimpleScalar simulator tool set (Chapter 2);
- describe the design of the page interleaved DDR3 memory performance simulation module (Chapter 3);
- propose the cache filtering algorithm for least frequently used data (Chapter 4);
- simulation and discussion of the cache filtering algorithm for least frequently used data (Chapter 5);
- give an overview of the current technology related to cache filtering and compare the performance gain of each technology (Chapter 6);
- Discuss the future work and conclusion (Chapter 7).

Chapter 2

Methodology

In this chapter, we will introduce the SimpleScalar Simulator tool set. The SimpleScalar Simulator tool set [12, 13 and 14] is a collection of many simulators written with the C language. Out of the many available simulators, we have chosen to evaluate our work on a simulator named sim-outorder.

Sim-outorder is a detailed simulator that can model a 32bit out-of-order execution superscalar processor. Out-of-order execution is a technique to boost processor performance by allowing the processor to, based on the availability of the processor's functional units, change the instruction execution order of a program. The approach is valid as long as the correctness of the data being processed is maintained and data are being written back to memory in the correct order.

Sim-outorder is also a performance simulator. That is, sim-outorder simulates by executing a compatible program binary/executable file like an ordinary computer, with an emulated processor, memory and hard disk (2GB of virtual memory). Throughout the simulation process, sim-outorder will collect performance statistics, such as cache hit rate and cycle per instruction (CPI), and display the information collected at the end of the simulation.

The SimpleScalar tool set also contains a number of simpler yet useful simulators, such as sim-fast, a simple processor functional simulator that does not have any error checking; sim-safe, a sim-fast equivalent with error checking capabilities; sim-cache, a simple cache simulator and sim-bpred, a branch predictor simulator.

Due to the nature of this work, we must briefly discuss how sim-outorder handles cache/memory access latencies. Sim-outorder was designed as a superscalar processor which has both instruction and data level 1 (L1) cache, optional separated or unified instruction and data level 2 cache and a main memory module.

Access to each of the above mentioned components is modeled by calling the corresponding access latency calculation function in the simulator. For cache accesses, the level 1 cache latency calculation function (L1 latency function) will perform all cache access activities and return an integer value that can accurately represent the cache access latency in processor clock cycles.

If the data being requested is found in the cache, the L1 latency function will compare the cache hit latency with the time when the cache block is accessible (when the block is completely loaded from the next level) and return the bigger of the two values. In case of cache miss where accesses to the next level of the hierarchy (writeback and data loading) are required, the L1 latency function will update the cache blocks and call the next level's latency calculation function. Such a calling sequence will end when one of the following scenarios is reached – the requested data is found in any subsequent level or the data is not found in all cache levels and access to memory is needed. Once the cache miss situation is being properly serviced by calling the appropriate access functions, the L1 latency function will sum up all the latencies returned by the subsequent level latency calculation functions and return the sum as the latency of such cache access.

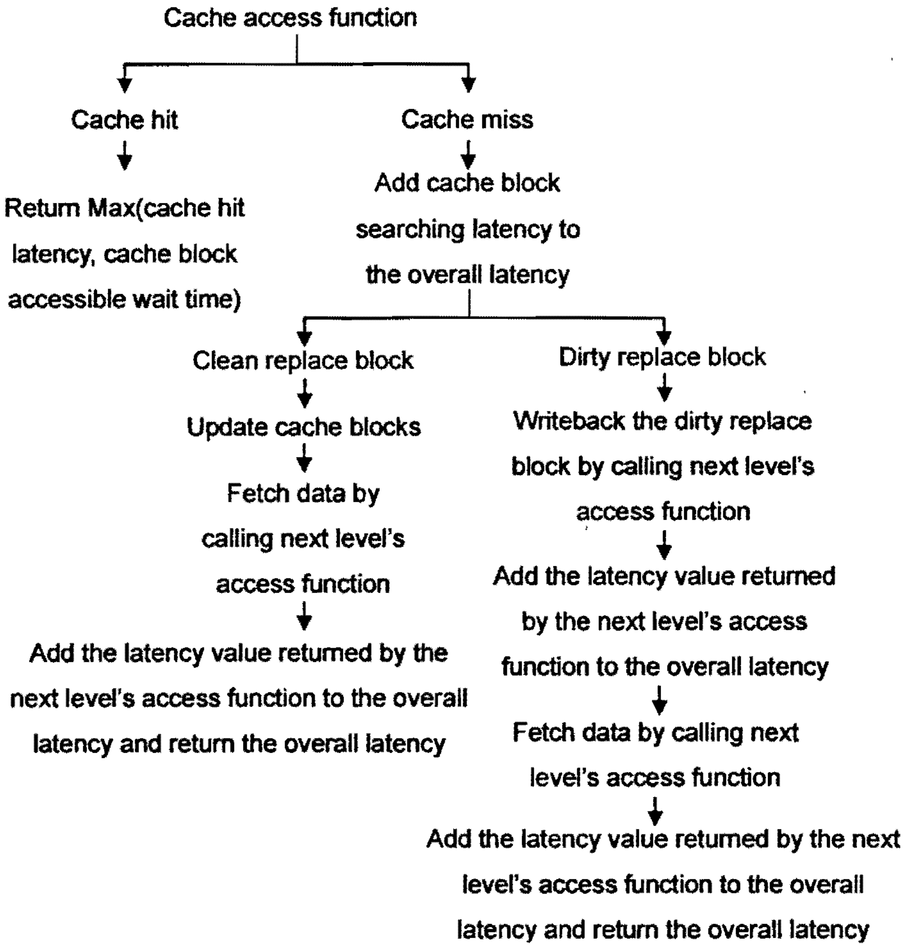

Cache access function
Cache hit
Cache miss
Return Max(cache hit latency, cache block accessible wait time)
Add cache block searching latency to the overall latency
Clean replace block
Dirty replace block
Update cache blocks
Fetch data by calling next level's access function
Add the latency value returned by the next level's access function to the overall latency and return the overall latency
Writeback the dirty replace block by calling next level's access function
Add the latency value returned by the next level's access function to the overall latency
Fetch data by calling next level's access function
Add the latency value returned by the next level's access function to the overall latency and return the overall latency

Figure 2.1: SimpleScalar cache access function work flow

Upon receiving the cache access latency, sim-outorder will generate a pipeline event. Pipeline event is a data structure which contains a register field and a time field. The register field in the pipeline event represents the destination of the data being loaded from or the source of the data being written to the cache. The time field is used to represent the clock cycle when the data loading or data writing is finished. It contains the value of T + latency of the cache access, where T is the clock cycle when the processor requests for the cache access. The pipeline event will be added to sim-outorder's event queue, which will be checked by sim-order in every simulated processor clock cycle. When the recorded clock cycle of the time field is reached, the pipeline event will be deleted by sim-outorder. The instructions that are utilizing the register specified by the pipeline event will then be considered as ready to be executed by the processor.

Figure 2.2 shows the function calling sequence of one particular cache access. In this example, sim-outorder accessed certain data from data L1 cache by calling the L1 cache access function (function A). The access generated a cache miss in L1 and a writeback is required. Function A then called the L2 cache access function (function B) to writeback the replacing block. The writeback block was found in the L2 cache, therefore function B could simply update the L2 cache's content and return a latency to function A as part of the total access latency.

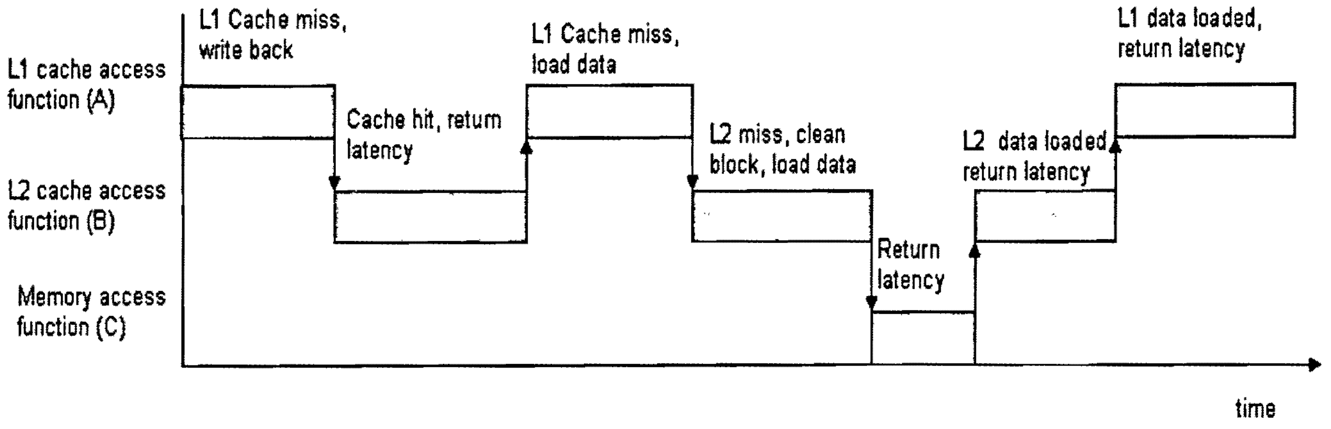

L1 cache access function (A)
L2 cache access function (B)
Memory access function (C)
L1 Cache miss, write back
Cache hit, retum latency
L1 Cache miss, load data
L2 miss, clean block, load data
Return latency
L2 data loaded return latency
L1 data loaded, return latency
time

Figure 2.2: SimpleScalar cache and memory access function sample calling sequence

After the writeback, function A must load data from the L2 cache, which caused a second call to function B. Unfortunately, that data load had also caused a cache miss in the L2 cache. This time, the L2 cache has a clean block and no writeback is needed. Therefore, function B must make a memory access by calling the memory access function (function C). At the end of function C's execution, the latency of loading the data from memory was calculated. This value would then be returned to function B and function A as part of the total accessing latency. Eventually, function A would sum up the writeback and data loading latencies and return the value to sim-outorder.

It is important to note that there exist several limitations in sim-outorder's memory hierarchy design. Firstly, when a writeback occurs, regardless it happens in the L1 or L2 cache, sim-outorder has an infinitely large write buffer with no access latency to temporarily store the cache block being written back. Hence, all cache and memory write actions will have an unrealistic latency of zero clock cycles.

Memory access latency = latency of first data column + (total amount of column to be accessed – 1) x latency of remaining column

In addition, sim-outorder's memory access latency calculation is relatively simple – the total memory access latency is equal to the latency of the first data column (including Tcmd, Trp, Trcd, Tcas, Tcwd and first column transmission) plus the latency of transmitting the remaining data columns multiplied by total amount of columns minus one (for the first column). From this formula, one can easily notice that sim-outorder's memory model is only valid when an outdated close page policy is used. Also, sim-outorder assumes memory accesses are always being scheduled to be far apart, therefore no memory access will post any burden to the data bus and causes future accesses to wait for the current access to finish.

Lastly, as all component access functions must return an exact latency to sim-outorder, it is impossible to implement a memory module which supports access reordering. This is because memory access reordering requires all accesses to be added and sorted within a queue first. Therefore the memory module cannot calculate the access latency until the access order is confirmed.

To conclude, we have discussed some internal details of the SimpleScalar Simulator tool set. We hope that this information will allow one to easily understand the operation of the SimpleScalar Similator and the design of our cache filter and memory module. In the next chapter, we will discuss our page interleaved DDR3 memory module for the SimpleScalar Simulator.

Chapter 3

The Page Interleaved DDR3 Memory Module for the SimpleScalar Simulator

In this chapter, we will first discuss the software design of the page interleaved DDR3 memory performance simulation model (will be referred as memory model) for the SimpleScalar simulator. Second, we will describe the operation of the model. At the end of the chapter, we will also simulate the system.

The remainder of this chapter will be organized as follows:

- The software modeling of the memory model (Section 3.1);
- The operation of the memory model (Section 3.2);
- Simulation and discussion (Section 3.3).

3.1 The software modeling of the memory model

To properly design the memory model, we did a study of the memory address format and the virtual memory address space (hard disk) of sim-outorder. As mentioned in chapter 2, sim-outorder models a 32bit computer. The virtual address space, or the hard disk, of this computer is 2GB in size (address starts from hexadecimal 0x00000000 to 0x7fffffff, according to the memory.c from the simulator's source files). This address range is perfectly addressable with 32bits because the hexadecimal range of 32bits starts from 0x00000000 to 0xffffffff. Therefore, sim-outorder represents memory addresses with a md_addr_t enumerated data type (user defined data type), which is C language's unsigned int (unsigned integer) type. And if one also studies the machine definition of sim-outorder (machine.h), he/she will notice that the virtual memory page size is defined as 4096 bytes (4KB).

We made two important decisions before we began the programming. First, we decided to directly map the 2GB of virtual memory space into 2GB of physical memory space with 4KB per memory row/page. This decision will greatly simplify the design of our memory model. Second, we decided to remove the infinite write buffer out of sim-outorder because of its unrealistic size and its zero access latency. In this context, sim-outorder must now wait for all cache/memory writebacks to complete and include the writeback latency in the latency calculations.

Our memory model has two components – the data and functional component. The data component, to be discussed in section 3.1.1, is the data structures that represent the memory hardware in the simulator. The functional component, covered in section 3.1.2, is used to describe the operation of the memory model.

3.1.1 The data components of the memory model

The data component of the memory model is written in the memory.h source file of SimpleScalar. It defines the new data structures which are necessary for the memory model to operate properly. It includes:

- modification to the mem_t data structure;
- a new channel_t data structure;
- a new bank_t data structure.

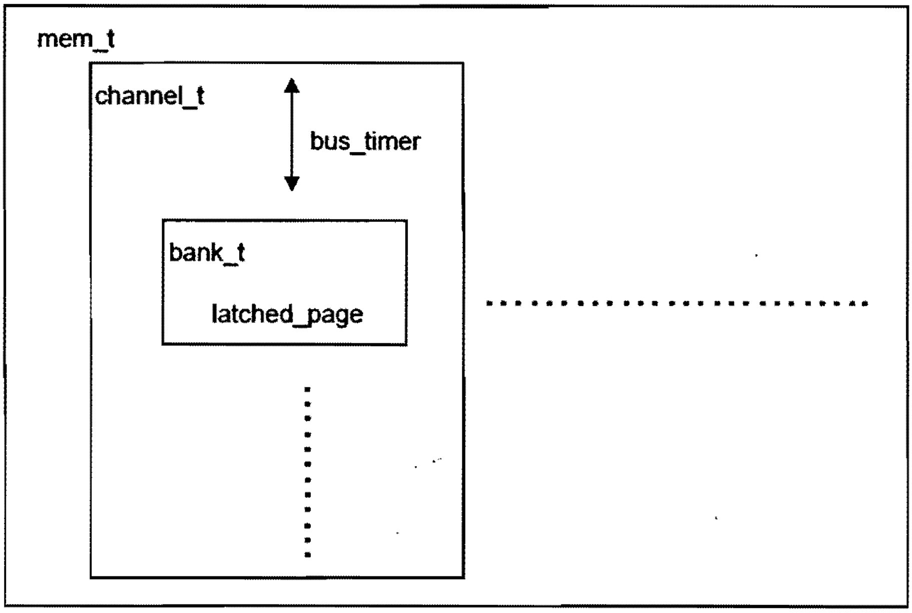


Figure 3.1: Graphical representation of the mem_t data structure

Modification to the mem_t data structure

A number of new data members were added to the mem_t data structure. The most noticeable addition is a channel_t pointer called channel. Channel is a pointer to a dynamically allocated array of channel_t data structures. This array of channel_t is used to represent one or multiple memory channels (the model can support up to two channels at this point).

Two floating point values known as mem_to_cpu_clk and cpu_to_mem_clk were also introduced to allow fast conversion between processor clock and memory clock. Given the clock speed of the processor and memory bus from the user input, the memory model will store the result of processor clock divided by the memory bus clock to mem_to_cpu_clk. The cpu_to_mem_clk, on the other hand, will hold the result of memory bus clock divided by the processor clock. To convert a value from memory clock to processor clock, the simulator can simply multiply the value with the mem_to_cpu_clk variable. A processor clock can also be converted easily by multiplying the value with the cpu_to_mem_clk variable.

Following the floating point values are several 64bit integer values (C language's long long type):

- access_count: a statistical value to store the total amount of memory access;
- unoverlaped_fp_read_clk and unoverlaped_fp_write_clk: statistical values to store the total amount of unoverlapped fast page read/write delay in processor clock (overlapped delay will be discussed in section 3.2);
- unoverlaped_filtered_read_clk and unoverlaped_filtered_write_clk: unused statistical values for debugging;
- unoverlaped_random_read_clk and unoverlaped_random_write_clk: statistical values to store the total amount of unoverlapped random read/write delay in processor clock.

Finally, there is a list of 32bit integer values (C language's int type) added to the mem_t data structure to store:

- the data bus width in bytes (bus_width);
- the total amount of memory channels (channels);
- bank –the total amount of banks per channel (bank);
- the column access strobe latency in memory clock cycles (Tcas);
- the row to column delay latency in memory clock cycles (Trcd);
- the row precharge latency in memory clock cycles (Trp);
- the burst length of the memory model (Tburst, a fixed value of 8 for DDR3);
- the write recover time in memory clock cycles (Twr);

- the rank to rank switching time and the data bus write to read switching time in memory clock cycles (Trtrs);
- the command duration time in memory clock cycles (Tcmd);
- the column write delay in memory clock cycles (Tcwd);
- the column to column delay in memory clock cycles (Tccd);
- the size of a bank(bank_size);
- the memory mapping, only page interleaving is supported in the current implementation and other memory mapping can be added in the future (map_type);
- the memory type, only DDR3 is supported in the current implementation but SDRAM can also be supported with minor changes to the code (mem_type).

The channel_t data structure

A new channel_t data structure was introduced to represent a memory channel. The most important data member of the data structure is a bank_t (to be discussed later) pointer. Same as the channel_t pointer found in the mem_t data structure, the bank_t pointer points to a dynamically allocated bank_t array to represent the collection of banks within a memory channel.

Following the bank_t pointer is a SimpleScalar enumerated mem_cmd data type variable known as previous_command. According to the memory.h source file, the mem_cmd enumerated data type is a flag to represent read or write memory commands/activities. The previous_command variable was added to represent the previous bus activity, which is important for memory access latency calculations (to be discussed in section 3.2).

Two 32bit integer values, prev_burst and prev_accessed_bank, were added to handle different memory access situations (to be discussed in section 3.2) during the latency calculation. The prev_burst value keeps the amount of processor time taken to make the previous data transmission. The prev_accessed_bank holds the previous accessed bank number/array index. Also, it is important to note that the C language uses pointer arithmetic to access the array elements, therefore the indexes starts from 0 to size of the array – 1.

Finally, there is also a list of 64bit integer values from the channel_t data:

- bus_timer: the value recorded by this variable represents the processor cycle when the data bus of this memory channel will become idle.
- cmd_timestamp: this value records the processor cycle when the last memory access command was issued by the memory controller.
- burst_timestamp: a value used to represent the processor cycle when the last data transmission began.
- access_count: a statistical value to record the total amount of memory access to the memory channel.
- actual_transfer: a statistical value to keep the total amount of data columns being transferred by the memory channel.
- row_hit: a statistical value to count the total amount of row hit happened in the memory channel.
- fp_reads and fp_writes: two statistical values to record the total amount of fast page read / write access to the memory channel.
- random_reads and random_writes: two statistical values to record the total amount of random read/write access to the memory channel.

- channel_unoverlaped_fp_read_clk and channel_unoverlaped_fp_write_clk: two statistical values to hold the total amount of unoverlapped fast page read / write delay in processor cycles.
- channel_overlaped_fp_read_clk and channel_overlaped_fp_write_clk: two statistical values to record the total amount of overlapped fast page read / write delay in processor cycles (overlapped delay will be discussed in section 3.2).
- channel_unoverlaped_random_read_clk and channel_unoverlaped_random_write_clk: two statistical values to keep the total amount of unoverlapped random read / write latency in processor cycles.
- channel_overlaped_random_read_clk and channel_overlaped_random_write_clk: two statistical values to keep the record of the total amount of overlapped random read / write latency in processor cycles.
- filtered_reads, filtered_write, channel_overlaped_filtered_read_clk, channel_overlaped_filtered_write_clk, channel_unoverlaped_filtered_read_clk and channel_unoverlaped_filtered_write_clk: unused statistical values for debugging purposes.

The bank_t data structure

The bank_t data structure was defined to represent a memory bank of the memory model. There are only two values stored in the data structure – the latched_page and the ras_timestamp. The latched_page variable is an unsigned integer value to represent the previously opened row of the memory bank. The ras_timestamp is the record of the processor cycle when the last row precharge happened to the memory bank.

3.1.2 The functional components of the memory model

The functional component of the memory model was implemented in the memory.c source file. It has a number of functions and we categorized them into three classes: setter functions to initialize the values of the data components; memory access latency function to define how the memory model should operate and helper functions to aid latency calculations. In this sub-section, we will list and briefly describe the functions.

Setter functions

There are six setter functions to allocate and initialize the memory model's data structures, they are:

- set_mem_size: a function that dynamically allocates the channel_t and bank_t data structure arrays and calculates the size of each memory bank based on the user specified amount of memory channel/s and bank/s per channel.
- set_bus_width: a function that takes the user inputted memory data bus width (in bytes) and initializes the mem_t data structure's bus_width variable.
- set_clk: this function takes the user inputted value for processor clock and memory data bus clock and compute the mem_t data structure's mem_to_cpu_clk and cpu_to_mem_clk variables.
- set_map: a function that takes the user inputted value for memory mapping and initializes the mem_t data structure's map_type variable. We intended to allow bank interleaving and cache line interleaving mode to be added in the future as extensions to the work, but in the current implementation, only page interleaving is supported.

Note: Some of the functions will have detailed explanation in section 3.2 as they are closely related to the operation of the memory model.

- set_type: this function sets the mem_type variable of the mem_t data structure with the user inputted value. This function was added to allow future memory model extensions such as SDRAM and DDR4 to be added with minor code modification.
- set_latency: this function sets the following latencies with the user inputted values: Tcas, Trcd, Trp, Tras, Tburst, Twr, Trtrs, Tcmd, Tcwd and Tccd [2].

Memory access latency calculation function

The memory access latency function (memory_access_latency) was implemented to calculate the latency of a memory access, based on the current state of the mem_t data structure. (refer to section 3.2.3 for more detail).

Helper functions

We have also defined some helper functions to aid latency calculations:

- max: max is a function to compare two integer values and return the bigger value of the two.
- is_latched: given a starting address, its corresponding channel and bank, this function returns a non-zero value if the address generates a row hit (the row of the starting address is the same as the row being latched by the given channel's bank), zero otherwise.
- get_mem_page: given a starting address, this function consults the page table and returns the physical memory page/row number which the address is located at (refer to section 3.2.2 for more detail).
- get_mem_actual_bank: based on the page/row number received from get_mem_page, this function returns the physical bank number which contains the page (refer to section 3.2.2 for more detail).

- get_mem_channel and get_mem_bank: these two functions will take the physical bank number returned by get_mem_actual_bank and calculate the channel array index and bank array index of the mem_t data structure respectively (refer to section 3.2.2 for more detail).

3.2 The operation of the memory model

In this section, we will discuss various aspects related to the operation of the memory model. These aspects include the overlapped memory access scheduling, memory addressing and most importantly, the memory access latency calculation.

3.2.1 Overlapped memory access scheduling

Back in section 3.1.1, several statistical variables from the mem_t and channel_t data structures to record the overlapped and unoverlapped access latencies were introduced. These latencies are related to a specific memory access scheduling technique known as the overlapped memory access scheduling [2]. In high speed memory systems with multiple banks, memory accesses can be pipelined into different phases. The phases are: react to an access request (Tcmd), row precharge (Trp), data ready (Trcd), column select (Tcas or Tcwd), data transmission and data recovery (Twr, for write only).

If the newly requested memory access is not reading from/writing to the same memory bank as the previous access, the memory controller can schedule the new access to start processing, up to the column selecting phase, before the previous access is completed (an example is shown in Figure 3.4). This type of scheduling is overlapped memory access scheduling as part of the memory accesses are being overlapped. This is possible because row precharging and column selecting actions will not affect the previous access as they are utilizing different memory banks. But it is important to note that the command reaction and the actual data transmission can not be overlapped within a memory channel because each memory channel has only one data and command bus for data and command transmission.

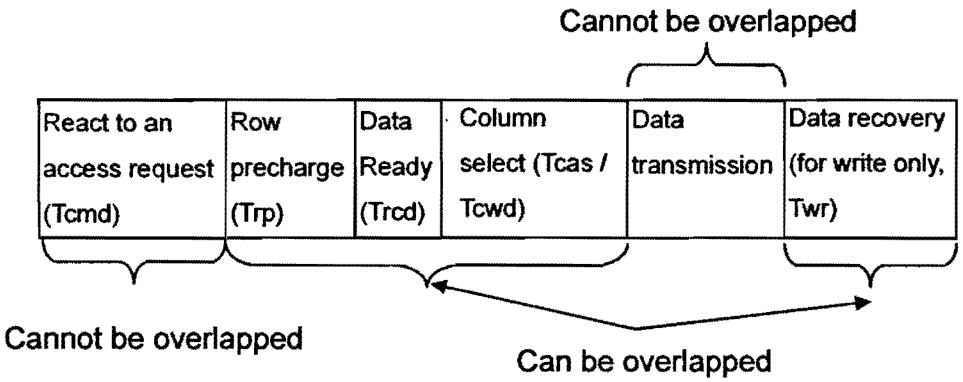


Figure 3.2: The overlappable and unoverlappable phases of a random memory access

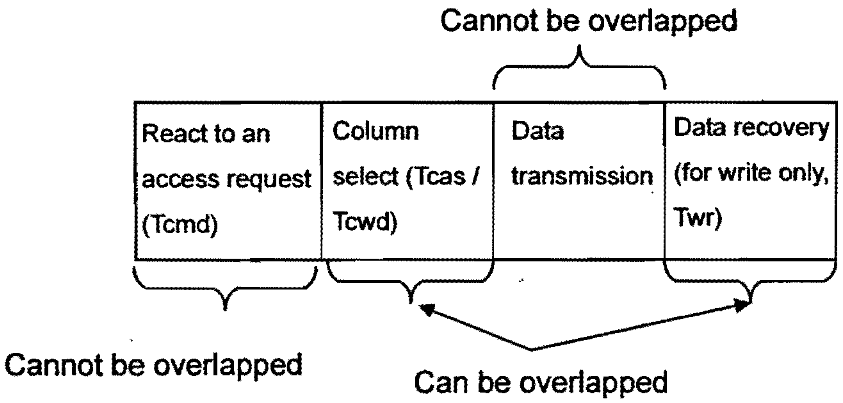

Cannot be overlapped
React to an access request (Tcmd)
Column select (Tcas / Tcwd)
Data transmission
Data recovery (for write only, Twr)
Cannot be overlapped
Can be overlapped

Figure 3.3: The overlappable and unoverlappable phases of a fast page memory access

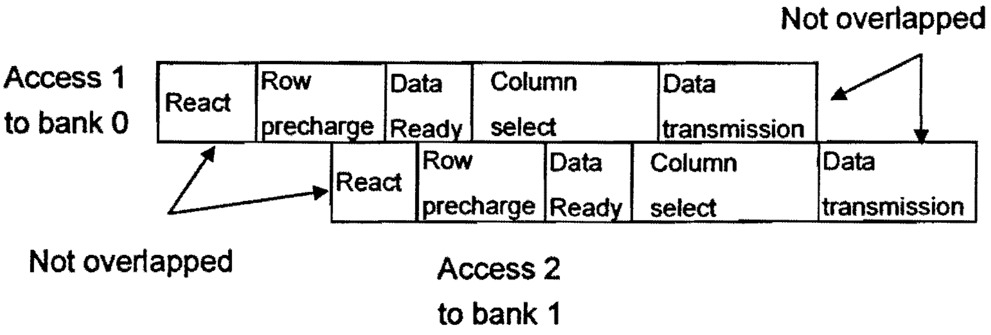


Figure 3.4: Overlapped memory access scheduling example

3.2.2 Memory addressing

According to the DDR3 standards, a memory channel has at least 8 banks. With our memory model supporting 2GB of single and dual channel memory with up to 16 banks per channel, the memory address has the following formats:

Dual channel, 8 banks per channel:

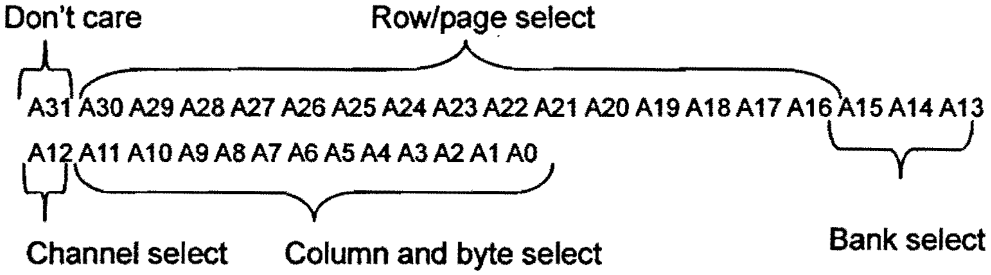

Don't care
Row/page select
A31 A30 A29 A28 A27 A26 A25 A24 A23 A22 A21 A20 A19 A18 A17 A16 A15 A14 A13
A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
Channel select
Column and byte select
Bank select

