The Dissertation Committee for Ciji Isen Certifies that this is the approved version of the following dissertation:

The Use of Memory State Knowledge to Improve Computer Memory System Organization

Committee:

Lizy Kurian John, Supervisor

Kathryn S McKinley

Mattan Erez

Adnan Aziz

Ravi Bhargava

Paul V Gratz

The Use of Memory State Knowledge to Improve Computer Memory System Organization

by

Ciji Isen, B. Tech