Figure 3.5: 32bit dual channel 8 bank memory address format

Dual channel, 16 banks per channel:

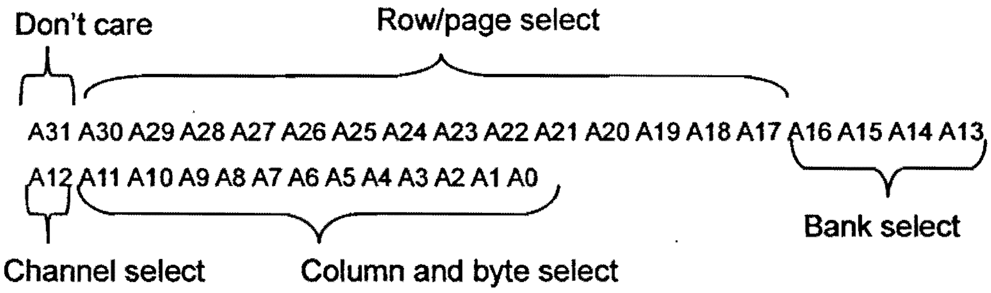

Don't care
Row/page select
A31 A30 A29 A28 A27 A26 A25 A24 A23 A22 A21 A20 A19 A18 A17 A16 A15 A14 A13
A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
Bank select
Channel select
Column and byte select

Figure 3.6: 32 bit dual channel 16 bank memory address format

Single channel, 8 banks:

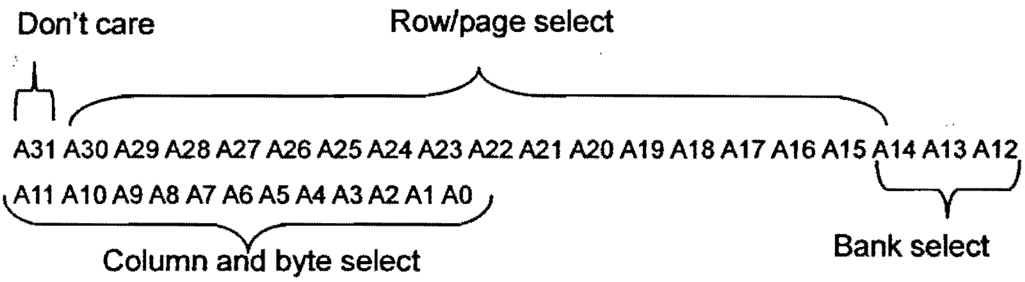

Don't care
Row/page select
A31 A30 A29 A28 A27 A26 A25 A24 A23 A22 A21 A20 A19 A18 A17 A16 A15 A14 A13 A12
A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
Column and byte select
Bank select

Figure 3.7: 32 bit single channel 8 bank memory address format

Single channel, 16 banks:

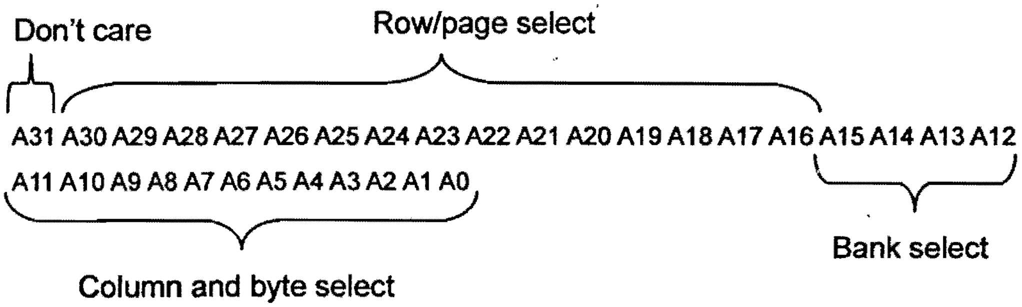

Don't care
Row/page select
A31 A30 A29 A28 A27 A26 A25 A24 A23 A22 A21 A20 A19 A18 A17 A16 A15 A14 A13 A12
A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
Bank select
Column and byte select

Figure 3.8: 32 bit single channel 16 bank memory address format

In our simulator, the decoding in the software can be achieved by either the logical approach (bit masking) or the mathematical approach (integer division and modulo). The benefit of the logical approach is the ease of implementation, but it is more restrictive as it requires the programmer to hard-code the bit masks. Keeping the future expendability of the simulator in our mind, we decided to take the mathematical approach and implemented the *get_mem_page*, *get_mem_actual_bank*, *get_mem_channel* and *get_mem_bank* functions.

Upon receiving a memory address, the memory model decodes the address by first calling the *get_mem_page* function. This function has the following function signature:

```
unsigned int get_mem_page(struct mem_t *mem, md_addr_t addr);
```

The mem variable is a mem_t pointer, which points to sim-outorder's mem_t data structure. The second parameter, addr, is the target memory address to be translated into page/row number.

The get_mem_page function takes the requested address and search for the physical page number from the page table. The page number is formed by dividing the memory address with the page size (4096 Bytes). If the page is not found from the page table, the *get_mem_page* function will divide the value and return the result as the physical page number. Note that the physical page number is unique across the memory space, the memory model can simply take the physical page number as the row identifier with the mathematical approach.

With the physical page number available, the memory model calculates the physical bank number where the address located at with the *get_mem_actual_bank* function. This function has the following function signature:

```
int get_mem_actual_bank(struct mem_t* mem, md_addr_t addr);
```

The addr value is the memory address that needs to be converted to a physical bank number.

The get_mem_actual_bank function takes the remainder of the division between the physical page number and the multiplication result of total amount of channel and amount of banks per channel.

*Physical bank number = physical page number MOD (total amount of channels * total amount of banks per channel)*

Since page interleaving distributes memory rows/pages in a zigzag manner across the banks and channels, he memory model must further process the physical bank number with the *get_mem_channel* and *get_mem_bank* functions to obtain the channel number and the bank number of the channel.

The *get_mem_channel* function has the following function signature:

```
int get_mem_channel(struct mem_t* mem, int bank);
```

The get_mem_channel function calculates the channel number by taking the remainder of the physical bank number divided by the total amount of channels.

Channel number = physical bank number MOD total amount of channels

Note: The bank variable is the physical bank number returned by the get_mem_actual_bank function.

Similarly, the *get_mem_bank* function has the following function signature:

int get_mem_bank(struct mem_t* mem, int bank);

The get_mem_bank function calculates the bank number of the channel by dividing the physical bank number with the total amount of channels.

Bank number of a channel = physical bank number / total amount of channels

We have included the following address conversion example with 4 memory channel and bank configurations to prove that our mathematical memory address decoding method is correct for all memory configurations:

Suppose the memory address to be accessed is 366882:

The binary form of 366882 is:

00000000000001011001100100100010

Following the bit naming convention from Figure 3.7, for **single channel 8 banks**:

The column and bank select bits are A11 to A0, which are "100100100010";

The bank select bits are A14 to A12, which are "001";

The row select bits are A30 to A15, which are "0000000000001011".

Therefore, we expect this address belongs to bank 1.

Using our mathematical approach:

Physical page number = address / page size (4KB)

= 366882 / 4096 = 89

Note: the mathematical approach uses the C language's integer division and modulo operations. Therefore both the division result and the remainder will be integers.

Physical bank number = physical page number MOD (total amount of channel * total amount of banks per channel)

= 89 MOD (1 * 8) = 89 MOD 8 = 1

Channel number = physical bank number MOD total amount of channels

= 1 MOD 1 = 0

Bank number = physical bank number / total amount of channels

= 1 / 1 = 1

By applying the mathematical approach to the single channel 8 bank scenario, the result is bank 1. Therefore we can conclude that the result of mathematical approach matched the logical approach's result.

Single channel 16 banks:

The column and bank select bits are A11 to A0, which are "100100100010";

The bank select bits are A15 to A12, which are "1001";

The row select bits are A30 to A16, which are "000000000000101".

Therefore, we expect this address belongs to bank 9.

Using our mathematical approach:

Physical page number = address / page size (4KB)

= 366882 / 4096 = 89

Physical bank number = physical page number MOD (total amount of channel * total amount of banks per channel)

= 89 MOD 16 = 9

Channel number = physical bank number MOD total amount of channels

= 9 MOD 1 = 0

Bank number = physical bank number / total amount of channels

= 9 / 1 = 9

By applying the mathematical approach to the single channel 16 bank scenario, the result is bank 9. The result of our mathematical approach is matching the logical approach's outcome again.

Dual channel 8 banks:

The column and bank select bits are A11 to A0, which are “100100100010”;

The channel select bit is A 12, which is “1“;

The bank select bits are A15 to A13, which are “100”;

The row select bits are A30 to A16, which are “000000000000101”.

Therefore, we expect this address belongs to channel 1's bank 4.

Using our mathematical approach:

Physical page number = address / page size (4KB)

= 366882 / 4096 = 89

Physical bank number = physical page number MOD (total amount of channel * total amount of banks per channel)

= 89 MOD (2*8) = 89 MOD 16 = 9

Channel number = physical bank number MOD total amount of channels

= 9 MOD 2 = 1

Bank number = physical bank number / total amount of channels

= 9 / 2 = 4

By applying the mathematical approach to the dual channel 8 bank scenario, the result is channel 1's bank 4. Therefore mathematical approach is also valid with the dual channel 8 bank configuration.

Dual channel 16 banks:

The column and bank select bits are A11 to A0, which are “100100100010”;

The channel select bit is A 12, which is “1“;

The bank select bits are A16 to A13, which are “1100”;

The row select bits are A30 to A17, which are “00000000000010”.

Therefore, we expect this address belongs to channel 1’s bank 12.

Using our mathematical approach:

Physical page number = address / page size (4KB)

= 366882 / 4096 = 89

Physical bank number = physical page number MOD (total amount of channel * total amount of banks per channel)

= 89 MOD (2 * 16) = 89 MOD 32 = 25

Channel number = physical bank number MOD total amount of channels

= 25 MOD 2 = 1

Bank number = physical bank number / total amount of channels

= 25 / 2 = 12

Finally, by applying the mathematical approach to the dual channel 16 bank scenario, the result is channel 1’s bank 12, which is matching our expected results. Through the proving process similar to mathematic induction, we conclude that our mathematical memory address decoding method is correct for all memory configurations.

Note: For the remainder of this document, we will refer to each of the memory latencies mentioned in section 3.1.1 by their short form (e.g. command duration time = Tcmd)

3.2.3 The memory_access_latency function

The memory_access_latency function models the memory operations, calculates the latency of each access and returns the value to sim-outorder. The function signature of memory_access_latency is shown below:

```
unsigned int memory_access_latency(struct mem_t* mem, int chunks, md_addr_t
                    addr, enum mem_cmd cmd, tick_t now);
```

The mem parameter is a mem_t pointer which points to the mem_t data structure of sim-outorder. The second parameter, chunks, is the total amount of bytes the processor/cache is requesting for the memory access (in sim-outorder, it is the L2 cache block size since the L2 cache is the only cache that requests for memory access). The addr parameter is the first address which the processor/cache is requesting. The last parameter, now, is the processor cycle when the processor/cache requested for the memory access.

The algorithm behind the memory_access_latency function

Before the description of the algorithm begins, one must note that the latency value returned by memory_access_latency is not simply the sum of the timing parameters (Tcmd, Trp, Trcd, Tcas, Tcwd and data transmission), but a latency relative to the 'now' parameter. That is, the value returned by the function can be significantly larger than the sum of those timings. This is caused by the fact that when the processor/cache requests for a memory access, the memory model can either be idle or busy servicing the pervious access/s. If the memory is servicing another access, it must wait for the previous access to complete before servicing the current access.

The most important goal of the memory_access_latency function (latency function) is to calculate the earliest possible starting processor cycle (start time) for the memory access requested by the processor/cache. When the latency function is called, it will first call the get_mem_actual_bank, get_mem_bank, get_mem_channel and is_latched functions to determine whether the access is a random type or fast page type.

Once the access type is determined, the latency function will check whether the access is the first memory access of the simulation (bus_timer = 0). If so, the start time can simply be set as "now" and the access type will be limited to random access (the first access of a simulation has to be random access. Also, when the mem_t data structure is initialized, the latched row/page of all memory banks is set as row 0, so we must override the result of the is_latch function).

If the access is not the first access of the simulation, the latency function compares the value of the "now" parameter with the channel's previous command start time (cmd_timestamp + Tcmd) and takes the larger value as the temporary start time. This is because an access cannot start earlier than the previous access. Note that the temporary start time will be increased (shifted further away from "now") as the function handles other situations.

If the access type was previously determined as *fast page mode*, the latency function will store the sum of the corresponding timing parameters (Tcmd + Tcas for read OR Tcwd for write) to an integer variable "latency" and detect the following situations:

Consecutive reads

If the previous access is a read operation and the current access is also a read, the function will first compare the value of the previous data transmission time (prev_burst from channel_t data structure) and the column to column delay (Tccd) and store the larger value of the two values to a temporary variable known as "temp". The latency function will then ensure the minimum distance between the current read's start time and the previous read's start time is "temp" by incrementing the temporary start time of the current access. This action allows sufficient time for the memory to complete the data transmission and column selection.

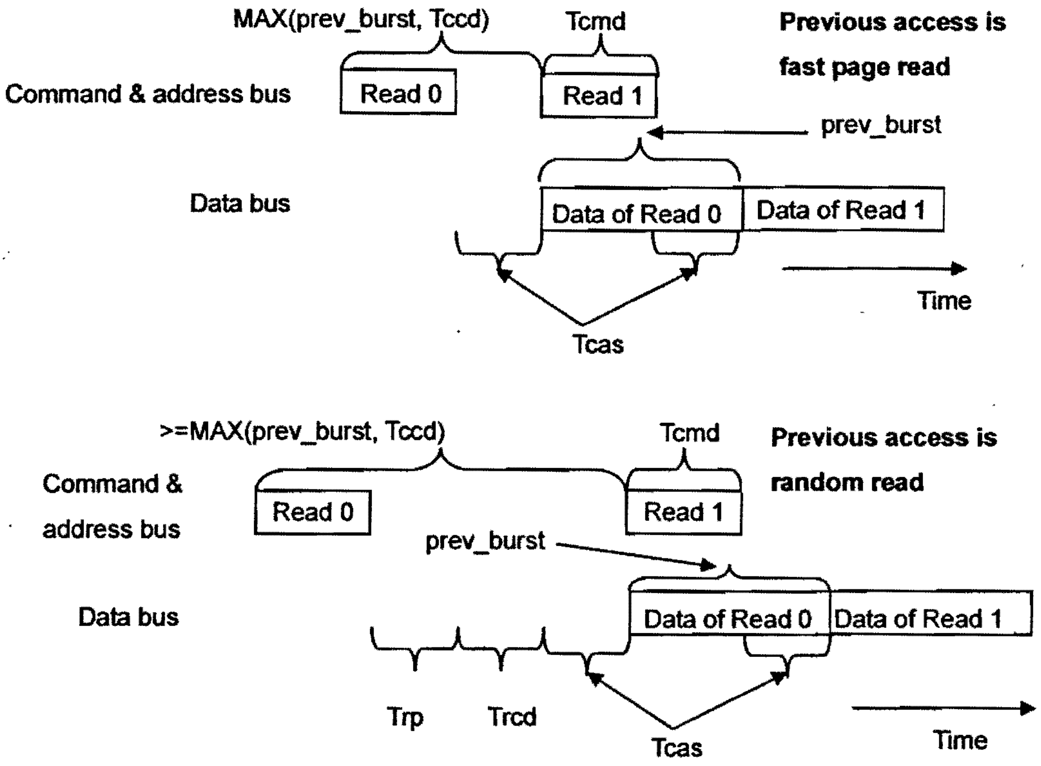


Figure 3.9: Consecutive read timing (current access is fast page read)

Fast page write following a fast page read

If the previous access is a fast page read and the current access is a write, the latency function must ensure the timing distance between the previous read and the current write is Tcas + Trtrs + Tcwd + prev_burst. Note that the Trtrs is added as a timing bubble to eliminate the conflict with the internal data movement of the read access.

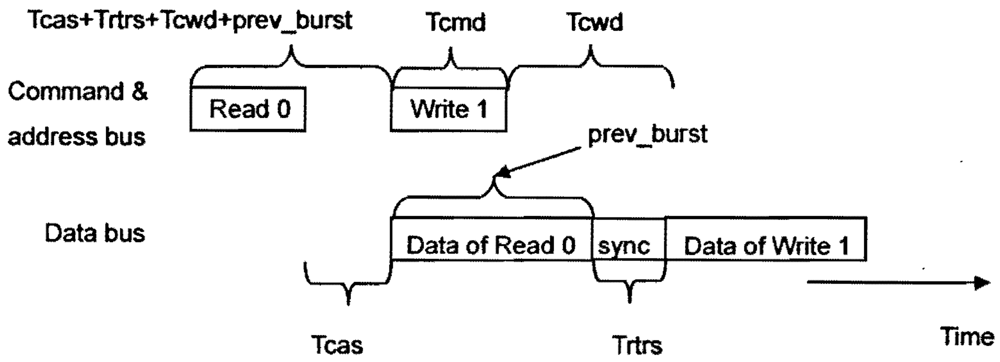


Figure 3.10: Write following a fast page read timing (current access is fast page write)

If the latency function determined the access type is *random access*, the function will store the sum of the corresponding timing parameters (Tcmd + Trp + Trcd + Tcas for read OR Tcwd for write) to the "latency" variable and record the latched row/page to the bank's latched_page variable. Also, if the latency function determines the current memory access is the first access of the simulation, it will skip the following start time calculation and jump to the *actual data transmission latency calculations.*

Because random accesses require a row precharge, the latency function must check the last ras_timestamp of the bank. If the timing difference between temporary start time and the last ras_timestamp (temporary start time – ras_timestamp) is less than the value of the Tras of the mem_t data structure, the latency function will increment the temporary start time with the value: Tras – (temporary start time – ras_timestamp).

Random access to the same bank as the previous write

After handling the Tras timing requirement, the latency function checks whether the previous access was accessing the same memory bank and the access was a write. If so, the memory must allow a minimum timing distance of Twr to allow the row latch to write the updated values back to the memory cells.

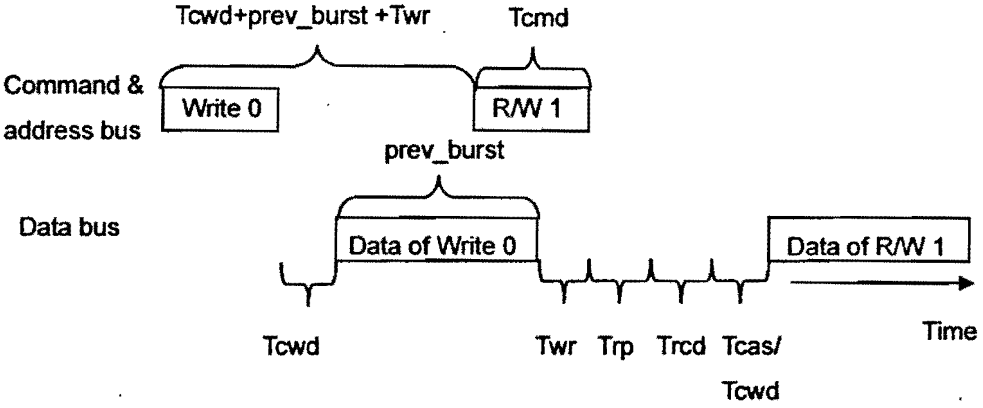


Figure 3.11: Random access to the same bank as the previous write timing

When the latency function finishes calculating the temporary start time of the current access, it will begin the *actual data transmission latency calculations* and record the followings:

- The bank number of the current access to the prev_bank variable of the channel_t data structure;
- The temporary start time of the current access to the cmd_timestamp variable of the channel_t data structure and to the ras_timestamp of memory bank (if row precharge needed);
- The current access command (Read or Write) to the previous_command variable of the channel_t data structure;
- The time when data transmission begins (cmd_timestamp + "latency") to the burst_timestamp variable of the channel_t data structure;

Actual data transmission latency calculation

The actual data transmission latency calculation starts by dividing the "chunks" parameter with the mem_t data structure's bus_width variable to obtain the total amount of data columns to be transmitted. Once the total amount of data columns is determined, the latency function will divide the amount by two and store the result to a temporary variable known as access_transfer (DDR3 memory can transmit twice per memory clock and transmit one data column per transmission).

At this point, the latency function is still in terms of memory clock cycles. Therefore the latency function will convert the result to processor clock cycles by multiplying the values ("latency" and access_transfer) with mem_t's mem_to_cpu_clk variable. With every value converted to processor clock cycle, the latency function will store the value of access_transfer to the channel's prev_burst variable for future latency calculations. The latency function will also store the value of start time + "latency" + access_transfer to the bus_timer variable of the channel because the data bus will be busy until the newest memory access (the current access) finishes.

Finally, the last action of the latency function is to calculate the memory access latency that is relative to the "now" parameter. This is completed by returning the value of bus_timer – "now".

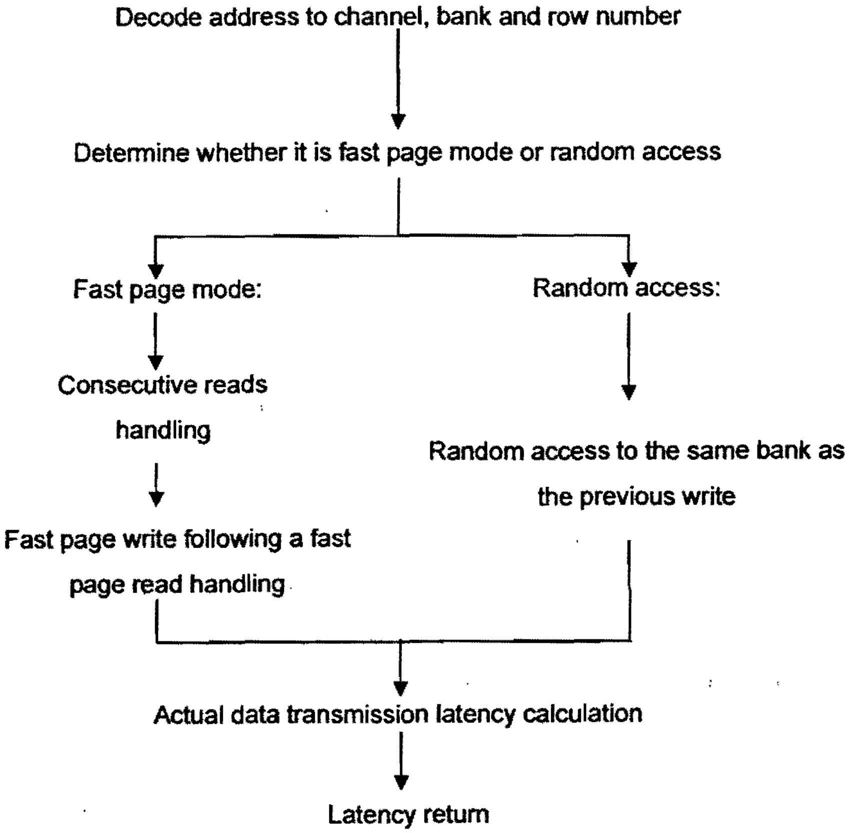

Decode address to channel, bank and row number
Determine whether it is fast page mode or random access
Fast page mode:
Random access:
Consecutive reads handling
Random access to the same bank as the previous write
Fast page write following a fast page read handling
Actual data transmission latency calculation
Latency return

Figure 3.12: memory_access_latency function work flow

3.3 Simulation and discussion

In this section, we will perform a memory channel and bank test to verify our memory module's behavior under different memory channel and bank settings. In this test, we will use the gcc benchmark from the SPEC CPU 2000 benchmark suite for the simulations and monitor the system performance and memory row hit rate under the following memory configurations:

- 1 channel 4 banks;
- 1 channel 8 banks;
- 2 channels 2 banks (4 banks in total);
- 2 channels 4 banks (8 banks in total);
- 2 channels 8 banks (16 banks in total).

Note that we will set all of the simulation settings, except the cache and memory related settings, with the default settings of sim-outorder. Also, our cache settings will be consistent among each sub-section to allow precise comparison between data points. Finally, as mentioned in section 3.1, our custom version of sim-outorder does not include the write buffer and memory access reordering capabilities.

Simulation settings

Table 3.1: Simulation settings for channel and bank test

Instruction L1 cache size	32KB
Data L1 cache size	32KB
L1 cache latency	1 processor clock
L2 cache size	1MB
L2 cache latency	10 processor clocks
Cache block size	64B
L1 cache set associativity	4
L2 cache set assocaitivity	8
Custom DDR3 memory model's memory bus speed (MHz)	800 (DDR3 1600)
Amount of memory banks	4, 8
Amount of memory channels	1, 2
Tcas, Trcd, Trp, Tras (memory clocks)	7, 7, 7, 21
Trtrs, Tcwd, Tcmd, Twr, Tccd (memory clocks)	1, 7, 1, 5, 4
Our DDR3 memory model's processor clock (MHz)	3200

Note: To verify the statistical values generated by our custom sim-outorder, we have also included a sample simulation output file verification in appendix A.2 on page 127.

Row hit rates and row hit improvement

Row hit rates

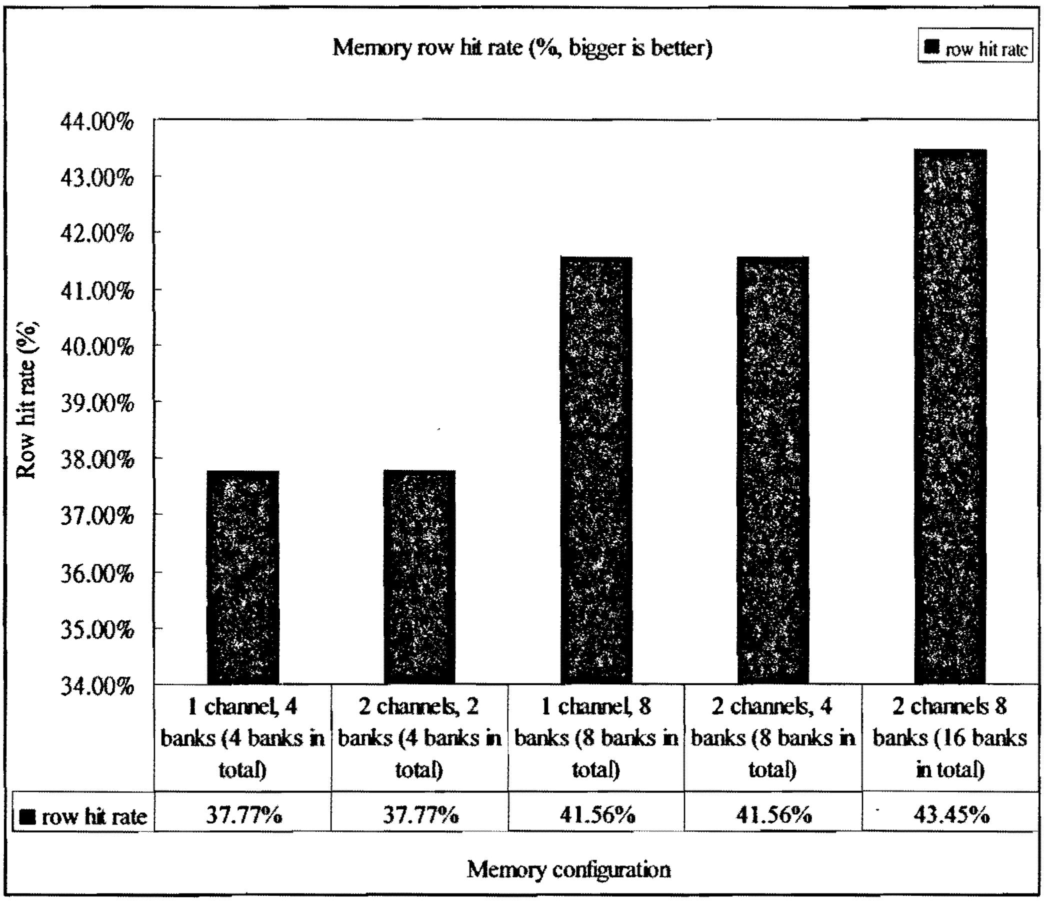
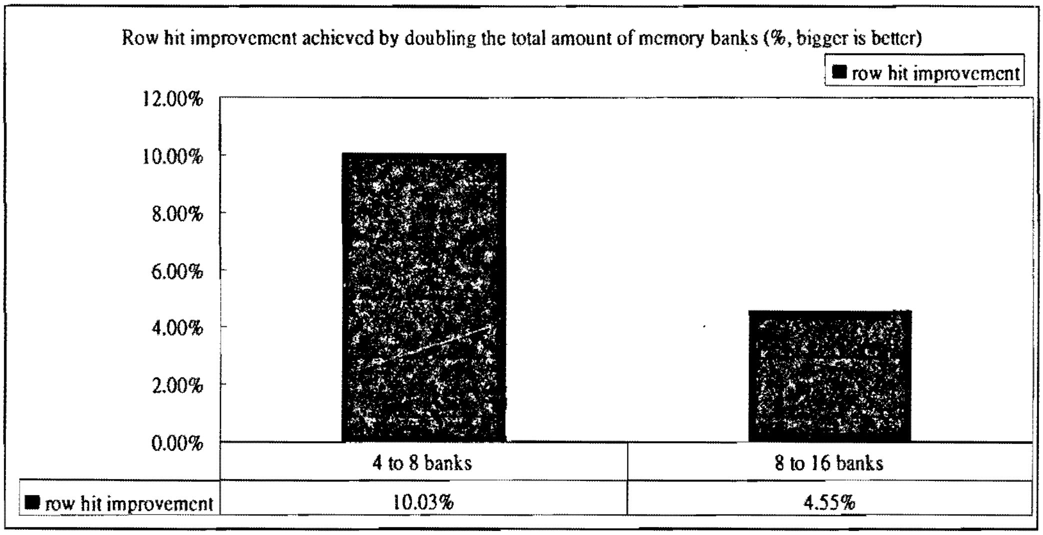


Figure 3.13: Memory row hit rates of the gcc simulation

Note: Please note that there has never been any DDR3 memory being manufactured with only 2 and 4 memory banks. Therefore, the simulations with 2 and 4 memory banks are never intended to reflect real world memory performance; instead, they are only used to illustrate the behavior of the memory row hit rate when the total amount of memory banks is being doubled. Finally, when considering the row hit rates of the multi channels setups, we must take the average row hit rate across the channels as the overall memory row hit rate.

Memory row hit rate improvement achieved by doubling the total amount of memory banks



	4 to 8 banks	8 to 16 banks
■ row hit improvement	10.03%	4.55%

Figure 3.14: Row hit improvement when total amount of memory banks is doubled

From the row hit rate plot, one can notice the followings:

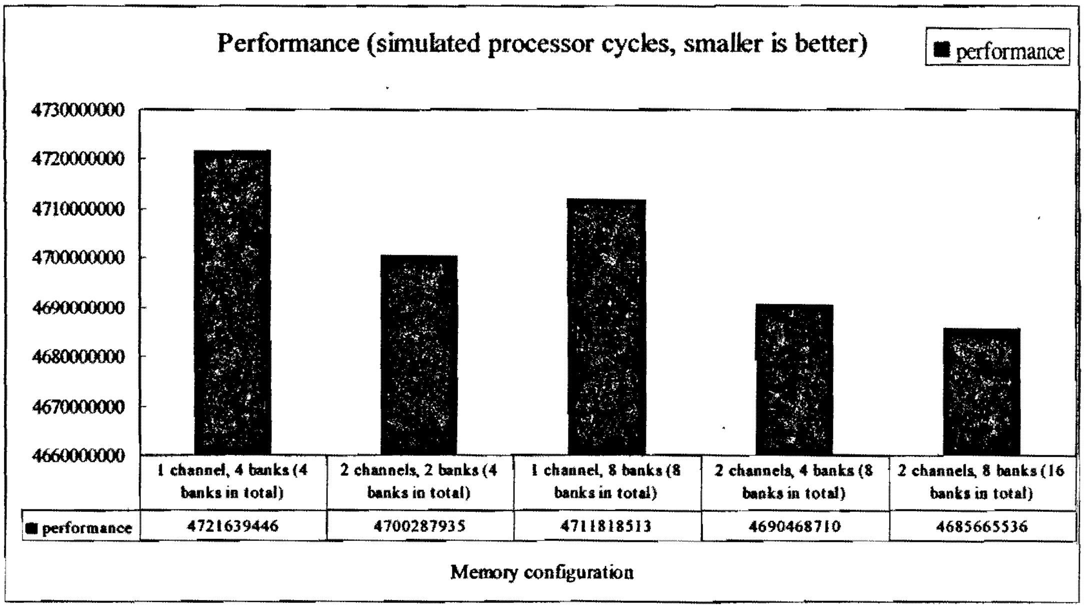
- The row hit rate improvement is not sensitive to the total amount of memory channels;
- The row hit improvement of doubling the total amount of memory banks (from 1 channel 4 banks OR 2 channels 2 banks to 1 channel 8 banks OR 2 channels 4 banks respectively) is consistent.
- The row hit rate improvement tends to drop as the total amount of memory banks increase.

The first and the second facts are the most important findings to prove our memory module is able to generate consistant results amoung different memory configurations - regardless the total amount of memory channels, if the total amount of memory banks is fixed, the memory row hit rate and memory row hit improvement should be consistent because the application's degree of locality is constant.

To explain the third finding, one can consider the multi bank main memory as an index of a set associative cache where the total amount of memory banks is the set associativity and each row latch is a cache block. If one keeps increasing the associativity or the memory bank count, it will eventually reach the limit of the set associative or multi bank design where additional set / bank will not benefit the overall hit rate. Therefore the row hit rate improvement will not double as the total amount of memory banks doubles. Instead, the row hit rate improvement should be a function of total amount of memory banks and the application's degree of locality. Hence it should converge to a value as the total amount of available row latches reaches the limit of the program's memory row request limit.

Simulated performance and performance comparison under different memory configurations

Simulated performance



	1 channel, 4 banks (4 banks in total)	2 channels, 2 banks (4 banks in total)	1 channel, 8 banks (8 banks in total)	2 channels, 4 banks (8 banks in total)	2 channels, 8 banks (16 banks in total)
■ performance	4721639446	4700287935	4711818513	4690468710	4685665536

Figure 3.15: gcc simulated performance

Performance improvement achieved by increasing the amount of memory channels

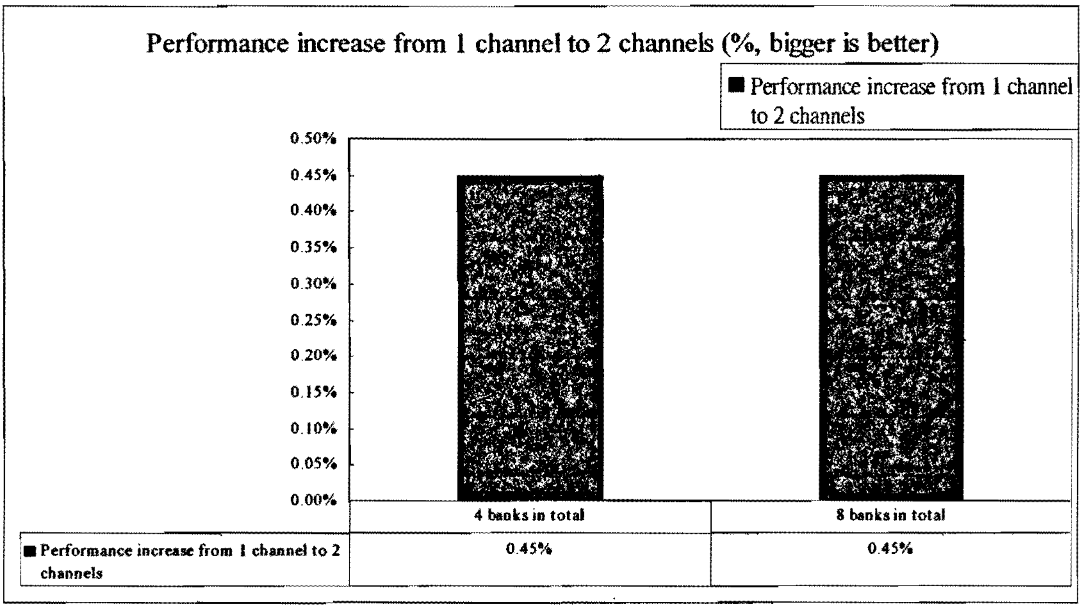


Figure 3.16: gcc performance improvement achieved by increasing the amount of memory channels

Performance improvement achieved by doubling the total amount of memory banks

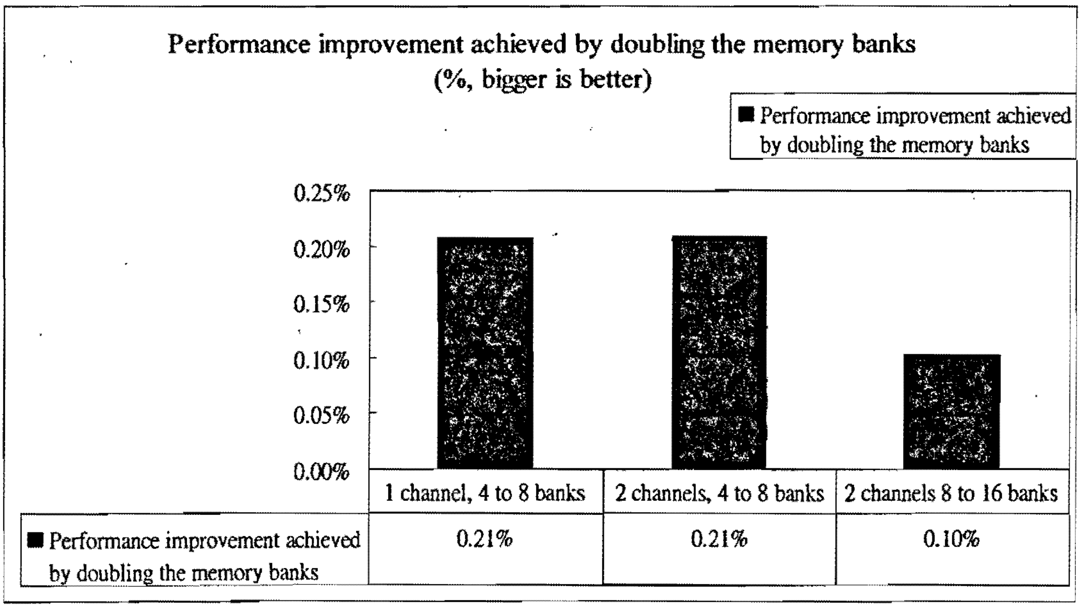


Figure 3.17: gcc performance improvement achieved by doubling the total amount of memory banks

From the performance plots, we have made the following observations:

- The performance gain of adding an additional memory channel is much bigger than doubling the amount of total memory banks;
- Regardless of the total amount of channels, the performance gain of doubling the total amount of memory banks from 4 to 8 is consistent;
- The benefits of doubling the total amount of memory banks are diminishing if one compares the improvement from 4 to 8 banks and from 8 to 16 banks.

The first finding can be considered as the benefit of an extra memory data bus. With an additional memory data bus, the average amount of access to each data bus is reduced. Therefore, the average amount of memory access wait time caused by incomplete previous memory accesses can be reduced, yielding better performance than simply doubling the total amount of memory banks per memory channel.

The second and third observations are, once again, proofs of the relationship between the total amount of memory banks and the limit of the multi memory bank design. As the row hit rate of the configurations with a total 4 memory banks should be consistent, one should expect to see the same amount of performance improvement achieved by doubling the total amount of memory banks, regardless the amount of memory channels present in the system. Also, due to the fact that there exists a row hit improvement limit in the multi memory bank design, it is reasonable to have decaying performance benefit of as we continue doubling the total amount of memory banks.

To conclude, we have designed and tested our DDR3 memory model for the SimpleScalar simulator which is capable of producing consistent simulation results under different simulation configurations. Our DDR3 memory model has the following feature set:

- Support up to 2 memory channels and 16 memory banks;
- Support any reasonable input of processor clock speed, memory bus speed and memory latencies;
- Generate consistent results amount different memory configurations.

Chapter 4

The Cache Filtering Algorithm for Least Frequently Used Data

In this chapter, we will propose the cache filtering algorithm for least frequently used data (cache filter algorithm). This algorithm allows the cache to identify and filter out rarely used and non-previously used memory addresses.

The remainder of this chapter will be organized as follows:

- The background;
- The baseline system's memory hierarchy characteristics;
- The cache filtering algorithm and the components;
- Simulation and discussion.

4.1 The background

Our cache filtering algorithm builds on the basics of two patented works. The first work, *Cache Filtering Method and Apparatus* [21], provided us important insights about how does the memory row hit signal can be used to identified previously used memory rows/pages and addresses. And the second work, *Methods and Apparatus for Accelerating Retrieval of Data from a Memory System with Cache by Reducing Latency* [22], had given us a hint about the possibility of using a small buffer to hold non-previously used and rarely used data. After serious considerations and modifications to the two works, we were able to combine them together and harness their strengths.

Our cache filtering algorithm makes use of a small buffer to store the filtered cache blocks and determine whether a cache block is frequently or previously used by checking:

- the content of the buffer AND
- the memory row/page hit signal generated by the memory controller

4.2 The baseline system's memory hierarchy characteristics

We decided to take the simulated system from section 3.3 as our baseline system to implement the cache filtering algorithm. Therefore we will assume our baseline system to have the follow memory hierarchy characteristics:

- 2 level cache hierarchy (separated instruction L1 cache and data L1 cache);
- No write buffer;
- No memory access reordering.

4.3 The cache filtering algorithm and its components

To ensure the data being fetched to the cache hierarchy are frequently/previously used, we will only allow data to be fetched to the cache hierarchy from the following sources:

- the L2 cache (only applicable to the L1 cache);
- the filter buffer;
- an opened memory row/page.

The remaining access types that are not allowed to enter the cache hierarchy stored in the filter buffer. These accesses include:

- fetches that causes a miss in all L1 cache, L2 cache, filter buffer and memory rows/pages;
- writebacks from data L1 cache to L2 cache that causes a L2 cache miss;
- all L2 writebacks.

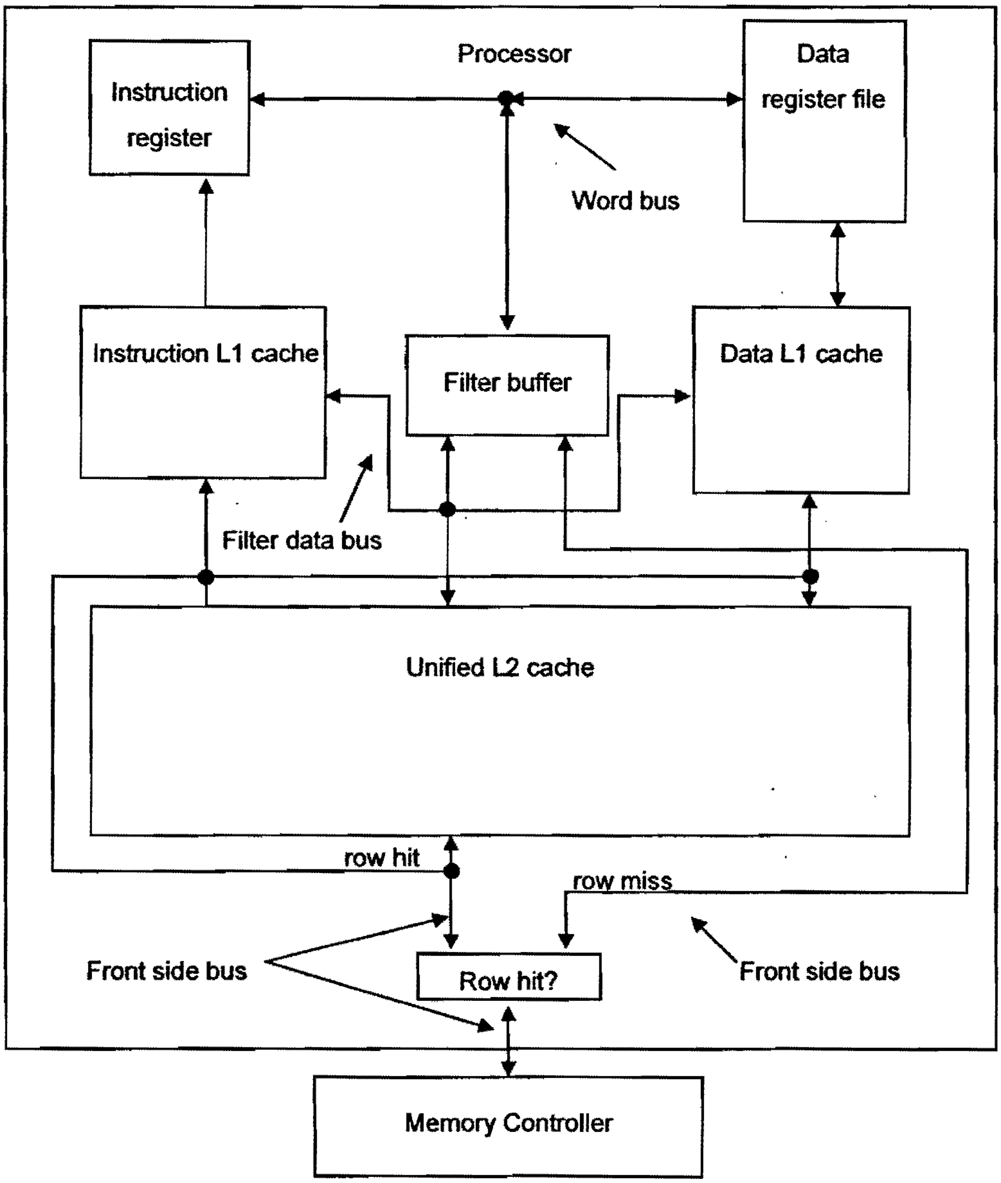


Figure 4.1: Block diagram of the cache filtering scheme

Figure 4.1 shows the block diagram of the cache filtering scheme. This diagram shows the major components of the cache filtering algorithm: instruction register, data register file, instruction L1 cache, data L1 cache, L2 cache and the filter buffer.

4.3.1 The instruction register (IR) and data register file (DRs)

Same as the IR and DRs of conventional memory hierarchies, our cache filtering scheme supplies data and instructions to the processor with the IR and DRs. The only difference is the IR and DRs can fetch/write data from/to the filter buffer when cache filtering is used.

4.3.2 The instruction and data L1 caches (I and D L1 caches)

The design of the I and D L1 caches remains largely unchanged from the design of the conventional memory hierarchy's. The only differences are the source of data fetching and destination of writebacks.

Fetch: The default data fetching source of the I and D L1 caches is the L2 cache. However, when the required data is not found in the L2 cache, the I and D L1 caches will fetch from the filter buffer.

Writeback (Data L1 cache only): The data L1 cache will only perform writebacks to the L2 cache if and only if the L2 cache has the writeback block. If the writeback block is not found in the L2 cache, the L1 cache will write its writeback block to the filter buffer to avoid eviction of useful cache blocks stored by the L2 cache (filtered writeback).

Note: Eviction of useful cache block happens when fetching block is replaced by the writeback block.

4.3.3 The L2 cache

The L2 cache uses the same design as the conventional L2 cache except its fetching and writeback method.

Fetch: The L2 cache first tries to fetch from the filter buffer. If the fetch block is not found in the filter buffer, the L2 cache will fetch from the main memory.

Writeback: In order to avoid unexpected closing of frequently accessed memory rows/pages, the L2 cache will always writeback to the filter buffer (filtered writeback).

4.3.4 The filter buffer

The filter buffer is a new component introduced by the cache filtering scheme. It is a small and high speed SRAM device which is similar to conventional victim caches but implements the write through policy. As an important component of the cache filtering algorithm, the filter buffer's purpose is to assist the cache filtering algorithm through the following ways:

- Provide an alternative path to supply data to the L1 and L2 caches;
- Provide a temporary storage space for filtered data;

Note: The simple write through policy was chosen due to its simplicity and its capability of maintaining most up-to-date values in both the filter buffer and the main memory. However, the write through policy can generate extra memory accesses.

The filter data bus

The filter data bus is a bi-directional connection between the filter buffer and the caches (the instruction L1 cache, data L1 cache and the unified L2 cache). This bus is added to handle cache block transmissions between the caches and the filter buffer when cache fetching and filtered writebacks happen (to be discussed later).

The word bus

The second bus connection between the filter buffer and the cache hierarchy is the word bus. The word bus allows the processor's instruction register and data register file to share the filter buffer. When filtered access (to be discussed later) happens, the word bus is used to transfer the processor requested instruction/data word between the filter buffer and the instruction/data registers.

The front side bus

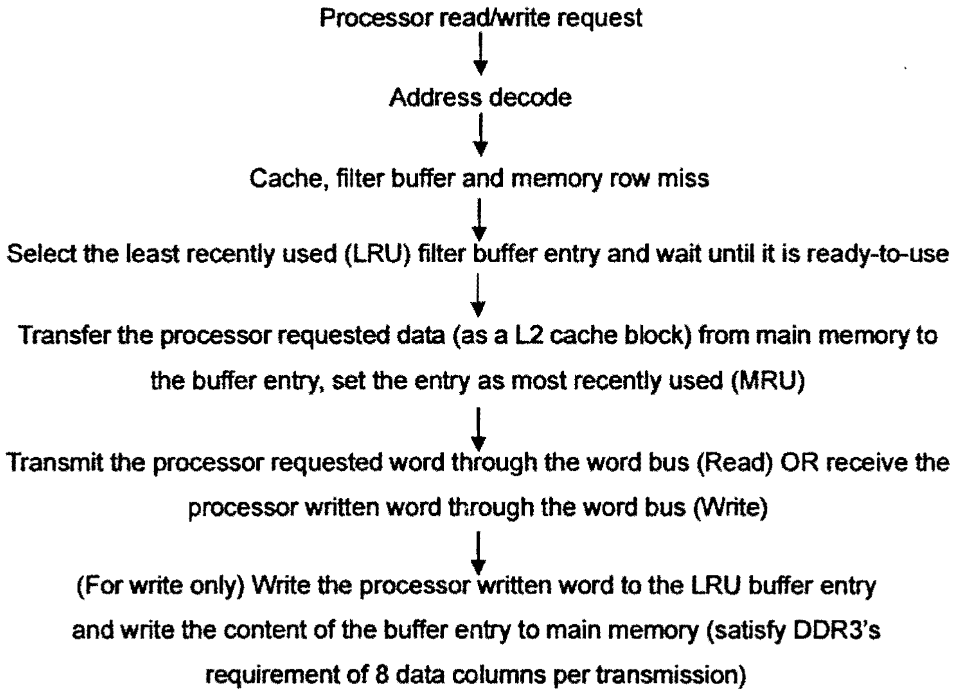
In our custom version of the Simplescalar simulator, the front side bus is being modeled as the data bus which allows the L2 cache, the filter buffer to share the main memory. It is important to note that this single bus is being routed to/shared by multiple components of the cache filtering scheme through the use of multiplexers and deultiplexers.

Filter buffer accesses

According to the cache filtering algorithm, there are several situations where the filter buffer will be accessed. These situations include filtered read/write when row miss, cache fetch from the filter buffer when L1 and L2 cache miss, filtered writebacks.

Filtered access to/from main memory:

When the processor requests for data that is not found in all cache levels, the filter buffer and all opened memory rows/pages, the cache filtering algorithm will perform a filtered read/write where the actions described by Figure 4.2 will happen:



Processor read/write request

Address decode

Cache, filter buffer and memory row miss

Select the least recently used (LRU) filter buffer entry and wait until it is ready-to-use

Transfer the processor requested data (as a L2 cache block) from main memory to the buffer entry, set the entry as most recently used (MRU)

Transmit the processor requested word through the word bus (Read) OR receive the processor written word through the word bus (Write)

(For write only) Write the processor written word to the LRU buffer entry and write the content of the buffer entry to main memory (satisfy DDR3's requirement of 8 data columns per transmission)

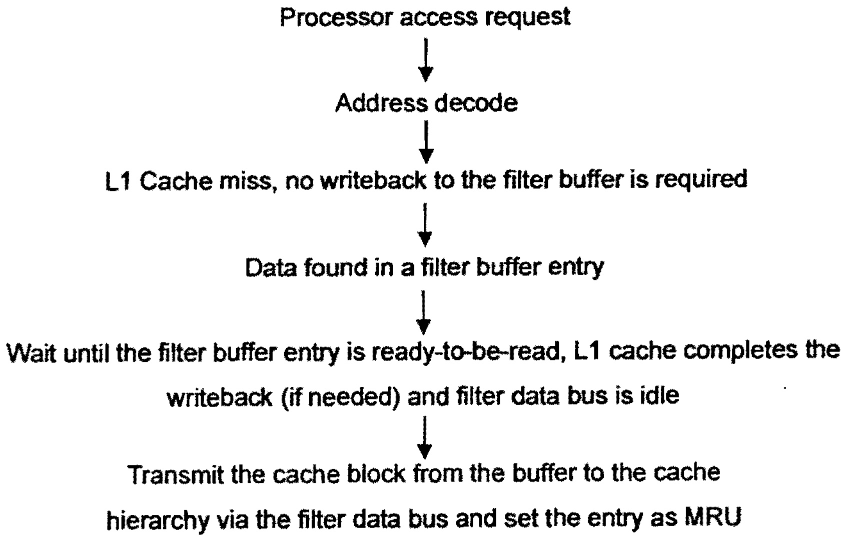
Figure 4.2: Filtered read/write work flow

Note: When handling filtered write, the buffer must first load the L2 cache block from main memory before writing the block back. This is because the processor has only one 32bits word to be written to the main memory, but the DDR3 standard requires at least 8 data columns (sixteen 32bits words) per transmission. Therefore, filtered write requires 2 memory accesses to happen.

Cache fetch from the filter buffer:

When the following conditions are satisfied, a cache fetch from the filter buffer with the steps described by Figure 4.3 will happen:

- processor requested data is not found in the cache hierarchy AND
- processor requested data is found in the filter buffer AND
- the L1 cache does not need to writeback OR L1 cache's writeback block is found in the L2 cache.



Processor access request
Address decode
L1 Cache miss, no writeback to the filter buffer is required
Data found in a filter buffer entry
Wait until the filter buffer entry is ready-to-be-read, L1 cache completes the writeback (if needed) and filter data bus is idle
Transmit the cache block from the buffer to the cache hierarchy via the filter data bus and set the entry as MRU

Figure 4.3: Cache fetching from the filter buffer work flow

Filtered writeback followed by cache fetch:

Filtered writeback followed by cache fetch (Figure 4.4) happens when the following conditions are met:

- The fetched block is found in the filter buffer (for cache fetching) AND
- The L1 writeback block is not found in the L2 cache AND/OR
- The L2 requires a writeback to service the cache fetching.

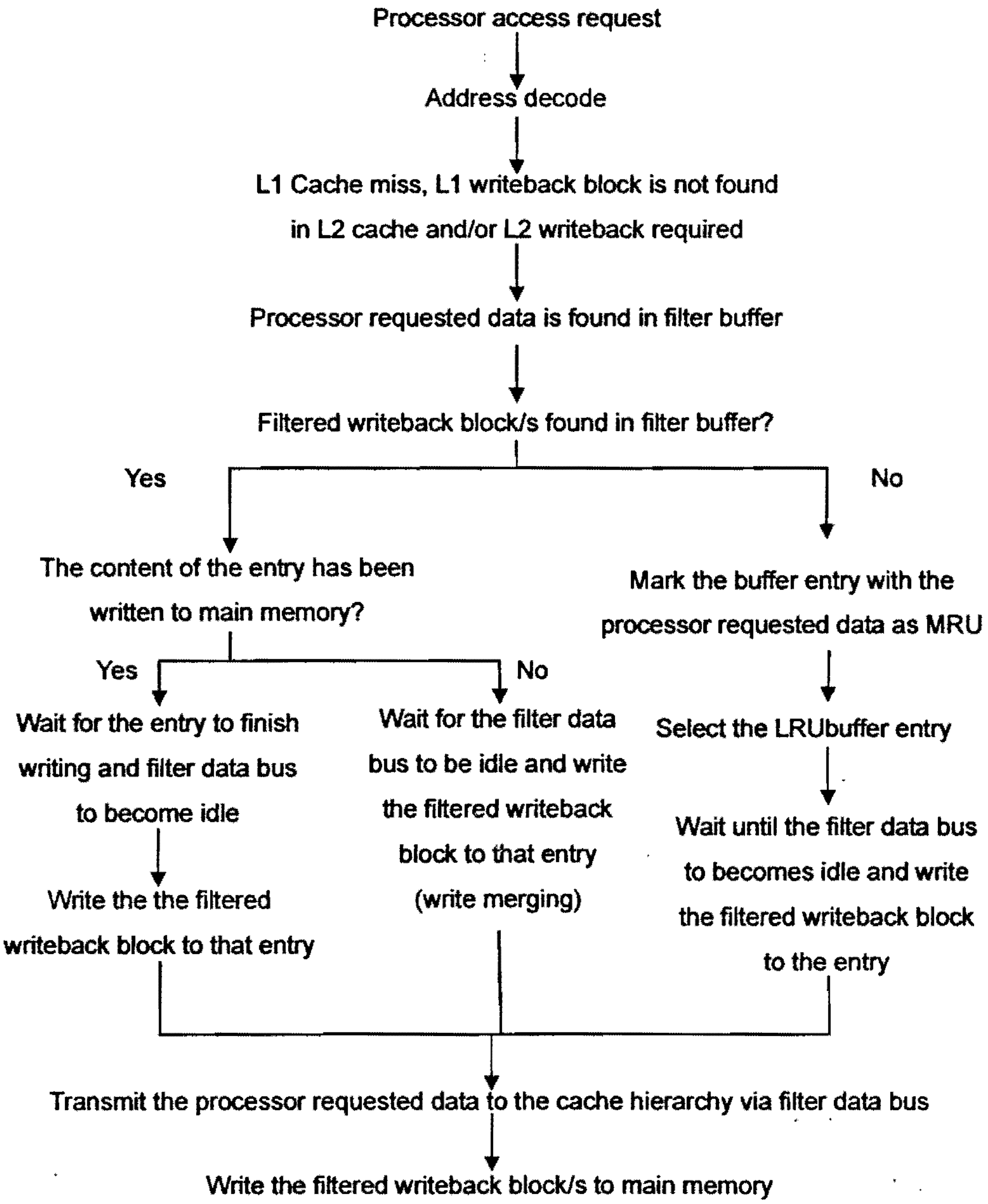

Processor access request
Address decode
L1 Cache miss, L1 writeback block is not found in L2 cache and/or L2 writeback required
Processor requested data is found in filter buffer
Filtered writeback block/s found in filter buffer?
Yes
No
The content of the entry has been written to main memory?
Mark the buffer entry with the processor requested data as MRU
Yes
No
Wait for the entry to finish writing and filter data bus to become idle
Wait for the filter data bus to be idle and write the filtered writeback block to that entry (write merging)
Select the LRUbuffer entry
Write the the filtered writeback block to that entry
Wait until the filter data bus to becomes idle and write the filtered writeback block to the entry
Transmit the processor requested data to the cache hierarchy via filter data bus
Write the filtered writeback block/s to main memory

Figure 4.4: Filtered writeback followed by cache fetching work flow

Filtered writeback without cache fetching:

Filtered writeback without cache fetching (Figure 4.5) happens when the following conditions are met:

- The fetching block is found in the L2 cache OR an opened memory row/page (the fetching block is not found in the L2 cache);
- The L1 writeback block is not found in the L2 cache AND/OR
- The L2 requires a writeback to service the cache fetching.

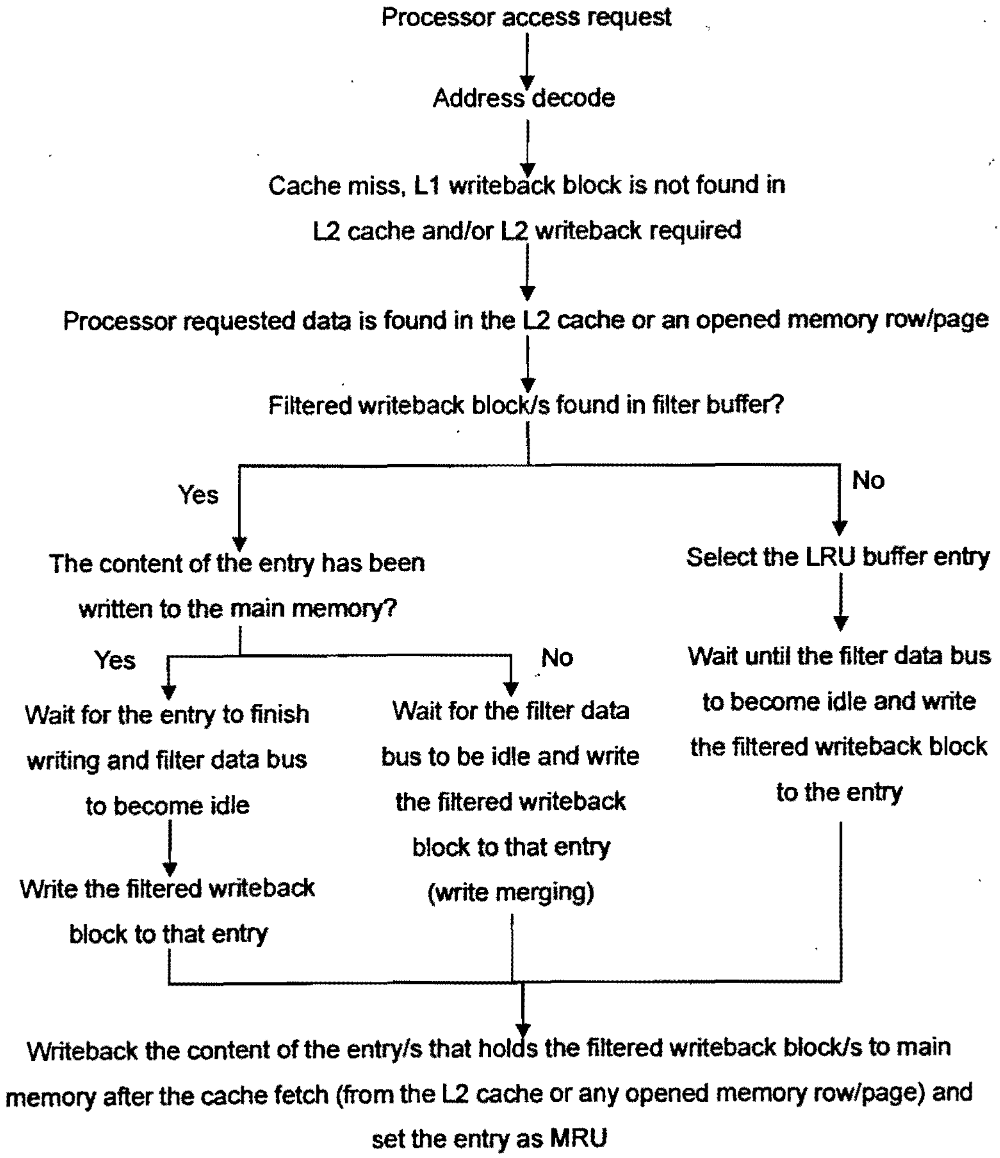

Processor access request
Address decode
Cache miss, L1 writeback block is not found in L2 cache and/or L2 writeback required
Processor requested data is found in the L2 cache or an opened memory row/page
Filtered writeback block/s found in filter buffer?
Yes
No
The content of the entry has been written to the main memory?
Select the LRU buffer entry
Yes
No
Wait until the filter data bus to become idle and write the filtered writeback block to the entry
Wait for the entry to finish writing and filter data bus to become idle
Wait for the filter data bus to be idle and write the filtered writeback block to that entry (write merging)
Write the filtered writeback block to that entry
Writeback the content of the entry/s that holds the filtered writeback block/s to main memory after the cache fetch (from the L2 cache or any opened memory row/page) and set the entry as MRU

Figure 4.5: Filtered writeback without cache fetching work flow

4.3.5 The algorithm

The cache filtering algorithm for least frequently used data was designed such that it is active when cache miss happens and to filter the cache hierarchy based on: the status of the cache hierarchy, the content of the filter buffer and the opened memory row/page. In this subsection, we will first revisit all possible cache miss situations and discuss how the cache filtering algorithm handles each of them.

Conventional cache miss situations

In the conventional two levels cache hierarchy, when L1 cache miss happens, the L1 cache can select either an empty, clean or dirty block to replace. If an empty or clean block is selected, one of the situations described by Figure 4.6 will happen:

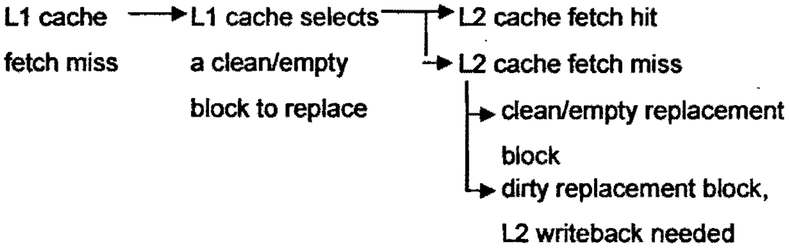

L1 cache fetch miss
L1 cache selects a clean/empty block to replace
L2 cache fetch hit
L2 cache fetch miss
clean/empty replacement block
dirty replacement block, L2 writeback needed

Figure 4.6: Possible outcomes for L1 fetch miss with L1 clean/empty replacement block

If a dirty block is selected by the L1 cache for replacement, one of the situations described by Figure 4.7 will happen instead:

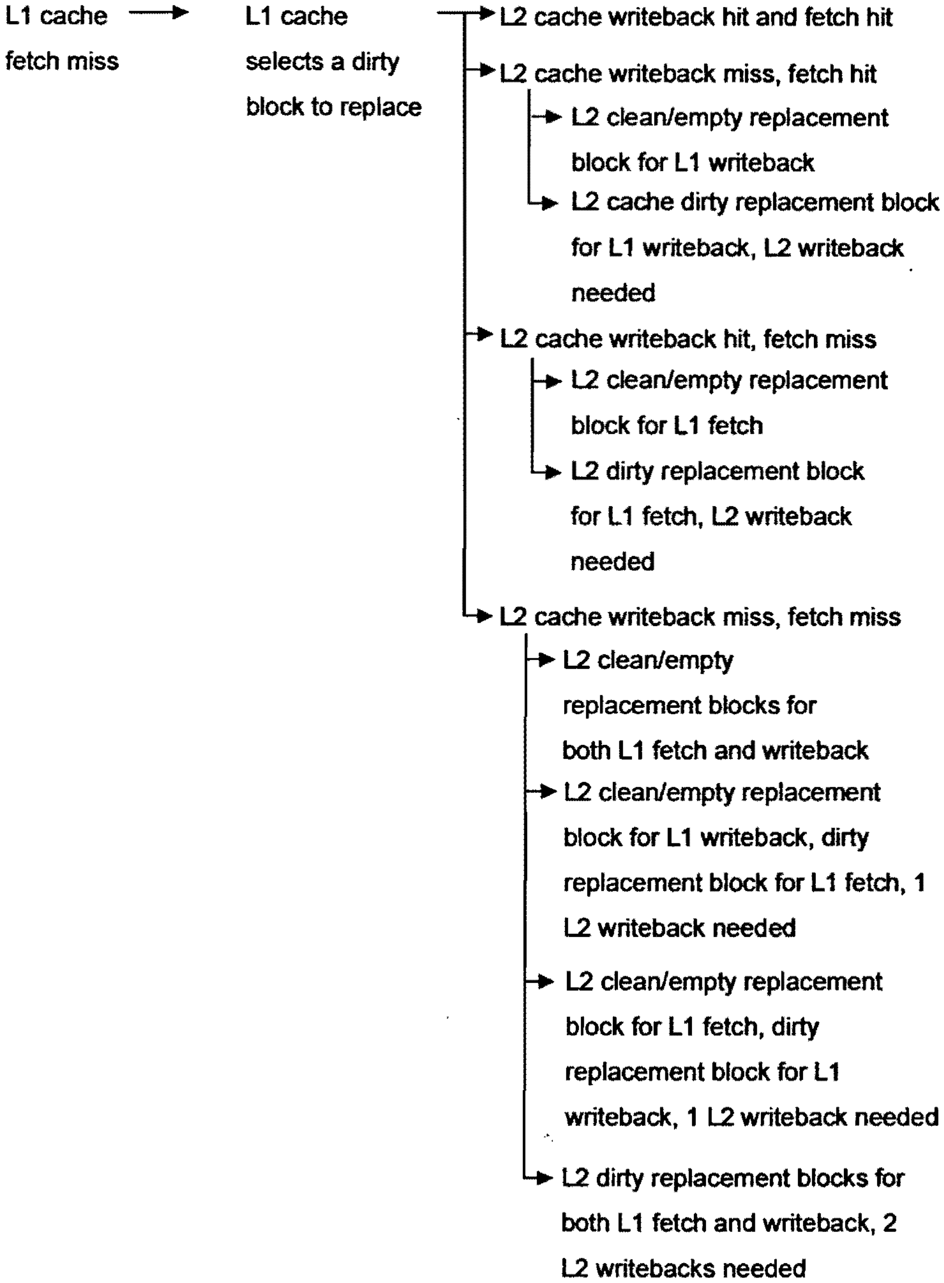

L1 cache fetch miss
L1 cache selects a dirty block to replace
L2 cache writeback hit and fetch hit
L2 cache writeback miss, fetch hit
L2 clean/empty replacement block for L1 writeback
L2 cache dirty replacement block for L1 writeback, L2 writeback needed
L2 cache writeback hit, fetch miss
L2 clean/empty replacement block for L1 fetch
L2 dirty replacement block for L1 fetch, L2 writeback needed
L2 cache writeback miss, fetch miss
L2 clean/empty replacement blocks for both L1 fetch and writeback
L2 clean/empty replacement block for L1 writeback, dirty replacement block for L1 fetch, 1 L2 writeback needed
L2 clean/empty replacement block for L1 fetch, dirty replacement block for L1 writeback, 1 L2 writeback needed
L2 dirty replacement blocks for both L1 fetch and writeback, 2 L2 writebacks needed

Figure 4.7: Possible outcomes for L1 fetch miss with L1 dirty replacement block

Note: Situations involving L1 and L2 writebacks can lead to eviction of useful L2 cache blocks and/or unexpected closing of frequently used memory rows/pages. Therefore, filtered writebacks are added as part of the cache filtering algorithm to avoid those unnecessary cache/row misses.

Cache filtering algorithm's cache miss handling

With all possible cache miss situations properly defined, we can now demonstrate how the cache filtering algorithm handles each of the situations. If the cache filtering algorithm is implemented on a 1 level cache hierarchy or the algorithm is handling L2 cache misses of a 2 level cache hierarchy, the algorithm described by Figure 4.8 (decision tree) and Figure 4.9 (psuedo code) will be used:

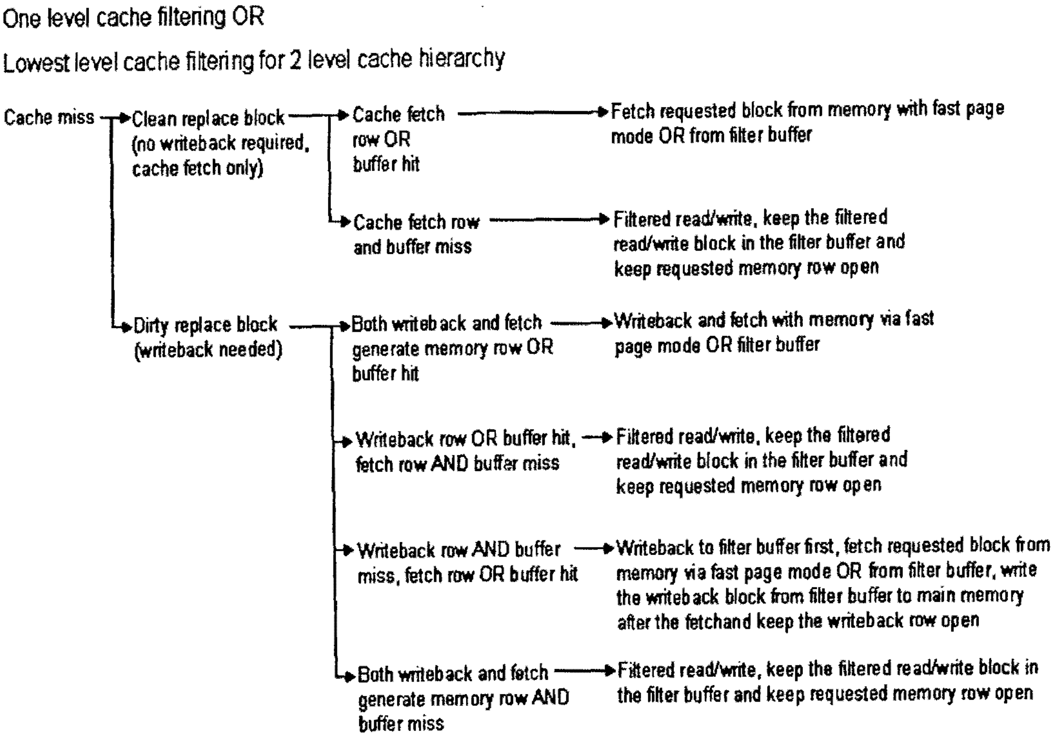
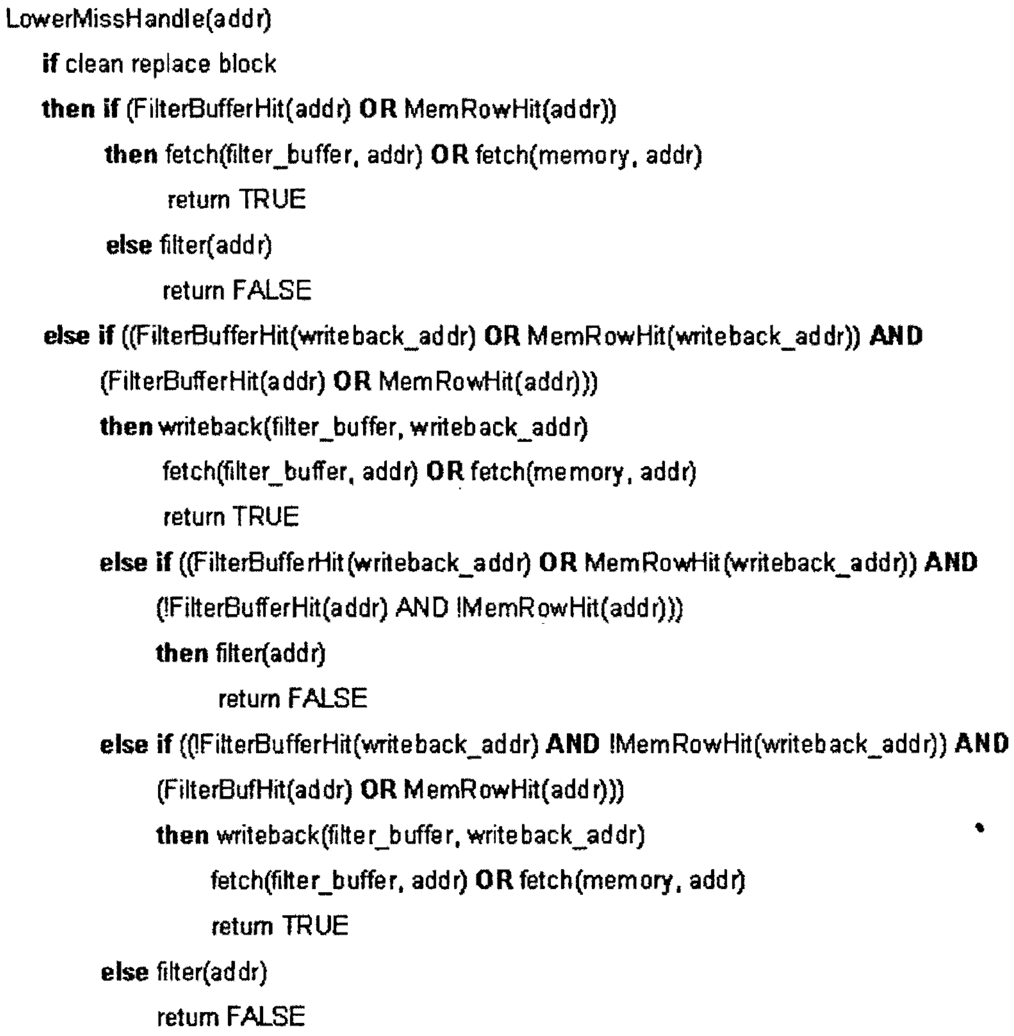

One level cache filtering OR
Lowest level cache filtering for 2 level cache hierarchy
Cache miss
Clean replace block (no writeback required, cache fetch only)
Cache fetch row OR buffer hit
Fetch requested block from memory with fast page mode OR from filter buffer
Cache fetch row and buffer miss
Filtered read/write, keep the filtered read/write block in the filter buffer and keep requested memory row open
Dirty replace block (writeback needed)
Both writeback and fetch generate memory row OR buffer hit
Writeback and fetch with memory via fast page mode OR filter buffer
Writeback row OR buffer hit, fetch row AND buffer miss
Filtered read/write, keep the filtered read/write block in the filter buffer and keep requested memory row open
Writeback row AND buffer miss, fetch row OR buffer hit
Writeback to filter buffer first, fetch requested block from memory via fast page mode OR from filter buffer, write the writeback block from filter buffer to main memory after the fetchand keep the writeback row open
Both writeback and fetch generate memory row AND buffer miss
Filtered read/write, keep the filtered read/write block in the filter buffer and keep requested memory row open

Figure 4.8: One level cache filtering and lowest level cache filtering decision tree for two level cache hierarchies



```
LowerMissHandle(addr)
    if clean replace block
    then if (FilterBufferHit(addr) OR MemRowHit(addr))
            then fetch(filter_buffer, addr) OR fetch(memory, addr)
                    return TRUE
            else filter(addr)
                    return FALSE
    else if ((FilterBufferHit(writeback_addr) OR MemRowHit(writeback_addr)) AND
            (FilterBufferHit(addr) OR MemRowHit(addr)))
            then writeback(filter_buffer, writeback_addr)
                    fetch(filter_buffer, addr) OR fetch(memory, addr)
                    return TRUE
            else if ((FilterBufferHit(writeback_addr) OR MemRowHit(writeback_addr)) AND
                    (!FilterBufferHit(addr) AND !MemRowHit(addr)))
                    then filter(addr)
                            return FALSE
            else if ((!FilterBufferHit(writeback_addr) AND !MemRowHit(writeback_addr)) AND
                    (FilterBufHit(addr) OR MemRowHit(addr)))
                    then writeback(filter_buffer, writeback_addr)
                            fetch(filter_buffer, addr) OR fetch(memory, addr)
                            return TRUE
            else filter(addr)
                    return FALSE
```

Figure 4.9: One level cache filtering and lowest level cache filtering psuedo code for two level cache hierarchies

If the cache filtering algorithm is implemented on a two level cache hierarchy and the algorithm is handling L1 cache misses, the following decision tree (Figure 4.10) and psuedo code (Figure 4.11) will be used:

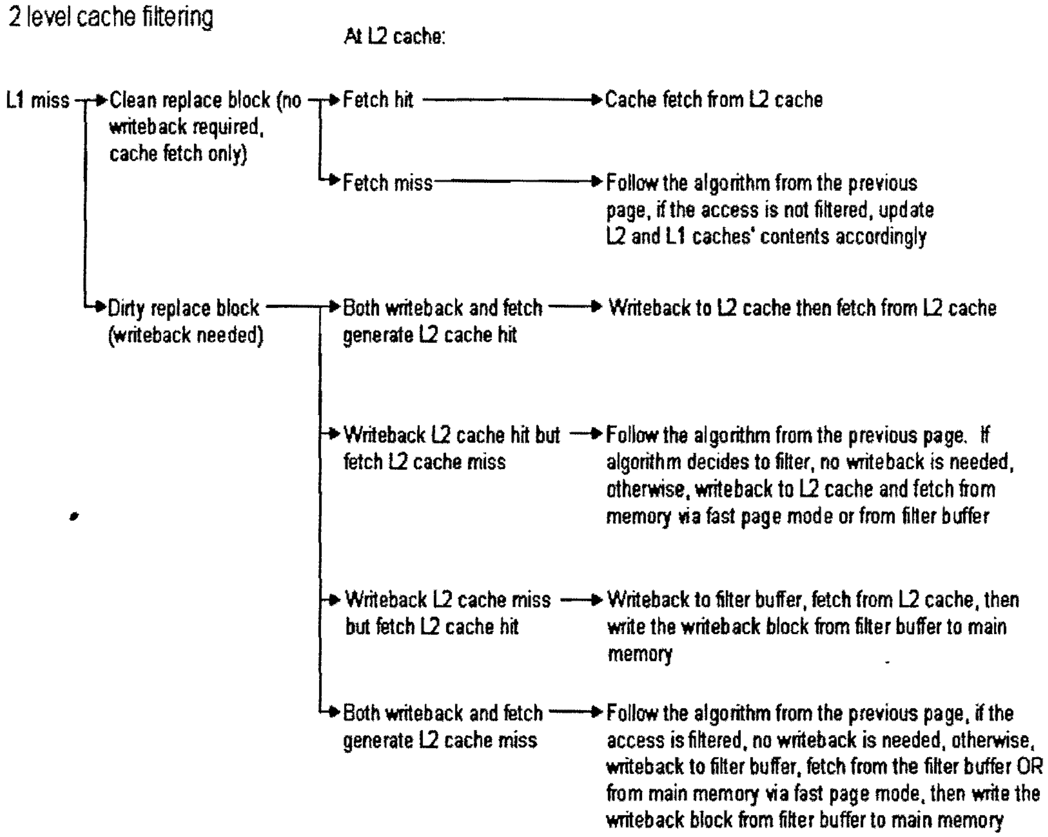

2 level cache filtering
At L2 cache:
L1 miss
Clean replace block (no writeback required, cache fetch only)
Fetch hit
Cache fetch from L2 cache
Fetch miss
Follow the algorithm from the previous page, if the access is not filtered, update L2 and L1 caches' contents accordingly
Dirty replace block (writeback needed)
Both writeback and fetch generate L2 cache hit
Writeback to L2 cache then fetch from L2 cache
Writeback L2 cache hit but fetch L2 cache miss
Follow the algorithm from the previous page. If algorithm decides to filter, no writeback is needed, otherwise, writeback to L2 cache and fetch from memory via fast page mode or from filter buffer
Writeback L2 cache miss but fetch L2 cache hit
Writeback to filter buffer, fetch from L2 cache, then write the writeback block from filter buffer to main memory
Both writeback and fetch generate L2 cache miss
Follow the algorithm from the previous page, if the access is filtered, no writeback is needed, otherwise, writeback to filter buffer, fetch from the filter buffer OR from main memory via fast page mode, then write the writeback block from filter buffer to main memory

Figure 4.10: L1 cache filtering decision tree for two level cache hierarchies

```
UpperMissHandle(addr)
if clean replace block
then if L2_hit(addr)
    then fetch(L2, addr)
         return TRUE
    else if LowerMissHandle(addr)
        then fetch(L2, addr)
        else  filter(addr)
              return FALSE
else if (L2_hit(addr) AND L2_hit(writeback_addr))
    then writeback(L2, writeback_addr)
        fetch(L2, addr)
    else if (L2_hit(writeback_addr) AND !L2_hit(addr))
        then if LowerMissHandle(addr)
            then writeback(L2, writeback_addr)
                 fetch(L2, addr)
    else if (!L2_hit(writeback_addr) AND L2_hit(addr))
        then writeback(filter_buffer, writeback_addr)
            fetch(L2, addr)
    else if LowerMissHandle(addr)
        then writeback(filter_buffer, writeback_addr)
            fetch(L2, addr)
        else filter(addr)
```

Figure 4.11: L1 cache filtering psuedo code decision tree for two level cache hierarchies

This concludes the description of our cache filtering algorithm for least frequently used data. In the next chapter, we will provide the simulation and result analyze for our cache filtering scheme.

Chapter 5

Simulation and analysis of the cache filtering algorithm

This chapter is divided into the following parts in order to analyze the performance of our cache filtering algorithm:

- The desktop/notebook computer environment and simulation settings;
- Simulation results and discussion (desktop/notebook environment);

5.1 The desktop/notebook computer environment and simulation settings

To explore the potential of our cache filtering algorithm, we will use our custom version of sim-outorder simulator, with our accurate DDR3 memory model and cache filtering algorithm, to simulate and compare a baseline desktop/notebook computer system (with cache filtering algorithm disabled) with a cache filtered desktop/notebook computer. We have also simulated the baseline and cache filtered system with four different L2 cache sizes (256KB, 512KB, 1MB and 2MB) to analyze our cache filtering algorithm's sensitivity to different L2 cache size.

In our simulations, we will use the ammp, equake, gcc, gzip, mcf, parser and vortex benchmarks from the SPEC CPU 2000 benchmark suite [19,20].

Note: Other than the processor speed, cache and memory settings, all simulator settings will be left as default.

The following is a table of simulation settings:

Table 5.1: Simulation settings for the desktop/notebook environment

Processor speed	3.2Ghz
L1 Instruction cache	32KB, 4 way set associative, 64B block size, access latency = 1 processor clock (clk)
L1 data cache	32KB, 4 way set associative, 64B block size, access latency = 1 processor clk
L2 cache	Size varies (256KB, 512KB, 1MB and 2MB), 8 way set associative, 64B block size, access latency = 10 processor clks
Memory bus	64bit (8B) wide, 800Mhz speed (DDR3 1600)
DRAM	8 banks, 4KB row buffer, page interleaving
DRAM timing (in memory clocks)	Tcas = 7, Trcd = 7, Trp = 7, Tras = 21, Tburst = 8, Twr = 5, Trtrs = 1, Tcmd = 1, Tcwd = 7, Tccd = 4
Filter buffer size (filter buffer only present in cache filtered system)	64 entries x 64B (entry size, same as the L2 cache block size) = 4KB
Filter buffer search latency	1 cpu clk
Filter buffer transfer latency	2 cpu clks
Filter data bus latency	1 cpu clk (Ceiling(0.24ns / 3.2Ghz))
Word bus latency	1 cpu clk (Ceiling(0.24ns / 3.2Ghz))

5.2 Simulation results and discussion (desktop/notebook environment)

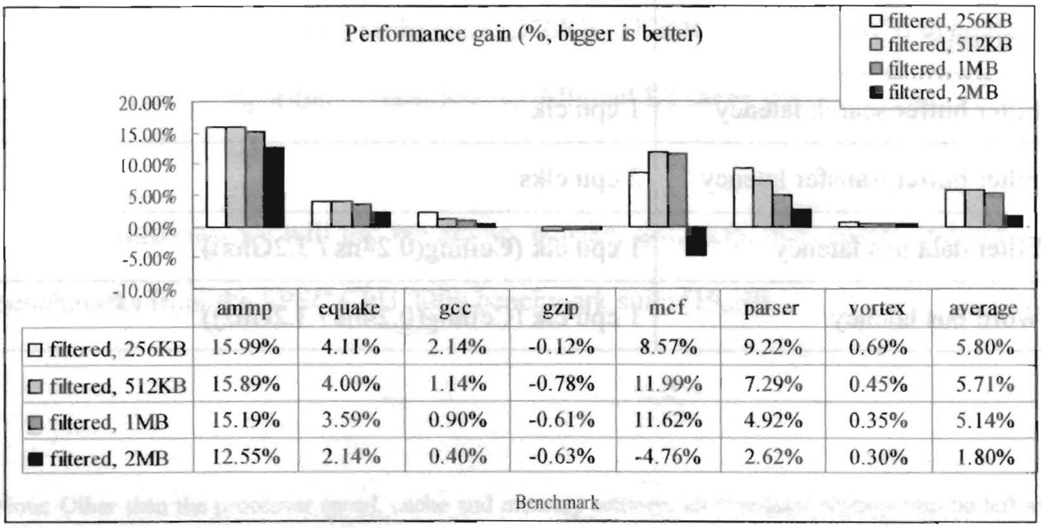
We will break this sub-section into the followings:

- Cache filtering performance and performance gain;
- Filter count;
- Extra memory accesses;
- Memory row hit rate;
- Ratio of filtered write versus total memory accesses;
- Cache hit rate (L1 and L2).

Cache filtering performance and performance gain

Figure 5.1 shows the cache filtered system's performance gain (in percentage) comparing to the baseline system. Figure 5.2 – 5.5 shows the performance (in instruction per cycle) of the basline and cache filtered systems with different L2 cache sizes.

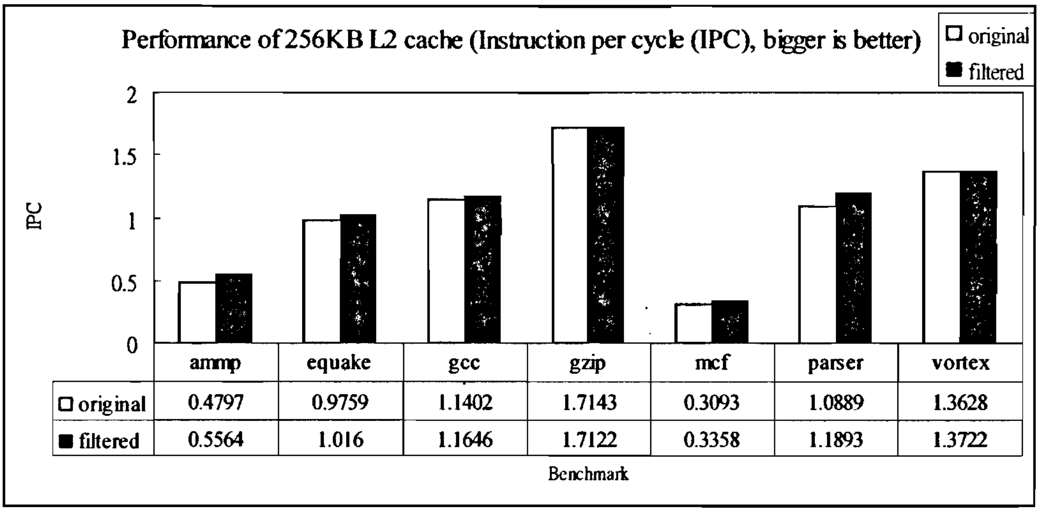
Performance gain of the cache filtered systems



	ammp	equake	gcc	gzip	mcf	parser	vortex	average
□ filtered, 256KB	15.99%	4.11%	2.14%	-0.12%	8.57%	9.22%	0.69%	5.80%
▨ filtered, 512KB	15.89%	4.00%	1.14%	-0.78%	11.99%	7.29%	0.45%	5.71%
▩ filtered, 1MB	15.19%	3.59%	0.90%	-0.61%	11.62%	4.92%	0.35%	5.14%
■ filtered, 2MB	12.55%	2.14%	0.40%	-0.63%	-4.76%	2.62%	0.30%	1.80%

Figure 5.1: Performance gain of the cache filtered systems

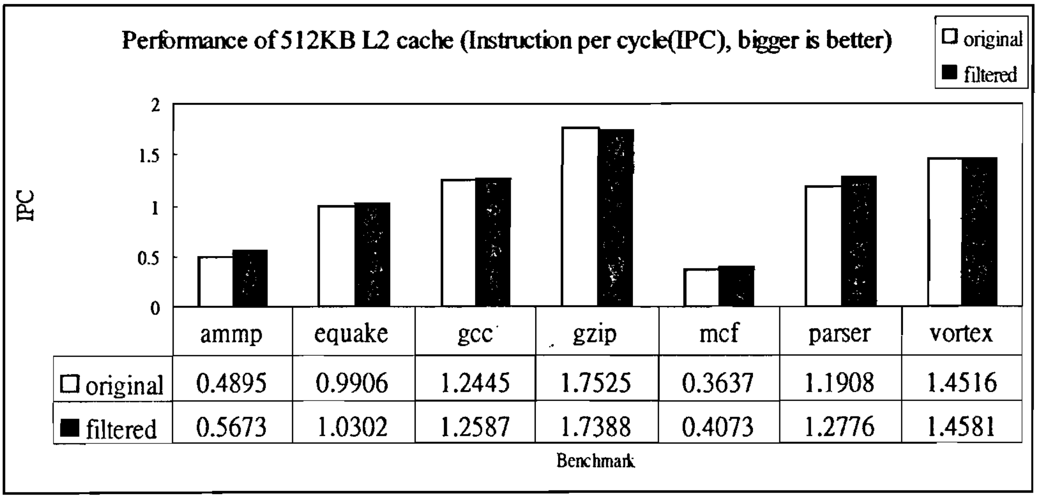
Performance of the systems with 256KB L2 cache



	ammp	equake	gcc	gzip	mcf	parser	vortex
□ original	0.4797	0.9759	1.1402	1.7143	0.3093	1.0889	1.3628
■ filtered	0.5564	1.016	1.1646	1.7122	0.3358	1.1893	1.3722

Figure 5.2: Performance of the systems with 256KB L2 cache

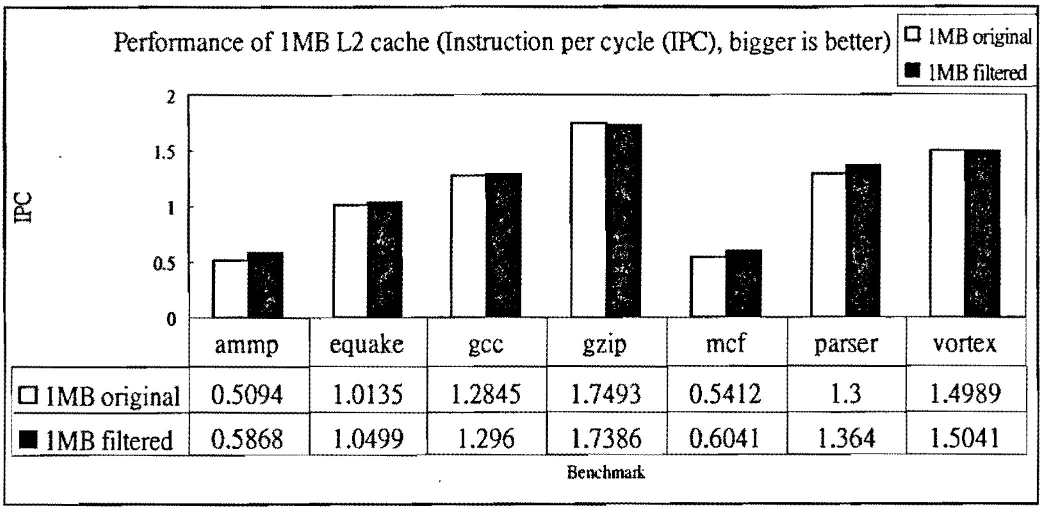
Performance of the systems with 512KB L2 cache



	ammp	equake	gcc	gzip	mcf	parser	vortex
□ original	0.4895	0.9906	1.2445	1.7525	0.3637	1.1908	1.4516
■ filtered	0.5673	1.0302	1.2587	1.7388	0.4073	1.2776	1.4581

Figure 5.3: Performance of the systems with 512KB L2 cache

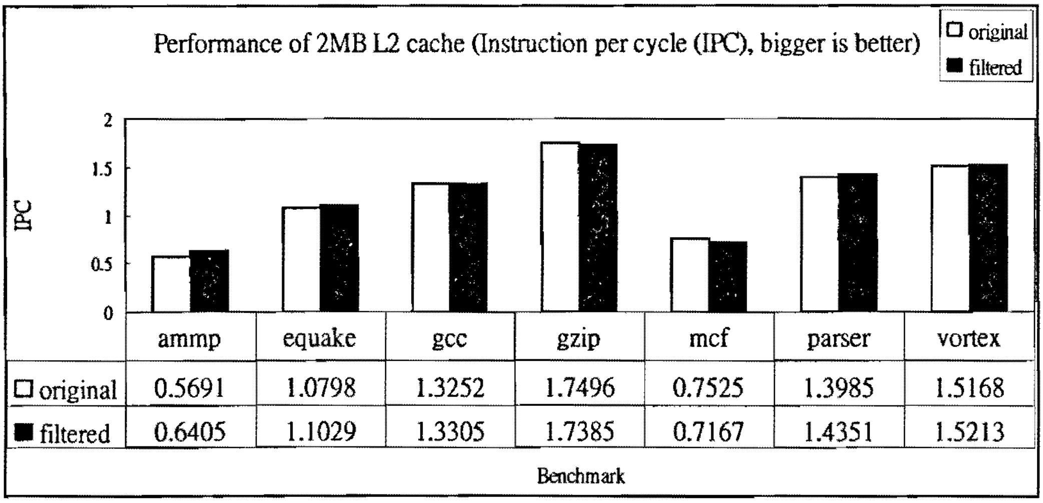
Performance of the systems with 1MB L2 cache



	ammp	equake	gcc	gzip	mcf	parser	vortex
□ 1MB original	0.5094	1.0135	1.2845	1.7493	0.5412	1.3	1.4989
■ 1MB filtered	0.5868	1.0499	1.296	1.7386	0.6041	1.364	1.5041

Figure 5.4: Performance of the systems with 1MB L2 cache

Performance of the systems with 2MB L2 cache



	ammp	equake	gcc	gzip	mcf	parser	vortex
□ original	0.5691	1.0798	1.3252	1.7496	0.7525	1.3985	1.5168
■ filtered	0.6405	1.1029	1.3305	1.7385	0.7167	1.4351	1.5213

Figure 5.5: Performance of the systems with 2MB L2 cache

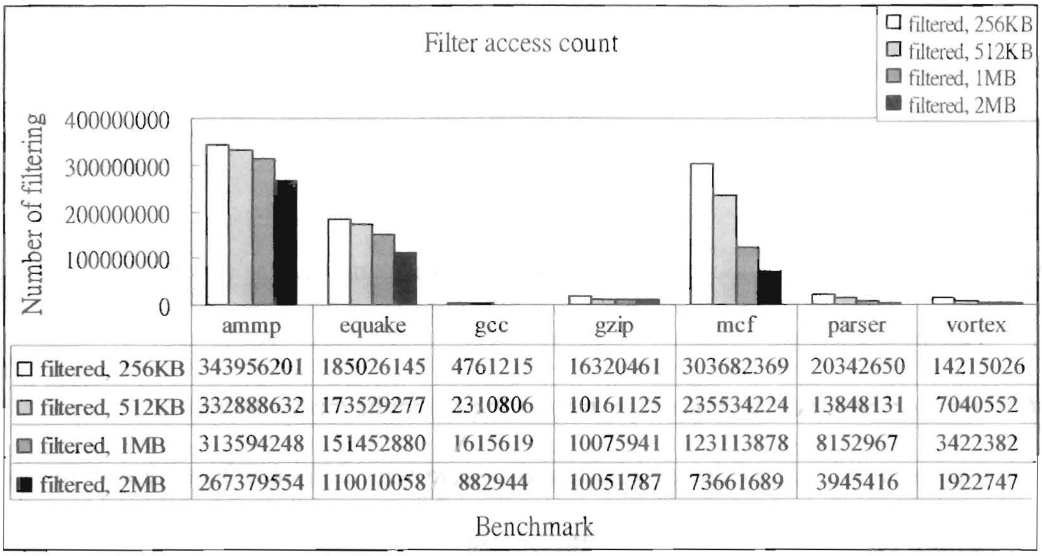
From the performance and performance gain figures, the following observations were made:

- Our cache filtering algorithm is able to produce an average of 5.80%, 5.71%, 5.14% and 1.80% of performance advantage over the baseline systems equiped with 256KB, 512KB, 1MB and 2MB L2 caches respectively.
- The algorithm provides big performance gain for ammp, mcf (excluding 2MB L2 cache) and parser.
- The performance gain tends to drop as the L2 cache size increases.
- The performance of the cache filtered system with 256KB L2 cache running the ammp benchmark is better than the baseline system with 512KB and 1MB of L2 cache.
- The performance of the cache filtered system with 256KB L2 cache running the equake and parser benchmark is better than the baseline system with 512KB L2 cache.

In the meantime, there is not enough information to explain the negative performance gains of gzip and mcf with 2MB L2 cache and the big performance gain for ammp, mcf and parser. We will try to explain the performance drop in the upcoming plots.

Filter access count

After the performance gain of the cache filtering algorithm, we will start analyzing the performance by studying the behavior of the cache filter through the total cache filtering count. The following plot shows the total amount of cache filtering happened throughout the runtime of the benchmarks.



	ammp	equake	gcc	gzip	mcf	parser	vortex
□ filtered, 256KB	343956201	185026145	4761215	16320461	303682369	20342650	14215026
▨ filtered, 512KB	332888632	173529277	2310806	10161125	235534224	13848131	7040552
▨ filtered, 1MB	313594248	151452880	1615619	10075941	123113878	8152967	3422382
■ filtered, 2MB	267379554	110010058	882944	10051787	73661689	3945416	1922747

Figure 5.6: Total amount of cache filtering throughout benchmark execution

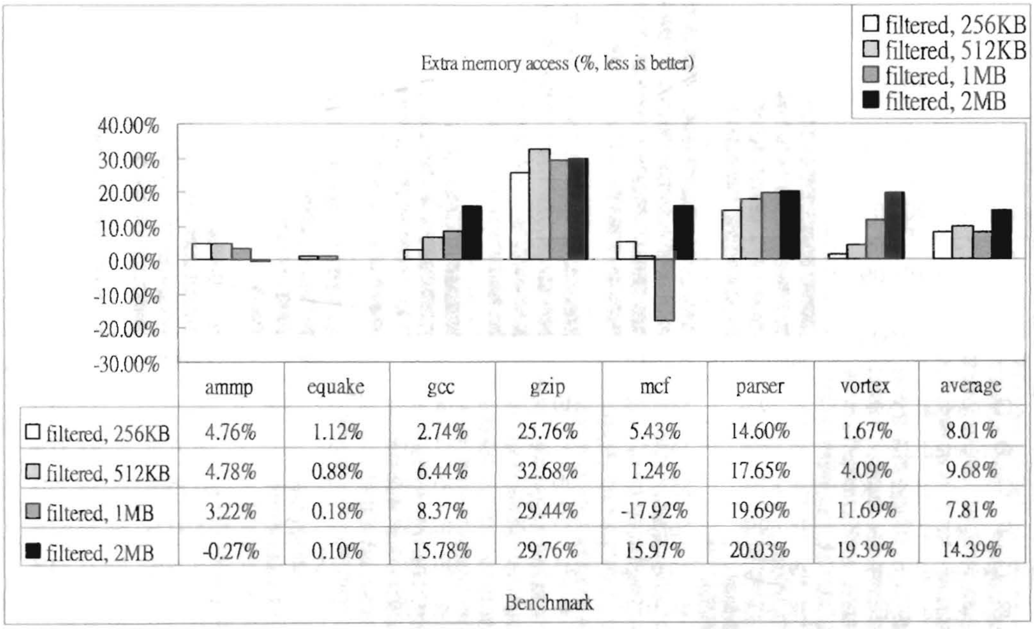
From the filter count plot, the following observation is made:

- The amount of cache filtering decreases as the L2 cache size increases.

To explain this observation, one must consider the fact that our cache filtering algorithm was designed as cache miss handling procedures - the filter only activates whenever cache miss happens. As the L2 cache size increases, the overall hit rate of the cache hierarchy will also increase, effectively reducing the total amount of cache filtering and rendering the cache filter less effective in systems equiped with bigger L2 cache configurations. This outcome can also be used to explain the third observation made from the performance gain plot where the performance gain tends to drop as the L2 cache size increases.

Extra memory accesses

The following plot shows the extra memory accesses, comparing to the baseline system, generated by the cache filtered system (particularly the write through filter buffer and the filtered write accesses). Note that percentage is used to make it easier to compare results from different simulations (simulations with different L2 cache sizes).



	ammp	equake	gcc	gzip	mcf	parser	vortex	average
□ filtered, 256KB	4.76%	1.12%	2.74%	25.76%	5.43%	14.60%	1.67%	8.01%
▨ filtered, 512KB	4.78%	0.88%	6.44%	32.68%	1.24%	17.65%	4.09%	9.68%
▩ filtered, 1MB	3.22%	0.18%	8.37%	29.44%	-17.92%	19.69%	11.69%	7.81%
■ filtered, 2MB	-0.27%	0.10%	15.78%	29.76%	15.97%	20.03%	19.39%	14.39%

Figure 5.7: Extra memory accesses of the cache filtered systems

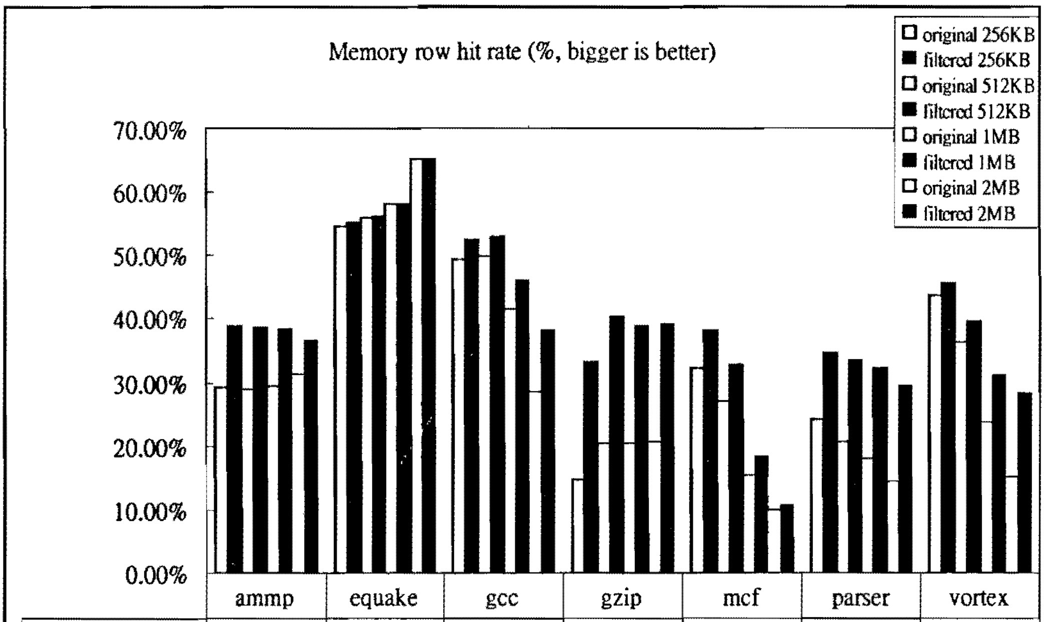
From the extra memory access plot, the following observations were made:

- ammp, equake and mcf (excluding 2MB L2 cache) are having a very small extra access (for mcf with 1MB L2 cache, a negative extra memory access is recorded);
- gzip has a big amount of extra memory accesses and the mcf with 2MB L2 cache simulation has a relatively bigger amount of extra memory access than other mcf simulations.

Considering the characteristics of the write through policy [1], one can conclude that the extra memory accesses were introduced by the filter buffer as memory writes.

Memory row hit rate

Figure 5.8 shows the baseline's and the cache filtered system's memory row hit rate.



	ammp	equake	gcc	gzip	mcf	parser	vortex
□ original 256KB	29.29%	54.58%	49.32%	14.80%	32.18%	24.29%	43.74%
■ filtered 256KB	38.99%	55.18%	52.46%	33.20%	38.24%	34.67%	45.57%
□ original 512KB	28.87%	55.90%	49.93%	20.33%	27.04%	20.75%	36.35%
■ filtered 512KB	38.69%	56.26%	52.99%	40.25%	32.82%	33.41%	39.62%
□ original 1MB	29.53%	58.07%	41.56%	20.48%	15.40%	18.08%	23.74%
■ filtered 1MB	38.34%	58.10%	46.08%	38.87%	18.22%	32.23%	30.98%
□ original 2MB	31.37%	65.25%	28.57%	20.60%	9.97%	14.56%	15.28%
■ filtered 2MB	36.59%	65.28%	38.28%	39.09%	10.74%	29.42%	28.24%

Benchmark

Figure 5.8: The memory row hit rate of the cache filtered systems

From the memory row hit plot, we have several important findings:

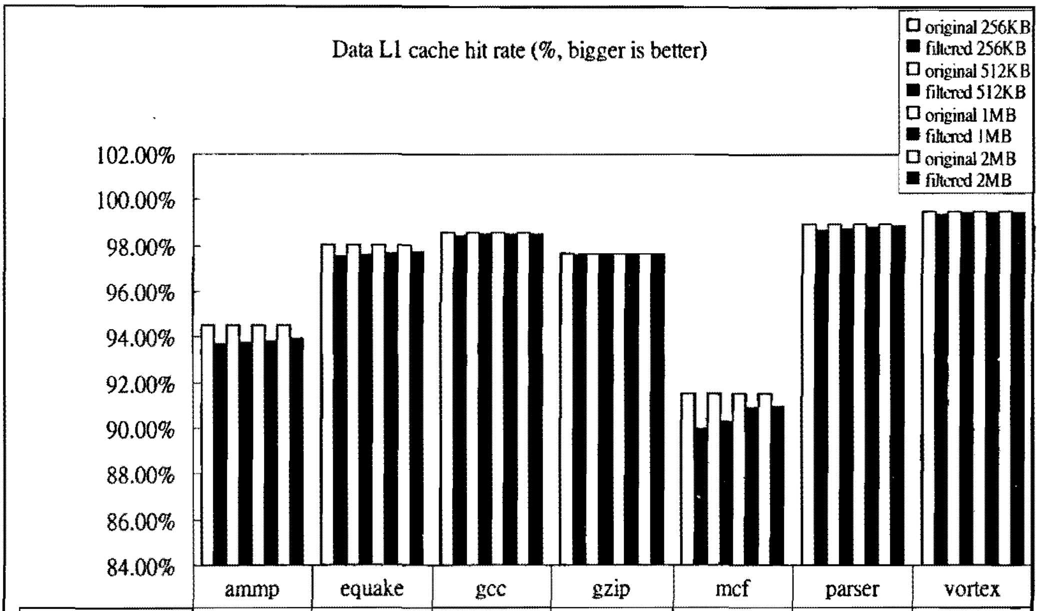
- No drop in memory row hit rate was found.
- The cache filtered system's row hit rate for ammp, gzip and parser are much higher than the baseline system.
- gcc, mcf excluding 2MB L2 cache and vortex are having noticeable-but-not-too-big gain in memory row hit rate.

- mcf with 2MB L2 cache has a much smaller row hit rate increase than other mcf simulations (0.77% vs. >2%)

These findings suggest that our cache filtering algorithm is able to efficiently utilize the feature of the page interleaving scheme to yield better overall system performance.

Cache hit rate (data L1 and L2)

Data L1 cache hit rate



	ammp	equake	gcc	gzip	mcf	parser	vortex
□ original 256KB	94.52%	98.03%	98.58%	97.69%	91.57%	98.96%	99.50%
■ filtered 256KB	93.72%	97.56%	98.47%	97.63%	90.02%	98.72%	99.43%
□ original 512KB	94.52%	98.03%	98.58%	97.69%	91.57%	98.96%	99.50%
■ filtered 512KB	93.77%	97.59%	98.51%	97.65%	90.34%	98.79%	99.46%
□ original 1MB	94.52%	98.03%	98.58%	97.69%	91.57%	98.96%	99.50%
■ filtered 1MB	93.84%	97.65%	98.53%	97.65%	90.95%	98.86%	99.47%
□ original 2MB	94.52%	98.03%	98.58%	97.69%	91.56%	98.96%	99.50%
■ filtered 2MB	93.98%	97.76%	98.55%	97.66%	91.00%	98.91%	99.48%

Benchmark

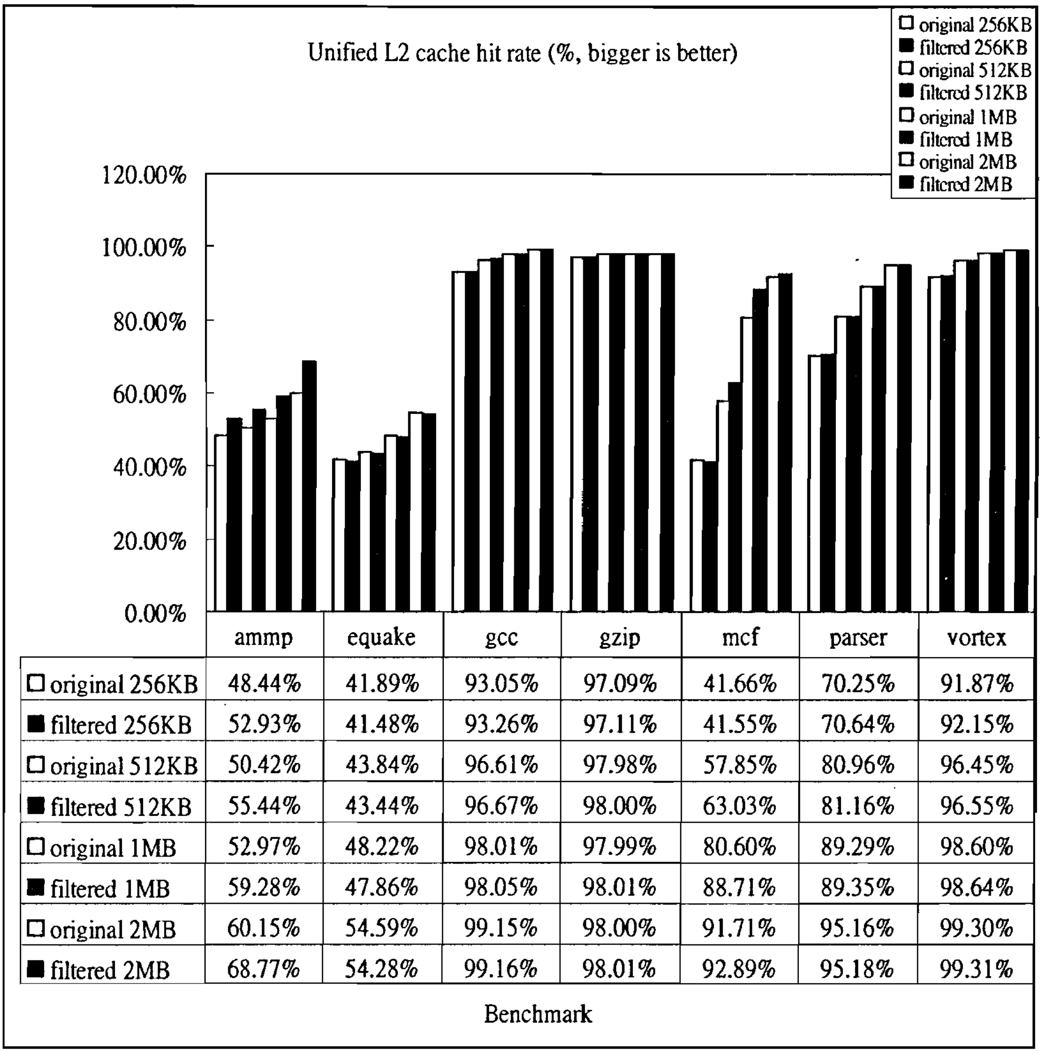
Figure 5.9: Data L1 cache hit rate

From Figure 5.9, we noticed small drop in data L1 cache rate from the benchmarks. This is because the data L1 cache is very frequently accessed. In the baseline system, when L1 cache miss happens, the L1 cache will fetch the processor requested data.

The cache filtered system, on the other hand, is likely that cache miss will occur more than or equal to two times before the corresponding cache block is moved to the cache. This is because the cache filtering algorithm will filter out all first-time cache misses and place the cache block into the filter buffer. Therefore the average amount of cache miss/es before the data is found in the cache is greater than or equal to 2. However, this drop in L1 hit rate is only negligible since the cache filtering algorithm is able to compensate such problem by eliminating cache pollution caused by rarely used data blocks.

Note: Due to the fact that there are no difference between the baseline's and cache filtered system's instruction L1 cache hit rate, we will omit the instruction L1 cache hit rate comparison.

L2 cache hit rate



	ammp	equake	gcc	gzip	mcf	parser	vortex
□ original 256KB	48.44%	41.89%	93.05%	97.09%	41.66%	70.25%	91.87%
■ filtered 256KB	52.93%	41.48%	93.26%	97.11%	41.55%	70.64%	92.15%
□ original 512KB	50.42%	43.84%	96.61%	97.98%	57.85%	80.96%	96.45%
■ filtered 512KB	55.44%	43.44%	96.67%	98.00%	63.03%	81.16%	96.55%
□ original 1MB	52.97%	48.22%	98.01%	97.99%	80.60%	89.29%	98.60%
■ filtered 1MB	59.28%	47.86%	98.05%	98.01%	88.71%	89.35%	98.64%
□ original 2MB	60.15%	54.59%	99.15%	98.00%	91.71%	95.16%	99.30%
■ filtered 2MB	68.77%	54.28%	99.16%	98.01%	92.89%	95.18%	99.31%

Figure 5.10: L2 cache hit rate

From the L2 cache hit rate plot, we have the following findings:

- ammp's L2 cache hit rate improvements have compensated the data L1 cache's degraded hit rate and helped maintaining a small extra memory access rate. And becuase accessing the L2 cache is more than 10 times faster than accessing the main memory, the improved L2 cache hit rate is also very benifitial to the overall performance.

- mcf with 512KB and 1MB L2 cache have bigger L2 cache hit rate (comparing to mcf with 256KB and 2MB L2) improvements, which yielded a very small and a negative extra memory access rate respectively.
- mcf with 256KB and 2MB L2 cache are having slightly degraded and very small L2 cache hit rate improvement respectively. These are the main causes of the relatively bigger (comparing to mcf with 512KB and 1MB) extra access rate.

The special case: gzip and mcf

According to the simulation results, we noticed two seemingly abnormal outcomes where the system with 1 and 2MB of L2 cache are slower than the same system with 512KB L2 cache and the cache filtered system with 2MB of L2 cache being slower than the baseline system and both the baseline. In this subsection, we will try to investigate and explain the cause of such outcome.

gzip:

To explain the gzip's situation, we must first point out that it is not unusual that a bigger cache is causing a slowdown, if and only if the simulated benchmark has reached the point of diminishing return at a certain cache size where other performance limiting factors such as memory bus activities start slowing down the system. To prove our point, we have included figure 5.11, extracted from [23]:

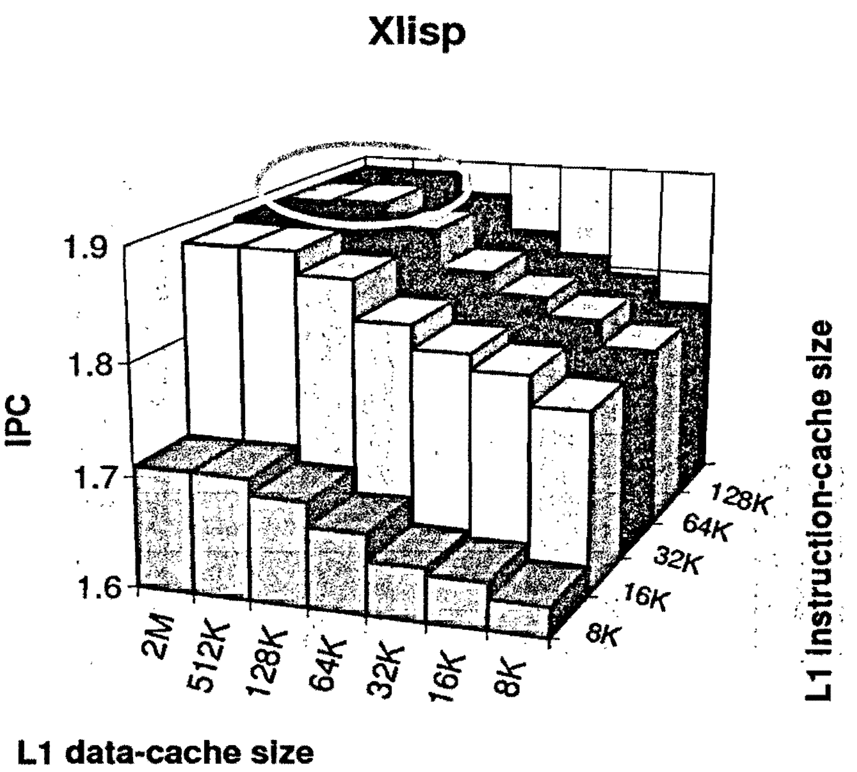


Figure 5.11: Xlisp simulation with varied I and D L1 cache size, extracted from [23]

Figure 5.11 is a simulation plot extracted from [23], which tries to vary both the size of instruction and data L1 cache size, in order to find the relationship between performance and the L1 cache sizes. According to figure 5.11, when the instruction L1 cache is set to be 64KB in size, the system reaches its peak performance when the data L1 cache reaches 512KB of size. If one pay attention to the circled area in the plot, he/she will noticed that the performance of the system with 512KB of L1 data cache is higher than the same system with 2MB of L1 data cache.

We have also completed a set of gzip simulations with the original SimpleScalar simulator to look at the relationship between gzip's performance and L2 cache size:

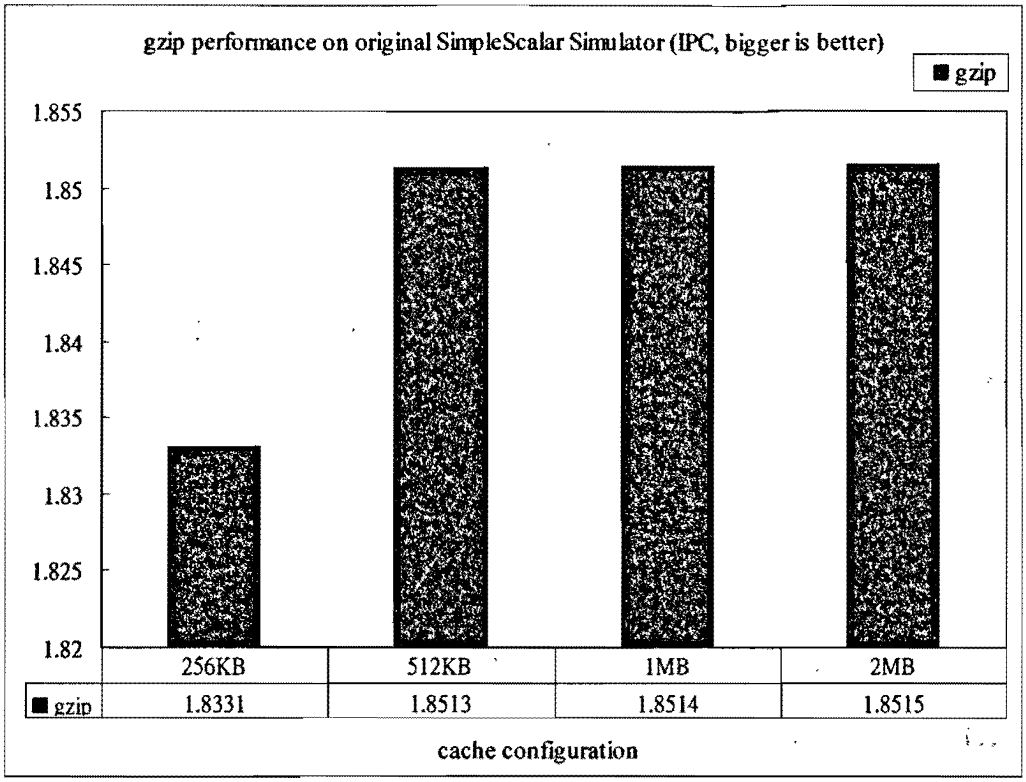


Figure 5.12: gzip simulations with varied L2 cache size on original SimpleScalar simulator

As suggested by figure 5.12, under the original SimpleScalar simulator running gzip, the effect of doubling the size of L2 cache after it reaches 512KB of size is minimal. This observation is supported by the following figure, which shows the total amount of memory accesses of each cache configuration:

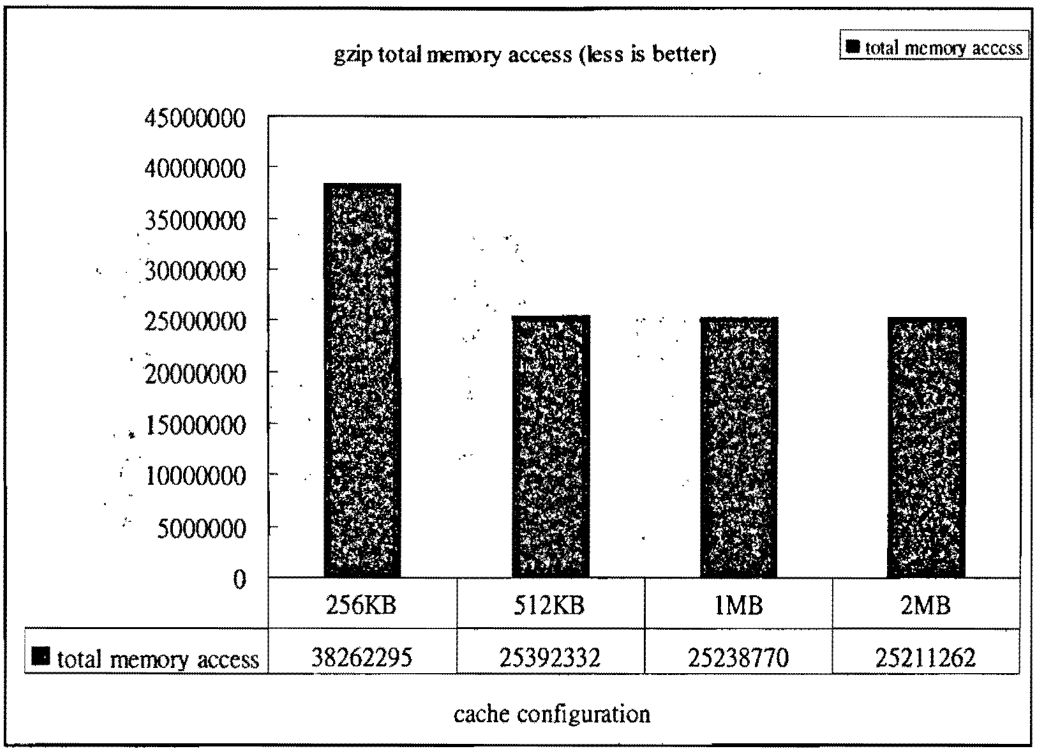
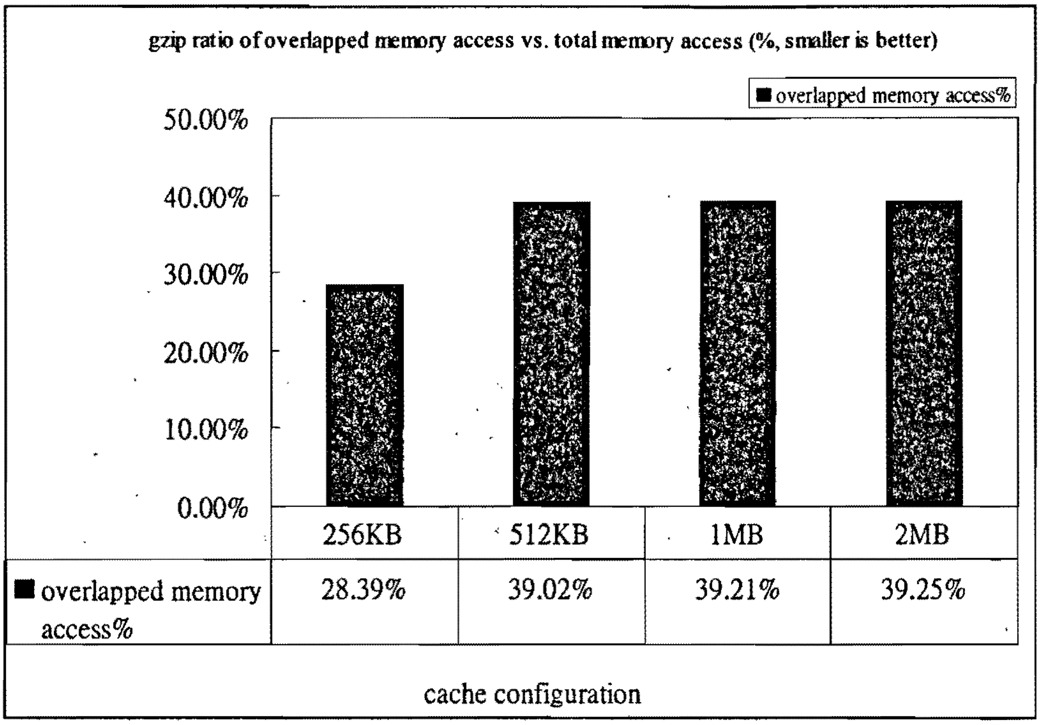


Figure 5.13: Total amount of gzip memory accesses

Therefore, we can conclude that gzip is reaching its peak performance when the L2 cache size reaches 512KB. And once the system reaches the point of diminishing return, factors such as the memory row refresh time and memory bus activity will start affecting the performance of the system, lowering the performance.

Throughout our study, we noticed that gzip generates a relatively large amount of overlapped memory accesses (as shown in figure 5.14), which happens when the cache is requesting data while the memory bus is busy servicing the previous memory access. This will cause the cache and processor to wait longer than regular memory accesses which happens when the memory data bus is idle.

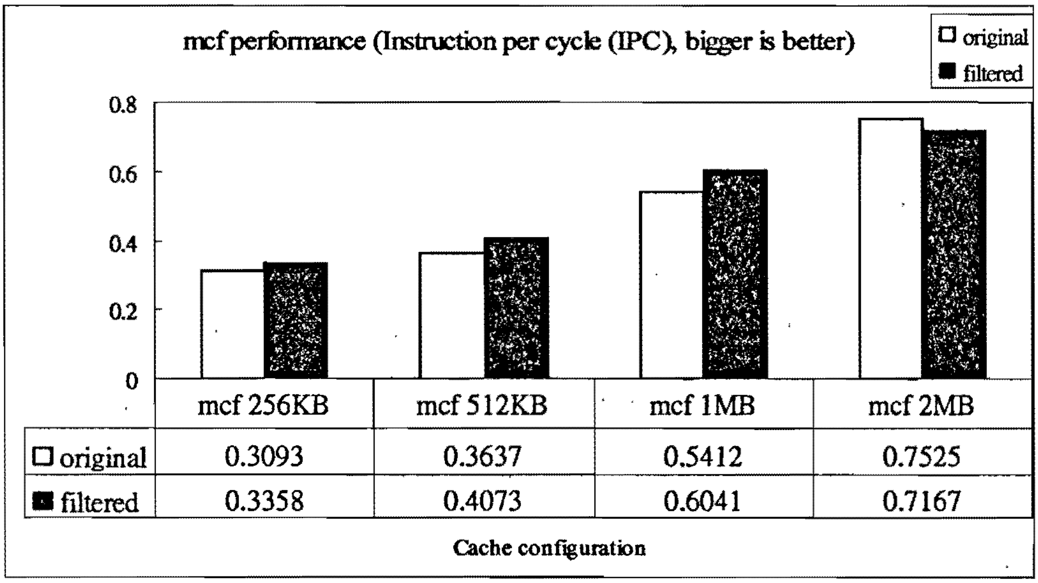


	256KB	512KB	1MB	2MB
■ overlapped memory access%	28.39%	39.02%	39.21%	39.25%

Figure 5.14: gzip ratio of overlapped memory access versus total memory accesses

Performance of mcf simulations

Figure 5.15 shows the performance of all mcf simulations in IPC:



	mcf 256KB	mcf 512KB	mcf 1MB	mcf 2MB
□ original	0.3093	0.3637	0.5412	0.7525
■ filtered	0.3358	0.4073	0.6041	0.7167

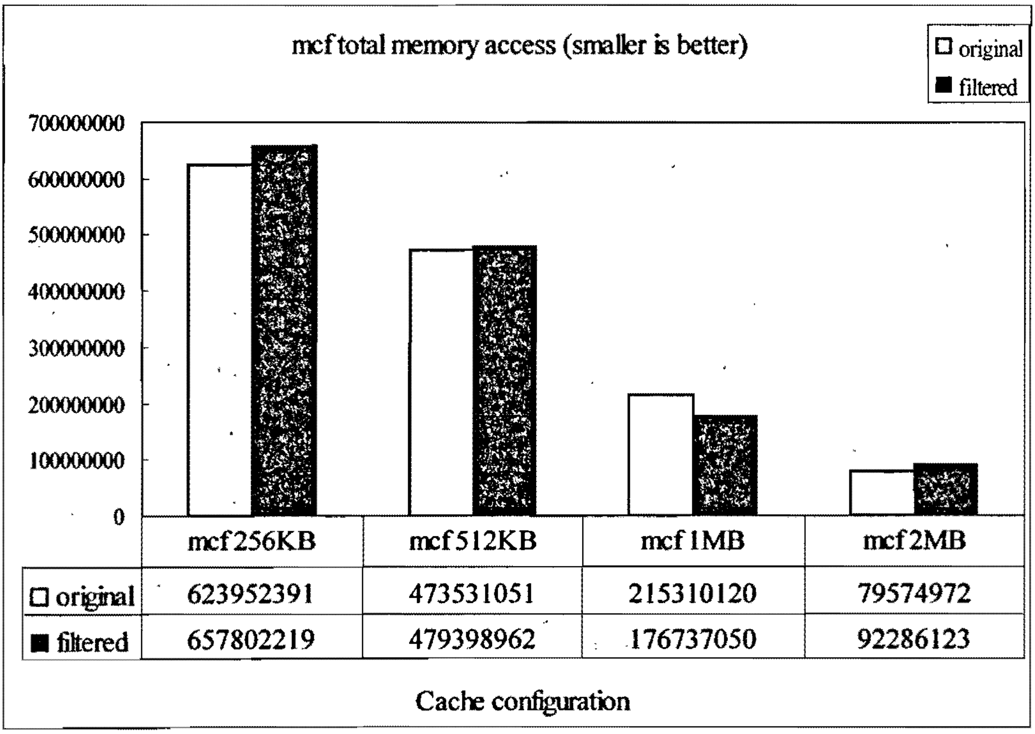
Figure 5.15: mcf performance in IPC

From the performance plot, the following observations were made:

- Both the original and cache filtered systems are having performance gain as the L2 cache size increases. However, the cache filtered system with 2MB L2 cache is showing negative performance gain comparing to the baseline.

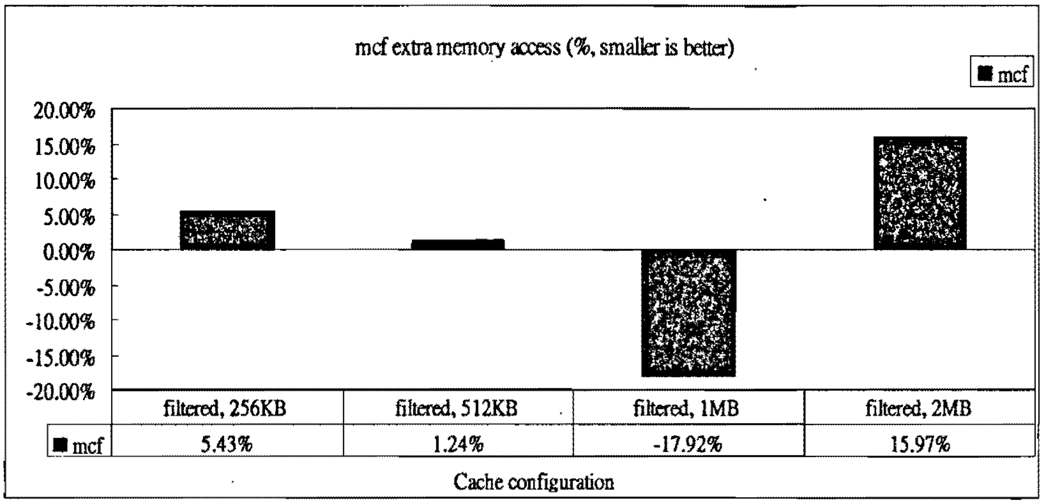
mcf total memory access count

Next, we will shift our focus towards the total amount of memory accesses and extra memory accesses happened throughout the execution of mcf (Figure 5.16 and 5.17).



	mcf256KB	mcf512KB	mcf1MB	mcf2MB
□ original	623952391	473531051	215310120	79574972
■ filtered	657802219	479398962	176737050	92286123

Figure 5.16: mcf total memory accesses



	filtered, 256KB	filtered, 512KB	filtered, 1MB	filtered, 2MB
■ mcf	5.43%	1.24%	-17.92%	15.97%

Figure 5.17: mcf extra memory access rate

According to figure 5.16 and 5.17, we can notice that:

- The total amount of memory accesses decreases as the L2 cache size increases. This is caused by the fact that the memory usage pattern of mcf is being classified by [23] to be having a very high degree of temporal locality.
- The cache filtered system is able to maintain a very small to negative extra memory access rate. However, as the L2 cache reaches 2MB, the benefit of doubling the L2 cache size shown by the baseline system has defeated the benefit offered by our cache filtering mechanism.

mcf memory row hit rate and filtered read versus total memory access ratio

Finally, due to the fact that our cache filter activates only when memory row misses (random accesses) happene, we will now show the memory row hit rate of the baseline and cache filtered systems in Figure 5.18:

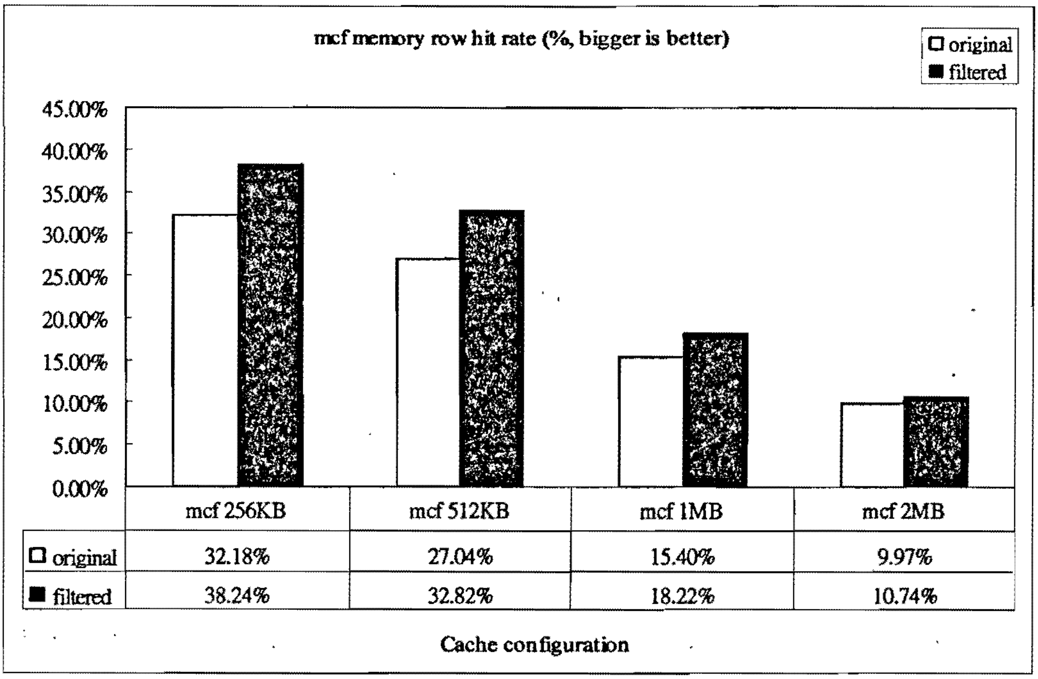


Figure 5.18: mcf memory row hit rate

As shown in figure 5.18 for both the baseline and the cache filtered systems, when the L2 cache size increases, the memory row hit rate drops. This is caused by the join effect of mcf's memory access pattern and the memory access intercepting capability of the bigger L2 cache. According to [23], mcf's memory access pattern is not only representing a very high temporal locality, but also a relatively low spatial locality where its requested data are located far from each other in the memory.

Therefore as the L2 cache is intercepting increasing amount of temporal accesses, the remaining address requests that are reaching the main memory will be located much further apart. If the size of the L2 cache maintains at a steady growth, eventually, memory address requests will always exceed the address range of the row latches, causing lowered row hit rate. This memory access pattern will also yield extra filtered read/write in our cache filtering scheme as the filter buffer with only 64 entries will also be unable to satisfy such largely diversified address requests.

With these findings, we can finalize our performance analyze with the followings:

- The performance gain of ammp is a result of the improved L2 cache and memory row hit rate, relatively small amount of extra memory accesses and the low filter buffer access latency.
- equake's performance gain is mainly contributed by the low filter buffer access latency, which is able to minimize the cache miss penalty.
- gcc's slight performance improvement is mainly caused by the improved memory row hit rate, which is capable of compensate the extra memory accesses.
- When running gzip, the cache filtered system will generate large amount of memory writes. However, the big gain in memory row hit rate had provided relieve to the situation, allowing the overall performance to be degraded slightly.
- mcf with 256KB to 1MB L2 cache's performance gain are mainly contributed by the small to negative amount of extra memory accesses and significantly better memory row and L2 cache hit rates.
- mcf with 2MB L2 cache's performance drop, relative to the other mcf simulations, is the end result of relatively small memory row hit improvement and the memory access pattern generated by the large L2 cache, which, is not favoring the cache filtered system.
- parser's performance gain is caused by the significantly better memory row hit rate.
- vortex's small performance improvement is the result of the better memory row hit rate.

To conclude, we have designed a cache filtering algorithm which is capable of:

- Distinguishing frequently used cache blocks from rarely used blocks;
- Filtering rarely used data out of the cache hierarchy;
- Improving the overall performance of the computer systems.

We suggest improving the filter buffer by changing the write through policy to the writeback policy, which will reduce the total amount of memory accesses, memory bandwidth usage and power consumption (more memory access implies more row precharging and DRAM activity, which in return, will consume more power). This modification should produce greater performance gain to the cache filtered system and allow the cache filtering algorithm to be implemented on value notebook computers where power consumption is a major concern.

Note: To verify our simulation results, we have included a sample verification for the cache filtered desktop system with 2MB L2 cache gzip simulation result in the appendix on page 110

Chapter 6

Overview and Comparison of Related Works

In this chapter, we will provide an overview of the following works and compare them with our work (hereafter referred as CF-LFU):

- Cache Filtering Techniques to Reduce the Negative Impact of Useless Speculative Memory References on Processor Performance (Spec) [5];
- Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines (Line distillation)[25];
- Reducing Cache Pollution via Dynamic Data Prefetch Filtering (prefetch filter) [26].

Due to the fact that out of the three related works to be compared in this chapter, more than one of them were not using instruction per cycle (IPC) and cache hit rates as the performance measurement. We will compare our work with their efficiency by using rate of performance improvement in percentage.

6.1 Cache Filtering Techniques to Reduce the Negative Impact of Useless Speculative Memory References on Processor Performance (Spec) [5]

O. Mutlu, H. Kim, D.N. Armstrong and Y.N. Patt proposed a cache filtering algorithm which uses the L1 cache as a filter to a particular type of useless data out of the L2 cache [5]. According to their concept, when the processor loads data to the cache during speculative execution mode (when the processor executes a program branch, which the branch predictor determined to have a high chance of being taken by the program), there is always a possibility that the branch predictor is making a wrong guess, rendering the data being loaded useless and causing cache pollution to both L1 and L2 caches.

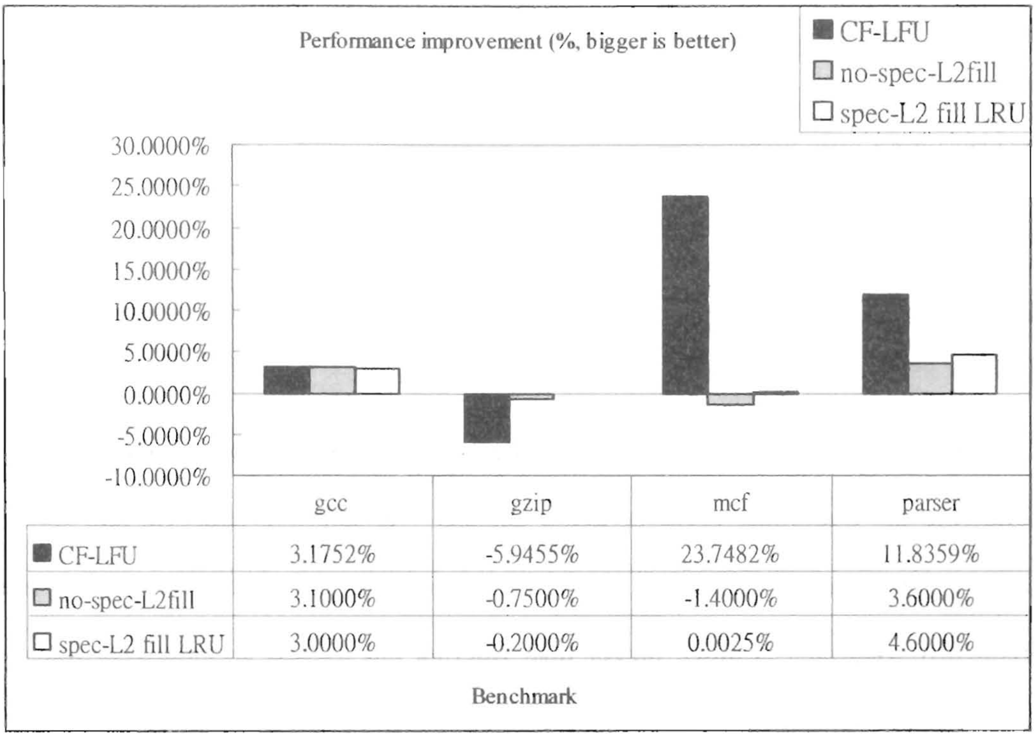
To address this issue, they designed a filtering mechanism with two filtering policies, the no-spec-L2fill policy and the spec-L2 fillLRU policy, that only fetches data to the L1 cache during speculative execution. If the processor determines that the branch prediction was taken, the speculatively fetched block will be allowed to be written back to the L2 cache when the block is being replaced, otherwise, under no-spec-L2fill policy, the speculatively fetched block will simply be discarded if the branch was not taken. If their spec-L2fill LRU policy was used, then the speculatively fetched L1 block will be written to the L2 cache index's least recently used set.

6.1.1 Performance improvement comparison (CF-LFU vs. Spec)

In this subsection, we will have a comparison between our CF-LFU and the two filtering policies, the no-spec-L2fill and spec-L2fillLRU, of [5]'s Spec filtering technique. The following is a table of simulation settings used to compare our CF-LFU with no-spec-L2fill and spec-L2fillLRU:

Table 6.1: Simulation settings for comparison between CF-LFU, no-spec-L2 fill and spec-L2 fill LRU

Fetch/Issue/Retire width	8 instructions, 8 functional units
Instruction window size	128 entry instruction window, 128 entry ld-st queue
Branch predictor	64K entry gshare, 64K entry PAs hybrid
L1 Instruction Cache	64KB, 4-way, 64B block size, LRU replacement
L1 Data Cache	64KB, 4-way, 64B block size, LRU replacement
L2 Unified Cache	512KB, 8-way, 64B block size, LRU replacement
Processor clock	3000Mhz
Memory clock	200Mhz
Memory bank configuration	2 channels, 16 banks per channel
Memory latency	6(CAS), 7(RCD), 7(RP), 21(RAS)
Execution range	Full execution
Benchmarks	gcc, gzip, mcf, parser



	gcc	gzip	mcf	parser
■ CF-LFU	3.1752%	-5.9455%	23.7482%	11.8359%
□ no-spec-L2fill	3.1000%	-0.7500%	-1.4000%	3.6000%
□ spec-L2 fill LRU	3.0000%	-0.2000%	0.0025%	4.6000%

Figure 6.1: Performance improvement comparison between CF-LFU, no-spec-L2fill and spec-L2fill LRU

Figure 6.1 shows that our cache filtering algorithm is able to keep a comparable performance gain in gcc, maintain significantly better performance gain in mcf and parser but a performance slowdown in gzip, comparing to O. Mutlu, H. Kim's, D.N. Armstrong's and Y.N. Patt's work. From the previous chapter, we are aware of the significant performance gains of the mcf and parser benchmark caused by the improved memory row hit rate at 512KB of L2 cache, as well as the performance drop of the gzip, which is a result of the big amount of filtered writes. Therefore, the results of figure 6.1 are proven to be consistent to our results shown in the previous chapter.

Figure 6.1 also suggests that out of the four benchmarks being compared, [5]'s work was only capable of producing positive performance gain in gcc and parser, and their spec-L2 fill LRU policy is capable of generating slightly better than the no-spec-L2 fill policy. According to [5], gcc and parser suffer most from L2 cache pollution; hence their work is capable of producing better results with those two benchmarks.

Our CF-LFU, on the other hand, was capable of producing better perofrmance gain by following the traditional focus of cache designs – to improve temporal and spatial locality. Support by figure 6.1, we can conclude that the benefit of filtering less frequently used data is much bigger than filtering out useless speculatively fetched data from the L2 cache.

6.2 Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines (Line distillation) [25]

M.K. Qureshi, M. A. Suleman and Y. N. Patt proposed a new technique (hereafter referred as Line distillation) which improves L2 cache's capacity by partitioning the L2 cache and only keep the useful words of a cache block upon eviction[25]. In their work, they pointed out that caches are organized into blocks where each block contains a sequence of consecutive words. This design is most suitable when applications are having high spatial locality in their memory usage pattern. However, if the application's spatial locality is low, most of the words in the cache block are not used and hence the cache capacity is not utililized efficiently.

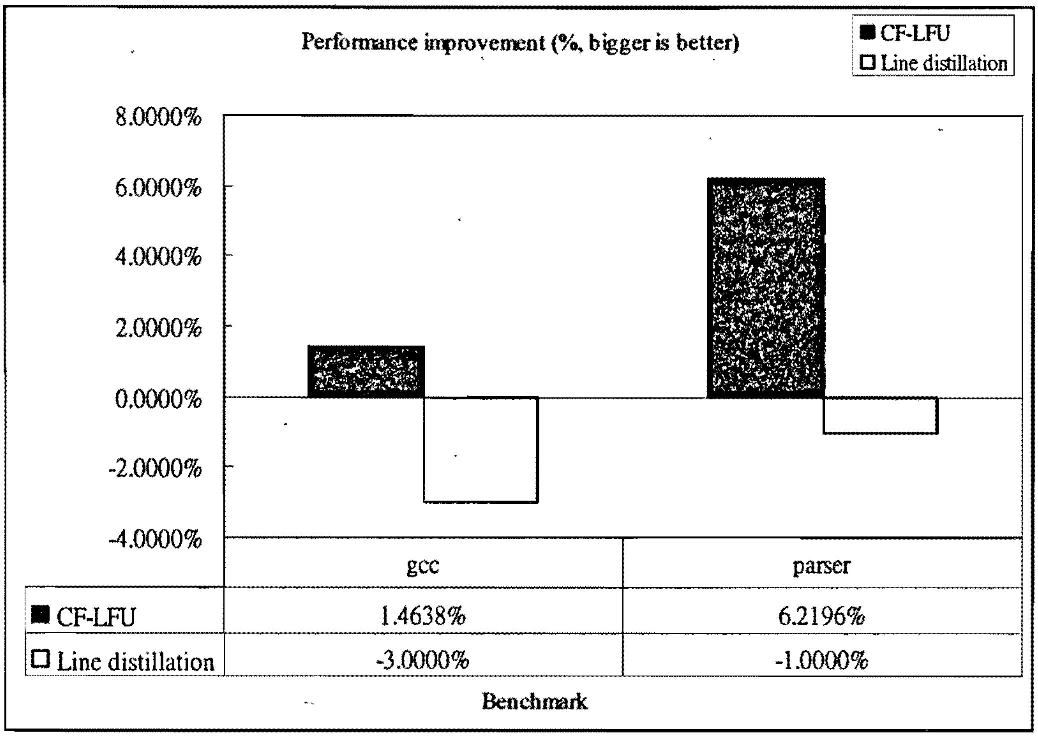
To solve this problem, they have a new cache design, known as distill cache, which partitions the cache into line organized cache (LOC) and word organized cache (WOC). The LOC is used to store standard cache blocks and has a footprint bit array to track the word usages of the cache block. The WOC, on the other hand, is used to store the previously used words of an evicted cache block. When a cache miss reaches the L2 distill cache, the LOC is first checked. If the LOC generates a hit, the block will be transferred to the L1 cache and the footprint array is updated. If the LOC generates a miss, the WOC is examined. If the WOC generates a hit, the WOC will transmit the word it contains to the L1 cache, together with a valid bit vector in order to mark which word of the L1 cache block is valid. However, if both the LOC and WOC are generating misses, the LOC will select a LRU block and the WOC will randomly select a block to replace. Upon replacement, the LOC will first wait for WOC to finish the writeback if needed, then the LOC will replace the LRU block with the request data and at the same time, transfer the previously used words to the WOC.

6.2.1 Performance improvemnet comparison between CF-LFU and Line distillation

In this section, we will perform the same efficiency comparison between our CF-LFU and [25]'s Line distillation. Note that [25] simulated their work with an execution range of 250 million instructions, i.e. from instruction 1 to instruction 250M, we will also simulate our work with the same range. Therefore, one should expect the result of our CF-LFU to be largely different from the previous sections where the entire program execution was simulated.

Table 6.2: Simulation settings for comparison between CF-LFU and Line distillation

Fetch/Issue/Retire width	8 instructions, 8 functional units
Instruction window size	128 entry instruction window, 128 entry ld-st queue
Branch predictor	64K entry gshare, 64K entry PAs hybrid
L1 Instruction Cache	16KB, 2-way, 64B block size, LRU replacement
L1 Data Cache	16KB, 2-way, 64B block size, LRU replacement
L2 Unified Cache	1MB, 8-way, 64B block size, LRU replacement
Processor clock	2000Mhz
Memory clock	100Mhz
Memory bank configuration	2 channels, 16 banks per channel
Memory latency	6(CAS), 7(RCD), 7(RP), 21(RAS)
Execution range	250M instructions
Benchmarks	gcc, parser



	gcc	parser
■ CF-LFU	1.4638%	6.2196%
□ Line distillation	-3.0000%	-1.0000%

Figure 6.2: Performance improvement comparison between CF-LFU and Line distillation

Figure 6.2 shows that our CF-LFU is capable of outperforming Line distillation with gcc and parser. According to [25], instructions on the wrong path can cause bigger usage of words stored in the cache block and reduce their Line distillation's performance. More importantly, [25] also mentioned that gcc is an instruction cache intensive benchmark, which caused their Line distillation algorithm to show slight performance drop. Our CF-LFU considers the entire cache block as a filtering unit and provides a filter buffer to hold more cache blocks. This approach, backed up by figure 6.2, is proven to yeild much better results when running the gcc and parser benchmarks.

6.3 Reducing Cache Polution via Dynamic Data Prefetch Filtering (Prefetch filter) [26]

In their work, Recuding Cache Polution via Dynamic Data Prefetch Filtering [26], X. Zhuang and H.S.Lee. pointed out that traditional cache prefetching mechanisms suffer from cache pollution caused by overly aggressive prefetches. To solve this problem, they introduced a new way of controlling cache prefetches through the use of branch predictors as prefetch pollution filters. According to [26], three types of branch predictors were used – bimodal predictors, two level bimodal predictors and gshare predictors. The predictors are organized as a 1D (bimodal) or 2D (2 level bimodal and gshare) array of 2bit counters and can be addressed by the least significant bits of either the missing cache block address (PA) or the program counter (PC), with the assistance of a branch histry register (for 2level bimodal and ghsare). Whenever a cache block is being evicted, the filter checks whether the cache block is being prefetched (indicated by a prefetch indicator bit) and whether the block was previously referenced (determined by a reference indication bit). If both conditions are true, the corresponding counter of the predictor will be increatmented and if the block was a prefetched block but has never been referenced, the corresponding counter will be decremented. Eventually, a prefetch history table is formed and the prefetcher can filter useless prefetches according to the table.

6.3.1 Performance improvement comparison between CF-LFU and Prefetch filter

In this section, we will perform the same efficiency comparison between our CF-LFU and [26]'s prefetch filter with both PA and PC branch prediction table addressing. Note that [26] simulated their work with an execution range of 300 million instructions, i.e. from instruction 1 to instruction 300M, we will also simulate our work with the same range.

Table 6.3: Simulation settings for comparison between CF-LFU and Prefetch filter

Fetch/Issue/Retire width	8 instructions per cycle
Instruction window size	128 entries instruction window, 64 entries load/store queue
Branch predictor	32K entry gshare, 32K entry PAs hybrid
L1 Instruction Cache	32KB, 4 way, 32B block size, LRU replacement
L1 Data Cache	32KB, 4 way, 32B block size, LRU replacement
L2 Unified Cache	512KB, 4 way, 32B block size, LRU replacement
Processor clock	2000Mhz
Memory clock	200Mhz
Memory bank configuration	2 channels, 16 banks per channel
Memory latency	1 (CAS), 1 (RCD), 1 (RP), 14 (RAS)
Execution range	300M instructions
Benchmarks	gcc, gzip

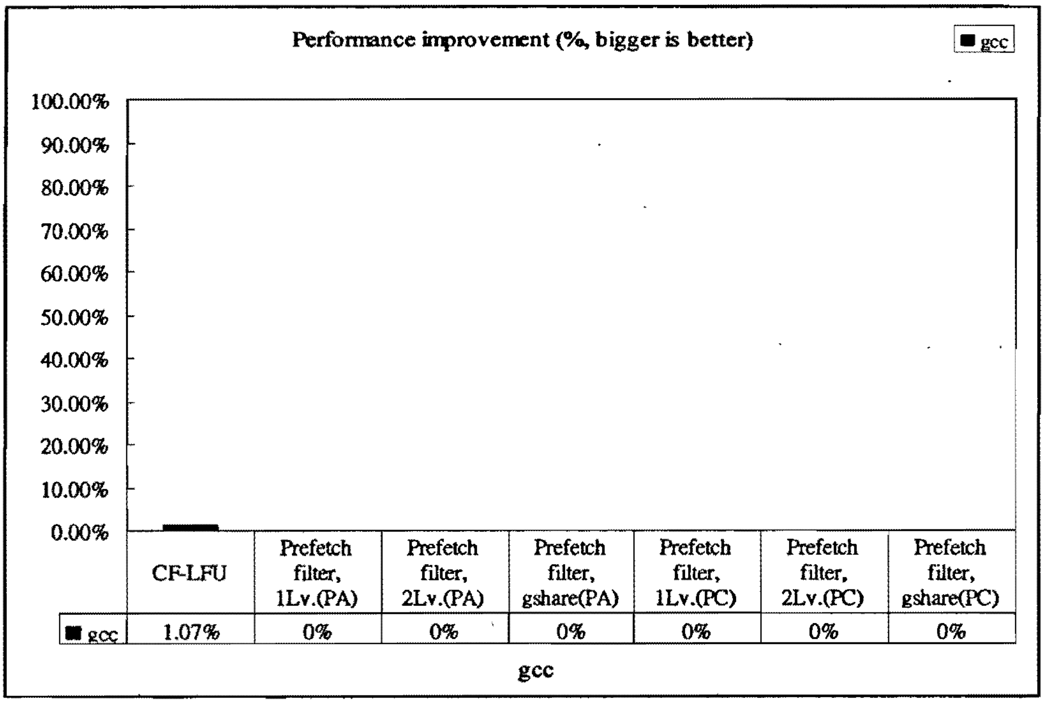


Figure 6.3: Performance improvement comparison between CF-LFU and prefetch filter for gcc

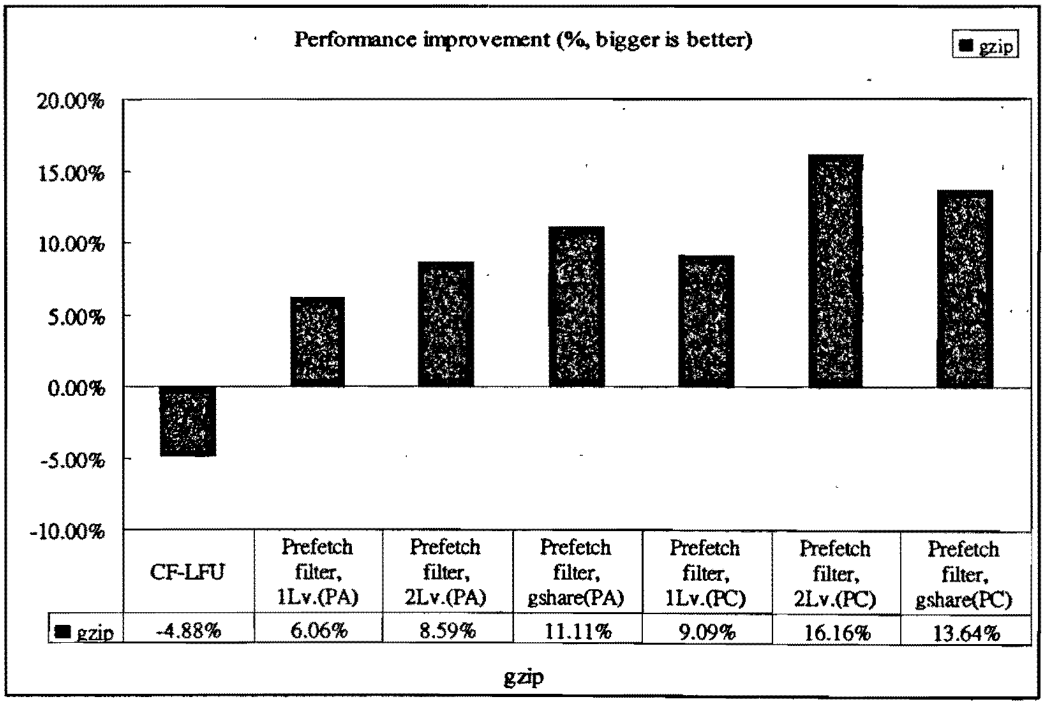


Figure 6.4: Performance improvement comparison between CF-LFU and prefetch filter for gzip

Figure 6.3 and 6.4 shows the performance improvement comparison between our CF-LFU and prefetch filter with both PC and PA settings. In terms of gcc, [26] mentioned that their prefetch filter happened to be reducing large amount of useful prefetches, which in turn, rendered no performance gain in gcc. Our CF-LFU, on the contrary, was able to keep a small performance gain which was attributed to the improvement of the memory row hit rate. Under gzip, prefetch filter showed significant improvement due to its nature of educated prefetching technique, especially when 2 level bimodal predictors are used. Our CF-LFU, however, was known to be generating extra memory accesses as large amount of filtered writes were involved. Therefore, we must agree that the prefetch filter is capable of producing better results when running gzip.

6.4 Summary of the comparison

In this chapter, we compared our Cache Filtering Algorithm for Least Frequently Used Data (CF-LFU) with other three related works, namely Cache Filtering Technique to Reduce the Negative Impact of Useless Speculative Memory Reference on Processor Performance (Spec), Line Distillation: Increasing Cache Capacity by Filtering Unused words in Cache Lines (Line distillation) and Reducing Cache Pollution via Dynamic Data Prefetch Filtering (Prefetch filer). Our CF-LFU focuses on filtering cache blocks that are not previously used and generate memory row misses. Spec tries to filter data fetches that are requested by untaken program branches out of the L2 cache. The Line distillation technique attemps to keep the previously accessed data words of evicted cache blocks in a separate partition of the L2 cache. Finally, Prefetch filter incorporates already well developed branch prediction techniques to determine whether a prefetch should be made. Although each of the related works is a true pioneer in the research field of cache design, we strongly believe, as proven by the simulated results, our Cache Filtering Algorithm for Least Frequently Used Data is comparable to them in terms of efficiency.

Chapter 7

Conclusion and future work

In this chapter, we will summarize our contributions and will propose the possible directions of the future work.

This thesis has successfully made two contributions – the Page Interleaved DDR3 Memory model for the SimpleScalar Simulator and Cache Filtering Algorithm for Least Frequently Used Data. The Page Interleaved DDR3 Memory model is a replacement for the SimpleScalar simulator's closed page SDRAM simulation model. By following the behavior of the DRAM and the DDR3 specification, our memory model is capable of generating consistent simulation results that resembles a real world computer system installed with dual channel DDR3 memory modules.

The cache filtering algorithm for least frequently used data is a performance optimization to the entire computer memory hierarchy. Its principle is to identify whether the processor requested data is frequently or rarely used, based on the contents of the filter buffer and the status of the memory rows. With the result of such analysis, the cache filtering algorithm will then make the decision to allow the data to be fetched to the cache hierarchy or to filter the data in order to prevent cache pollution.

As an extension to this thesis work, there are several directions that we are particularly interested in pursuing:

- Implement a filter buffer that uses a writeback policy to replace the write through scheme implemented by our current filter buffer model. With a writeback policy, the filter buffer should be able to eliminate majority of the memory accesses generated by the write through filter buffer. Hence, the filter buffer can conserve more bandwidth, reduce the overall power consumption of the cache filtering algorithm and allows the cache filtering algorithm to be more suitable for embedded and desktop systems with bigger amount of caches.
- Adding cache coherence and memory consistency protocols to the cache filter and filter buffer. By adding such protocols, the cache filter can be made compatible to modern shared memory multi processor systems.
- Implement a victim cache, write buffer and data prefetching unit – victim caches, write buffers and data prefetching units are fundamental elements of modern memory hierarchies. Adding these components allows us to complete SimpleScalar's memory hierarchy and simplify future computer architecture and memory system researches involving the SimpleScalar simulator.

Bibliography

[1] J. L Hennessy and D. A. Patterson. *Computer architecture: A quantitative approach*, 4th Ed. Morgan Kaufmann Publishers, In, 2007

[2] B. Jacob, S. W. Ng and D. T. Wang. *Memory systems: Cache, DRAM, Disk.* Morgan Kaufmann Publishers, In, 2008

[3] D. Burger, J. R. Goodman and A. Kagi. "Memory bandwidth limitations of future microprocessors." *Proc. 23rd Int'l Symp. Computer Architecture (ISCA 96)*, ACM Press, 1996, pp. 90-101

[4] N. Mekhiel. "Multi-level cache with most frequently used policy: a new concept in cache design." *In International Conference on Computer Applications in Industry and Engineering*, Nov 1995.

[5] O. Multu, H. Kim, D. N. Armstrong and Y. N. Patt. "Cache filtering technique to reduce the negative impact of useless speculative memory references on processor performance." *In 16th Symposium on Computer Architecture and High Performance Computing*, Oct 2004.

[6] B. Jacob. "A case for studying DRAM issues at the system level." *IEEE Micro* 23(4): 44-56 July - Aug2003

[7] Z. Zhang, Z. Zhu and X. Zhang. "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality." In Proceedings of the *33rd Annual International Symposium on Microarchitecture (Micro-33)*, Monterey, California, December 10-12, 2000, pp 32-41.

[8] N. Mekhiel, "Methods for improving main memory performance." *ISCA International Conference on Computer Applications in Industry and Engineering Honolulu*, Hawaii, December 15-17, 1993.

[9] V. Cuppu, B. Jacob, B. Davis and T. Mudge. "A performance comparison of contemporary DRAM architectures." *Proc. 26th International Symposium on Computer Architecture (ISCA 1999)*, pp. 222-233. Atlanta GA, May 1999.

[10] V. Cuppu, B. Jacob, B. Davis and T. Mudge. "High performance DRAMs in workstation environments." *IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1133-1153. November 2001. (TC Special Issue on High-Performance Memory Systems)

[11] N. Mekhiel. "LHA: Latency hiding algorithm for DRAM." *2nd Annual Workshop on Memory Performance Issue (WMPI 2002)*. Held in conjunction with the ISCA 2002, May 25, 2002, Anchorage, Alaska.

[12] D. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0." *In Computer Architecture News*, 25 (3), pp. 13-25, June, 1997.

[13] I. Kim. "Macro-op scheduling and execution." Ph.D. diss., University of Wisconsin-Madison, United States, 2004.

[14] R. Srinivasan. "Techniques for accelerating microprocessor simulation." M.Sc. thesis, New Mexico State University, United States, 2004.

[15] Brigham Young University Trace Distribution Center, (http://tds.cs.byu.edu/tds/)

[16] J. Xiao. "Location-based key management, data authentication and aggregation in wireless sensor networks". MASc. thesis, Ryerson University, Canada, 2006.

[17] D. E. Culler, J. P. Singh and A. Gupta. *Parallel computer architecture : A hardware / software approach.* Morgan Kaufmann Publishers, In, 1999

[18] M. Gries and A. Romer: "Performance Evaluation of Recent DRAM Architectures for Embedded Systems." TIK Report Nr. 82, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, November, 1999

[19] M. Postiff, D. Greene, C. Lefurgy, D. Helder and T. Mudge. *The MIRV SimpleScalar/PISA Compiler*. University of Michigan EECS Department Tech. Report CSE-TR-421-00. April 2000.

[20] Standard Performance Evaluation Corporation. SPEC CPU2000 version 1.30, January 2005

[21] N. Mekhiel, "Cache Filtering Method and Apparatus", Patent Application 1301-01US-00-75. September 2008.

[22] N. Mekhiel, "Methods and Apparatus for Accelerating Retrieval of Data from a Memory System with Cache by Reducing latency", Patent No. US7318123, Jan 08, 2008, Patent No. US 6,892,279 B2 May 10, 2005.

[23] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques", *IEEE Trans. Comput.*, vol. 48, pp. 1260 - 1281, 1999

[24] R. C. Murphy and P. M. Kogge, "On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications", *IEEE Trans. Comput.* vol. 56, pp. 937 – 945, 2007

[25] M. K. Qureshi, M. A. S and Y. N. Patt, "Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines", *13th International Symposium on High Performance Computer Architecture (HCPA 2007)*, pp.250 – 259, Scottsdale, AZ, Feb 2007.

[26] X. Zhuang and H. H. S. Lee, "Reducing cache pollution via dynamic data prefetch filtering", *IEEE Trans. Comput.*, vol. 56, pp. 18 2007.

[27] S. H. Chitra and P. T. Vanathi, "Design and Analysis of Dynamically Configurable Bus Arbiters for SoCs", *Journal of Programmable Devices, Circuits and Systems (ICGST-PDCS 2008)*, vol. 8, Issue 1, pp. 45 – 52, Dec 2008.

Appendix

A.1 Sample verification of cache filtered desktop system with 2MB L2 cache running gzip

To prove the simulator's output and our performance analyze conclusions are correct, we have included a cache filtered gzip simulation with 2MB L2 cache simulation output file (Figure A.1) on page 116:

From the file, we are able to extract the following information:

Total memory random reads = 10051787

Total memory random writes = 9874222

Total memory fast page reads = 5233822

Total memory fast page writes = 7554673

Total memory reads = 10051787 + 5233822 = 15285609

Total memory writes = 9874222 + 7554673 = 17428895

Total filter buffer reads = 9917372

Total filter buffer memory fetches (DL1 filtered read + IL1 filtered read) = 2497325

Total filter buffer memory writes = 17428895

I L1 misses = 111476

I L1 filtered accesses = 4959

D L1 misses = 607825571

D L1 writebacks = 164522187

D L1 filtered accesses (D L1 filtered read + D L1 filtered write) = 10046828

L2 accesses = 762407441

L2 misses = 15151194

L2 writebacks = 9874427

First, we must prove that the numbers extracted from the simulation file are correct. According to the cache filtering algorithm, only data L1 cache's filtered writebacks and the L2 cache's writebacks are written directly to the filter buffer then to the main memory. Hence, we must first compare the filter buffer's memory write with the total memory write:*

Total filter buffer memory writes = 17428895

Total memory writes

= total memory random writes + total memory fast page writes

= 9874222 + 7554673 = 17428895

Since the value of total memory write equals to the value of total filter buffer memory writes, we can conclude that these two numbers are correct.

Secondly, we must check whether the total L2 fetches from the main memory equals to the total amount of fast page reads. Since the L2 cache is the entry point for data to enter the cache hierarchy, if the values are equal, then the simulator's operation is following the cache filtering algorithm's criteria: only data from an opened memory page or from the filter buffer can enter the cache hierarchy.

L2 fetches from main memory= L2 misses – total filter buffer reads

= 15151194 – 9917372 = 5233822

Total fast page memory reads = 5233822

*Note: since the filter buffer and cache are defined in the cache.c source file and the main memory is defined in the memory.c source file, if we can prove the values are equal, it will automatically implies that the program code written in both of the files are correct.

As the value of the L2 fetches from main memory is matching the total amount of fast page memory reads, it is reasonable to state that the value of L2 cache miss, filter buffer read and total fast page memory reads are all correct.

Next, we must prove that the total filtered fetch is equal to the total amount of random read. Note that filtered fetch happens in both filtered read and filtered write. This is because all filtered write must start by performing a random read as discussed earlier. Also, the instruction L1 cache can only perform reading, hence we can conclude that the I L1 filter count is 100% filtered read.

Total filtered fetch = DL1 filtered read + D L1 filtered write + I L1 filtered read

= DL1 filter count + IL1 filter count

= 10046828 + 4959 = 10051787

Total memory random read = 10051787

This allows us to conclude that the total random read and the total amount of cache filtering are correct. At this point, the data extracted by the simulation output file are all correct.

We must now determine the total amount of data L1 cache's filtered read/write/writebacks, data L1 cache's writeback to the L2 cache, data L1 cache's fetch from the L2 cache, instruction L1 cache's fetch from L2 cache and the total amount of fast page writes that is not a part of a filtered write (pure fast page write):

Data L1 cache's filtered read = total filter buffer memory fetch - I L1 filtered (read)

= 2497325 – 4959 = 2492366

Data L1 cache's filtered write (also a part of the total fast page write and random reads)

= total D L1 filtered (access) – data L1 cache's filtered read

= 10046828 – 2492366 = 7554462

Instruction L1 cache's fetch from L2 = IL1 miss – total IL1 filtered (read)

= 111476 – 4959 = 106517

Total L2 accesses that are requested by data L1 cache

= L2 access – IL1 cache's fetch from L2

= 762407441 – 106517 = 762300924

Total data L1 cache fetch from L2 = DL1 misses – DL1 filtered access

= 607825571 – 10046828 = 597778743

Total data L1 cache writebacks that's written to L2

= total L2 accesses that are requested by DL1 - total DL1 cache fetch from L2

= 762300924 – 597778743 = 164522181

Total data L1 cache's filtered writebacks

= total DL1 writebacks – total DL1 writebacks that's written to L2

= 164522187 – 164522181 = 6

Pure fast page writes = Total fast page writes – DL1 filtered write

= 7554673 – 7554462 = 211

With these values, we can once again check whether the total amount of each memory access types (random read/write and fast page read/write) are matching the total amount of each filter access (data L1 cache's filtered read/write/writeback, instruction L1 cache's filtered read and L2 cache's filtered writebacks).

According to the cache filtering algorithm's definition, the total amount of memory writes should equal to the total amount of filter buffer memory writes. At the same time, it should also equal to the following:

The total amount of data L1 cache's filtered write + data L1 cache's filtered writebacks + L2 cache's (filtered) writebacks = 7554462 + 6 + 9874427 = 17428895.

Comparing this value to the total amount of memory writes, which is 17428895, we can conclude that the total amount of data L1 cache's filtered writes/writebacks and L2 cache's filtered writebacks are correct.

Finally, if we add the total amount of random writes and pure fast page writes, the value should equal to the sum of the total amount of L2 writebacks and data L1 cache's filtered writebacks:

Total amount of random writes + total amount of pure fast page writes

= 9874222 + 211 = 9874433

Total amount of L2 writebacks + total amount of data L1 cache's filtered writebacks

= 9874427 + 6 = 9874433

Since the results of the above calculations are matched, we can finally conclude that our simulation result is totally correct.

To prove our statement about the gzip having a massive amount of filtered writes in the desktop/notebook performance analyze, we did the same calculation to find out the total amount of filtered write of the cache filtered system with 2MB L2 cache running equake (equake has a close-to-zero filter buffer entry wait time) and the total amount of equake's filtered write is 537676. Comparing such value with gzip's 7554462 filtered writes, one can conclude that gzip's filtered write count is 14 times bigger than equake's, which can justify our statement immediately.

```
sim-outorder: SimpleScalar/PISA Tool Set version 3.0 of August, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
All Rights Reserved. This version of SimpleScalar is licensed for academic
non-commercial use.   No portion of this work may be used by any commercial
entity, or for any commercial purpose, without the prior written permission
of SimpleScalar, LLC (info@simplescalar.com).

sim: command line: ./sim-outorder -cache:il1 il1:128:64:4:l -cache:dl1 dl1:128:64:4:l
-cache:dl2 ul2:4096:64:8:l -cache:il2 dl2 -cache:il1lat 1 -cache:dl1lat 1 -cache:dl2lat
10 -c
ache:il1_filter TRUE -cache:dl1_filter TRUE -cache:il2_filter TRUE -cache:dl2_filter
TRUE -filtbuf:size 64 -filtbuf:searchlat 1 -filtbuf:transferlat 2 -cpu:clk 3200 -mem:clk
800 -
mem:lat 7 7 7 21 -mem:width 8 -mem:channel 1 -mem:bank 8 -mem:map 2
-mem:type 2 -tlb:itlb itlb:1:4096:128:l -tlb:dtlb dtlb:1:4096:128:l -tlb:lat 30 -res:ialu 3
-res:imult 2 -res:m
emport 2 -res:fpalu 4 -res:fpmult 1 -fetch:ifqsize 64 -fetch:mplat 2 -fetch:speed 1
-bpred bimod -bpred:bimod 4096 -bpred:2lev 1 1024 12 0 -bpred:comb 1024
-bpred:ras 32 -bpred:bt
b 2048 2 -decode:width 4 -issue:width 4 -issue:inorder false -issue:wrongpath true
-commit:width 4 -ruu:size 16 -lsq:size 8 -redir:sim
out_results/memmod/filter/gzip/l1_l2/cache/g
zip_sim_2M_channel1_bank8_buf64.out -redir:prog
out_results/memmod/filter/gzip/l1_l2/cache/gzip_prog_2M_channel1_bank8_buf64.o
ut SPEC/gzip_train/gzip.ss SPEC/gzip_train/input.comb
ined 32

sim: simulation started @ Sun Oct 25 12:21:32 2009, options follow:

sim-outorder: This simulator implements a very detailed out-of-order issue
superscalar processor with a two-level memory system and speculative
execution support.   This simulator is a performance simulator, tracking the
latency of all pipeline operations.

# -config                          # load configuration from a file
# -dumpconfig                      # dump configuration to a file
# -h                         false # print help message
# -v                         false # verbose operation
# -d                          false # enable debug message
# -i                          false # start in Dlite debugger
-seed                             1 # random number generator seed (0 for timer
seed)
# -q                          false # initialize and terminate immediately
# -chkpt                     <null> # restore EIO trace execution from <fname>
# -redir:sim
out_results/memmod/filter/gzip/l1_l2/cache/gzip_sim_2M_channel1_bank8_buf64.ou
t # redirect simulator output to file (non-interactive only)
# -redir:prog
out_results/memmod/filter/gzip/l1_l2/cache/gzip_prog_2M_channel1_bank8_buf64.o
ut # redirect simulated program output to file
-nice                             0 # simulator scheduling priority
-max:inst                         0 # maximum number of inst's to execute
-fastfwd                          0 # number of insts skipped before timing starts
# -ptrace                    <null> # generate pipetrace, i.e., <fname|stdout|stderr>
<range>
-fetch:ifqsize                   64 # instruction fetch queue size (in insts)
-fetch:mplat                      2 # extra branch mis-prediction latency
-fetch:speed                      1 # speed of front-end of machine relative to
execution core
-bpred                        bimod # branch predictor type
{nottaken|taken|perfect|bimod|2lev|comb}
-bpred:bimod          4096 # bimodal predictor config (<table size>)
-bpred:2lev        1 1024 12 0 # 2-level predictor config (<l1size> <l2size>
<hist_size> <xor>)
-bpred:comb           1024 # combining predictor config (<meta_table_size>)
-bpred:ras                       32 # return address stack size (0 for no return stack)
-bpred:btb           2048 2 # BTB config (<num_sets> <associativity>)
# -bpred:spec_update         <null> # speculative predictors update in {ID|WB}
(default non-spec)
-decode:width                     4 # instruction decode B/W (insts/cycle)
-issue:width                      4 # instruction issue B/W (insts/cycle)
-issue:inorder                false # run pipeline with in-order issue
-issue:wrongpath               true # issue instructions down wrong execution paths
-commit:width                     4 # instruction commit B/W (insts/cycle)
-ruu:size                        16 # register update unit (RUU) size
-lsq:size                         8 # load/store queue (LSQ) size
```

```
-cache:dl1          dl1:128:64:4:l # l1 data cache config, i.e., {<config>|none}
-cache:dl1lat                 1 # l1 data cache hit latency (in cycles)
-cache:dl2          ul2:4096:64:8:l # l2 data cache config, i.e., {<config>|none}
-cache:dl2lat                10 # l2 data cache hit latency (in cycles)
-cache:il1          il1:128:64:4:l # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1lat                 1 # l1 instruction cache hit latency (in cycles)
-cache:il2                  dl2 # l2 instruction cache config, i.e.,
{<config>|dl2|none}
-cache:il2lat                 6 # l2 instruction cache hit latency (in cycles)
-cache:flush              false # flush caches on system calls
-cache:icompress          false # convert 64-bit inst addresses to 32-bit inst
equivalents
-mem:lat            7 7 7 21 # memory access latency (<Tcas> <Trcd><Trp><Tras>)
-mem:lat2           8 5 1 1 7 4 # memory access latency (<Tburst>, <Twr>, <Trtrs>,
<Tcmd>, <Tcwd>, <Tccd>)
-mem:width                    8 # memory access bus width (in bytes)
-mem:channel                  1 # memory controller channel (1 or 2)
-mem:bank                     8 # amount of memory banks (1 - 16)
-mem:map                      2 # memory address-bank mapping method, (1 for
simple linear, consecutive rows are mapped to the same bank;2 for page interleaving
-mem:type                     2 # memory type, 1 for SDR, 2 for DDR3
-cpu:clk              3200.0000 # processor clock.
-mem:clk               800.0000 # memory clock.
-cache:il1_filter          true # Activate instruction L1 cache filtering (TRUE or
FALSE)
-cache:dl1_filter          true # Activate data L1 cache filtering (TRUE or FALSE)
-cache:il2_filter          true # Activate instruction L2 cache filtering (TRUE or
FALSE)
-cache:dl2_filter          true # Activate data L2 cache filtering (TRUE or FALSE)
-filtbuf:size                64 # Size of filter buffer
-filtbuf:searchlat            1 # filter buffer's latency for searching for an empty or
earliest finish entry
-filtbuf:transferlat          2 # filter buffer's transfer latency
-tlb:itlb           itlb:1:4096:128:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb           dtlb:1:4096:128:l # data TLB config, i.e., {<config>|none}
-tlb:lat                     30 # inst/data TLB miss latency (in cycles)
-res:ialu                     3 # total number of integer ALU's available
-res:imult                    2 # total number of integer multiplier/dividers
```

```
available
-res:memport                2 # total number of memory system ports available
(to CPU)
-res:fpalu                 4 # total number of floating point ALU's available
-res:fpmult                 1 # total number of floating point multiplier/dividers
available
# -pcstat              <null> # profile stat(s) against text addr's (mult uses ok)
-bugcompat               false # operate in backward-compatible bugs mode (for
testing only)
```

Pipetrace range arguments are formatted as follows:

```
{{@|#}<start>}:{{@|#|+}<end>}
```

Both ends of the range are optional, if neither are specified, the entire execution is traced. Ranges that start with a '@' designate an address range to be traced, those that start with an '#' designate a cycle count range. All other range values represent an instruction count range. The second argument, if specified with a '+', indicates a value relative to the first argument, e.g., 1000:+100 == 1000:1100. Program symbols may be used in all contexts.

```
Examples:    -ptrace FOO.trc #0:#1000
             -ptrace BAR.trc @2000:
             -ptrace BLAH.trc :1500
             -ptrace UXXE.trc :
             -ptrace FOOBAR.trc @main:+278
```

Branch predictor configuration examples for 2-level predictor:

```
Configurations:   N, M, W, X
   N    # entries in first level (# of shift register(s))
   W    width of shift register(s)
   M    # entries in 2nd level (# of counters, or other FSM)
   X    (yes-1/no-0) xor history and address for 2nd level index
Sample predictors:
   GAg      : 1, W, 2^W, 0
   GAp      : 1, W, M (M > 2^W), 0
   PAg      : N, W, 2^W, 0
```

```
      PAp       : N, W, M (M == 2^(N+W)), 0
      gshare    : 1, W, 2^W, 1
  Predictor `comb' combines a bimodal and a 2-level predictor.

  The cache config parameter <config> has the following format:

    <name>:<nsets>:<bsize>:<assoc>:<repl>

    <name>    - name of the cache being defined
    <nsets>   - number of sets in the cache
    <bsize>   - block size of the cache
    <assoc>   - associativity of the cache
    <repl>    - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random

    Examples:   -cache:dl1 dl1:4096:32:1:l
                 -dtlb dtlb:128:4096:32:r

  Cache levels can be unified by pointing a level of the instruction cache
  hierarchy at the data cache hiearchy using the "dl1" and "dl2" cache
  configuration arguments.   Most sensible combinations are supported, e.g.,

    A unified l2 cache (il2 is pointed at dl2):
      -cache:il1 il1:128:64:1:l -cache:il2 dl2
      -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

    Or, a fully unified cache hierarchy (il1 pointed at dl1):
      -cache:il1 dl1
      -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l



sim: ** starting performance simulation **

sim: ** simulation statistics **
sim_num_insn                 77693332344 # total number of instructions committed
sim_num_refs                 26629788025 # total number of loads and stores
committed
sim_num_loads                19265035676 # total number of loads committed
```

```
sim_num_stores              7364752349.0000 # total number of stores committed
sim_num_branches              11519867492 # total number of branches committed
sim_elapsed_time                    96247 # total simulation time in seconds
sim_inst_rate                 807228.6133 # simulation speed (in insts/sec)
sim_total_insn                84601667750 # total number of instructions executed
sim_total_refs               29095252565 # total number of loads and stores executed
sim_total_loads               21117777015 # total number of loads executed
sim_total_stores          7977475550.0000 # total number of stores executed
sim_total_branches            12502422295 # total number of branches executed
sim_cycle                     44717071765 # total simulation time in cycles
sim_IPC                            1.7374 # instructions per cycle
sim_CPI                            0.5756 # cycles per instruction
sim_exec_BW                         1.8919 # total instructions (mis-spec +
committed) per cycle
sim_IPB                            6.7443 # instruction per branch
IFQ_count                   1566892957508 # cumulative IFQ occupancy
IFQ_fcount                    13518869642 # cumulative IFQ full count
ifq_occupancy                     35.0402 # avg IFQ occupancy (insn's)
ifq_rate                           1.8919 # avg IFQ dispatch rate (insn/cycle)
ifq_latency                       18.5208 # avg IFQ occupant latency (cycle's)
ifq_full                           0.3023 # fraction of time (cycle's) IFQ was full
RUU_count                    614946697909 # cumulative RUU occupancy
RUU_fcount                    28473471729 # cumulative RUU full count
ruu_occupancy                     13.7519 # avg RUU occupancy (insn's)
ruu_rate                           1.8919 # avg RUU dispatch rate (insn/cycle)
ruu_latency                        7.2687 # avg RUU occupant latency (cycle's)
ruu_full                           0.6367 # fraction of time (cycle's) RUU was full
LSQ_count                    225685433941 # cumulative LSQ occupancy
LSQ_fcount                     8038603209 # cumulative LSQ full count
lsq_occupancy                      5.0470 # avg LSQ occupancy (insn's)
lsq_rate                           1.8919 # avg LSQ dispatch rate (insn/cycle)
lsq_latency                        2.6676 # avg LSQ occupant latency (cycle's)
lsq_full                           0.1798 # fraction of time (cycle's) LSQ was full
sim_slip                -2578299233055046588 # total number of slip cycles
avg_sim_slip                -33185592.0614 # the average slip between issue and
retirement
bpred_bimod.lookups           15674898831 # total number of bpred lookups
bpred_bimod.updates           11519867492 # total number of updates
bpred_bimod.addr_hits       10644257456 # total number of address-predicted hits
bpred_bimod.dir_hits        10661282975 # total number of direction-predicted hits
(includes addr-hits)
bpred_bimod.misses            858584517 # total number of misses
bpred_bimod.jr_hits           324501052 # total number of address-predicted hits for
JR's
bpred_bimod.jr_seen           341522178 # total number of JR's seen
bpred_bimod.jr_non_ras_hits.PP          52348 # total number of address-predicted
hits for non-RAS JR's
bpred_bimod.jr_non_ras_seen.PP         160704 # total number of non-RAS JR's
seen
bpred_bimod.bpred_addr_rate       0.9240 # branch address-prediction rate (i.e.,
addr-hits/updates)
bpred_bimod.bpred_dir_rate       0.9255 # branch direction-prediction rate (i.e.,
all-hits/updates)
bpred_bimod.bpred_jr_rate       0.9502 # JR address-prediction rate (i.e., JR
addr-hits/JRs seen)
bpred_bimod.bpred_jr_non_ras_rate.PP       0.3257 # non-RAS JR addr-pred rate (ie,
non-RAS JR hits/JRs seen)
bpred_bimod.retstack_pushes       469998938 # total number of address pushed onto
ret-addr stack
bpred_bimod.retstack_pops       454526877 # total number of address popped off of
ret-addr stack
bpred_bimod.used_ras.PP       341361474 # total number of RAS predictions used
bpred_bimod.ras_hits.PP       324448704 # total number of RAS hits
bpred_bimod.ras_rate.PP       0.9505 # RAS prediction rate (i.e., RAS hits/used RAS)
il1.accesses               104219153196 # total number of accesses
il1.hits                   104219041720 # total number of hits
il1.misses                       111476 # total number of misses
il1.replacements                 106005 # total number of replacements
il1.writebacks                        0 # total number of writebacks
il1.invalidations                     0 # total number of invalidations
il1.miss_rate                    0.0000 # miss rate (i.e., misses/ref)
il1.repl_rate                    0.0000 # replacement rate (i.e., repls/ref)
il1.wb_rate                      0.0000 # writeback rate (i.e., wrbks/ref)
il1.inv_rate                     0.0000 # invalidation rate (i.e., invs/ref)
il1.filtered                       4959 # total filtered access
il1.filtered_read                     0 # total filtered read
```

```
il1.filtered_write                 0 # total filtered write
dl1.accesses              25920318799 # total number of accesses
dl1.hits                  25312493228 # total number of hits
dl1.misses                  607825571 # total number of misses
dl1.replacements            597778231 # total number of replacements
dl1.writebacks              164522187 # total number of writebacks
dl1.invalidations                   0 # total number of invalidations
dl1.miss_rate                  0.0234 # miss rate (i.e., misses/ref)
dl1.repl_rate                  0.0231 # replacement rate (i.e., repls/ref)
dl1.wb_rate                    0.0063 # writeback rate (i.e., wrbks/ref)
dl1.inv_rate                   0.0000 # invalidation rate (i.e., invs/ref)
dl1.filtered                 10046828 # total filtered access
dl1.filtered_read                   0 # total filtered read
dl1.filtered_write                  0 # total filtered write
ul2.accesses                762407441 # total number of accesses
ul2.hits                    747256247 # total number of hits
ul2.misses                   15151194 # total number of misses
ul2.replacements             15118426 # total number of replacements
ul2.writebacks                9874427 # total number of writebacks
ul2.invalidations                   0 # total number of invalidations
ul2.miss_rate                  0.0199 # miss rate (i.e., misses/ref)
ul2.repl_rate                  0.0198 # replacement rate (i.e., repls/ref)
ul2.wb_rate                    0.0130 # writeback rate (i.e., wrbks/ref)
ul2.inv_rate                   0.0000 # invalidation rate (i.e., invs/ref)
ul2.filtered                        0 # total filtered access
ul2.filtered_read                   0 # total filtered read
ul2.filtered_write                  0 # total filtered write
ul2.buf.access               29843592 # total filter buffer accesses
ul2.buf.read                  9917372 # total amount of time for any cache to read
from the filter buffer
ul2.buf.write                17428895 # total amount of time for any cache to write
to the filter buffer
ul2.buf.mem_write            17428895 # total amount of time the filter buffer
performed a write to memory.
ul2.buf.mem_fetch             2497325 # total amount of time the filter buffer fetch
data from memory.
ul2.buf.merge                       0 # total amount of time for the filter buffer to
perform a write merge
```

```
ul2.buf.wait                    6690054 # total amount of of time that the processor
must wait for a buffer entry to be cleared
ul2.buf.wait_cycle          2811554458 # total amount of cycle spent on waiting for
an empty buffer entry
itlb.accesses             104219153196 # total number of accesses
itlb.hits                 104219153159 # total number of hits
itlb.misses                         37 # total number of misses
itlb.replacements                    0 # total number of replacements
itlb.writebacks                      0 # total number of writebacks
itlb.invalidations                   0 # total number of invalidations
itlb.miss_rate                  0.0000 # miss rate (i.e., misses/ref)
itlb.repl_rate                  0.0000 # replacement rate (i.e., repls/ref)
itlb.wb_rate                    0.0000 # writeback rate (i.e., wrbks/ref)
itlb.inv_rate                   0.0000 # invalidation rate (i.e., invs/ref)
itlb.filtered                        0 # total filtered access
itlb.filtered_read                   0 # total filtered read
itlb.filtered_write                  0 # total filtered write
dtlb.accesses              27694826537 # total number of accesses
dtlb.hits                  27694517323 # total number of hits
dtlb.misses                     309214 # total number of misses
dtlb.replacements               309086 # total number of replacements
dtlb.writebacks                      0 # total number of writebacks
dtlb.invalidations                   0 # total number of invalidations
dtlb.miss_rate                  0.0000 # miss rate (i.e., misses/ref)
dtlb.repl_rate                  0.0000 # replacement rate (i.e., repls/ref)
dtlb.wb_rate                    0.0000 # writeback rate (i.e., wrbks/ref)
dtlb.inv_rate                   0.0000 # invalidation rate (i.e., invs/ref)
dtlb.filtered                        0 # total filtered access
dtlb.filtered_read                   0 # total filtered read
dtlb.filtered_write                  0 # total filtered write
sim_invalid_addrs                    0 # total non-speculative bogus addresses seen
(debug var)
ld_text_base                0x00400000 # program text (code) segment base
ld_text_size                    230448 # program text (code) size in bytes
ld_data_base                0x10000000 # program initialized data segment base
ld_data_size                    351008 # program init'ed `.data' and uninit'ed `.bss'
size in bytes
ld_stack_base               0x7fffc000 # program stack segment base (highest
address in stack)
ld_stack_size                    16384 # program initial stack size
ld_prog_entry              0x00400140 # program entry point (initial PC)
ld_environ_base            0x7fff8000 # program environment base address address
ld_target_big_endian                0 # target executable endian-ness, non-zero if
big endian
mem.page_count                   24867 # total number of pages allocated
mem.page_mem                    99468k # total size of memory pages allocated
mem.ptab_misses                  53882 # total first level page table misses
mem.ptab_accesses         597696304327 # total page table accesses
mem.ptab_miss_rate              0.0000 # first level page table miss rate
mem.access_count              32714504 # Total amount of memory access
mem.unoverlaped_fp_read_clk      173328584 # Total amount of unoverlaped fast
page read delay across the channels in processor clock
mem.unoverlaped_fp_write_clk      362619455 # Total amount of unoverlaped fast
page write delay across the channels in processor clock
mem.unoverlaped_filtered_read_clk               0 # Total amount of unoverlaped
filtered read delay across the channels in processor clock
mem.unoverlaped_filtered_write_clk              0 # Total amount of unvoerlaped
filtered write delay across the channels in processor clock
mem.unoverlaped_random_read_clk       843035395 # Total amount of unoverlaped
random read delay across the channels in processor clock
mem.unoverlaped_random_write_clk       973462985 # Total amount of unoverlaped
random write delay across the channels in processor clock
mem.channel[0].access_count        32714504 # Total amount of memory access to
this channel
mem.channel[0].row_hit        12788495 # Total amount of row hits for this channel
mem.channel[0].fp_reads         5233822 # Total amount of fast page reads to this
channel
mem.channel[0].channel_unoverlaped_fp_read_clk       173328584 # Total amount of
unoverlaped fast page read to this channel in processor clock cycles
mem.channel[0].channel_overlaped_fp_read_clk        77894872 # Total amount of
overlaped fast page read to this channel in processor clock cycles
mem.channel[0].fp_writes          7554673 # Total amount of fast page writes to this
channel
mem.channel[0].channel_unoverlaped_fp_write_clk       362619455 # Total amount
of unoverlaped fast page write to this channel in processor clock cycles
mem.channel[0].channel_overlaped_fp_write_clk            4849 # Total amount of
```

```
overlaped fast page write to this channel in processor clock cycles
mem.channel[0].filtered_reads                  0 # Total amount of filtered reads to this
channel
mem.channel[0].channel_unoverlaped_filtered_read_clk             0 # Total
amount of unoverlaped filtered read to this channel in processor clock cycles
mem.channel[0].channel_overlaped_filtered_read_clk             0 # Total amount
of overlaped filtered read to this channel in processor clock cycles
mem.channel[0].filtered_writes                  0 # Total amount of filtered writes to
this channel
mem.channel[0].channel_unoverlaped_filtered_write_clk             0 # Total
amount of unoverlaped filtered write to this channel in processor clock cycles
mem.channel[0].channel_overlaped_filtered_write_clk             0 # Total
amount of overlaped filtered write to this channel in processor clock cycles
mem.channel[0].random_reads         10051787 # Total amount of filtered reads to this
channel
mem.channel[0].channel_unoverlaped_random_read_clk      843035395 # Total
amount of unoverlaped random read to this channel in processor clock cycles
mem.channel[0].channel_overlaped_random_read_clk      202350453 # Total amount
of overlaped random read to this channel in processor clock cycles
mem.channel[0].random_writes         9874222 # Total amount of random writes to
this channel
mem.channel[0].channel_unoverlaped_random_write_clk      973462985 # Total
amount of unoverlaped random write to this channel in processor clock cycles
mem.channel[0].channel_overlaped_random_write_clk      53456103 # Total
amount of overlaped random write to this channel in processor clock cycles
```

Figure A.1: A gzip sim-outorder simulation output file

A.2 Memory simulation result verification

Referring to the sample simulation output file from appendix A.1 on page 116, the following memory related statistics are found:

Processor clock = 3200MHz

Memory bus speed = 800MHz

Tcas = 7

Trcd = 7

Trp = 7

Tcmd = 1

Tcwd = 7

Total memory accesses = 32714504

Total row hit = 12788495

Total fast page read = 5233822

Total unoverlapped fast page read latency = 173328584

Total overlapped fast page read latency = 77894872

Total fast page write = 7554673

Total unoverlapped fast page write latency = 362619455

Total overlapped fast page write latency = 4849

Total random read = 10051787

Total unoverlapped random read latency = 843035395

Total overlapped random read latency = 202350453

Total random write = 9874222

Total unoverlapped random write latency = 973462985

Total overlapped random write latency = 53456103

First, we must prove that the total amounts of different types of memory accesses are correct. From the statistics, the total amount of memory accesses should equal to the sum of the total amount of fast page read/write and random read/write.

Total amount of memory accesses = 32714504

Total amount of row hits = 12788495

Total amount of fast page accesses

= total amount of fast page read + total amount of fast page write

= 5233822 + 7554673 = 12788495

Total amount of random accesses

= total amount of random read + total amount of random write

= 10051787 + 9874222 = 19926009

Total amount of memory accesses

= Total amount of fast page accesses + total amount of random accesses

= 1288495 + 19926009 = 32714504

From the calculation results, we can see that the total amount of fast page accesses is matching the total amount of row hits, and the total amount of memory accesses calculated from the sum of all fast page and random accesses is correct.

Next, we must check whether the latecies recorded by the simulation file is correct. This is achieved by first calculating the expected fast page read/write latency and random read/write latency:

The processor to memory clock ratio is:

3200(processor clock)/800(memory clock) = 4

The data transmission (data burst) is:

(64B (cache block size) / 8B (data bus width))/2 (DDR) = 4 memory clocks

Random access read/write latency in processor clock cycles:

4(processor to memory clock ratio) X (1(Tcmd) + 7(Trp) + 7(Trcd) + 7(Tcas for read) OR 7(Tcwd for write) + 4(data burst)) = 104 processor clocks

Fast page read/write latency in processor clock cycles:

4(processor to memory clock ratio) X (1(Tcmd) + 7(Tcas for read) OR 7(Tcwd for write) + 4(data burst)) = 48 processor clocks.

From the statistics, the total amount of fast page read is 5233822, the total amount of unoverlapped fast page read latency is 173328584 and the total amount of overlapped fast page read latency is 77894872. By calculating the total amount of fast page read latency and compare it with the overlapped and unoverlapped fast page read latency, we can then prove that the fast page read latency statistical values are correct:

Total fast page read latency:

= fast page read latency x total amount of fast page reads

= 48 x 5233822 = 251223456

Sum of overlapped and unoverlapped fast page read latency:

= 173328584 + 77894872 = 251223456

Since the total fast page read latency is matching the sum of overlapped and unoverlapped fast page read latency, we can conclude that the fast page read statistical values are correct.

We will also perform the same check for fast page write and random read/write:

Total fast page writes = 7554673

Total unoverlapped fast page write latency = 362619455

Total overlapped fast page write latency = 4849

Total fast page write latency:

= fast page write latency x total amount of fast page writes

= 48 x 7554673 = 362624304

Sum of overlapped and unoverlapped fast page write latency:

= 362619455 + 4849 = 362624304 = Total fast page write latency

Total random reads = 10051787

Total unoverlapped random read latency = 843035395

Total overlapped random read latency = 202350453

Total random read latency:

= random read latency x total amount of random reads

= 104 x 10051787 = 1045385848

Sum of unoverlapped and overlapped random read latency:

= 843035395 + 202350453 = 1045385848 = Total random read latency

Total random writes = 9874222

Total unoverlapped random write latency = 973462985

Total overlapped random write latency = 53456103

Total random write latency:

= random write latency x total amount of random writes

= 104 x 9874222 = 1026919088

Sum of unoverlapped and overlapped random write latency:

= 973462985 + 53456103 = 1026919088 = Total random write latency

Since all calculated fast page read/write and random read/write latencies are matching the sum of the overlapped and unoverlapped latencies of the corresponding memory access type, we can now conclude that our custom version of sim-outorder is generating correct statistical values.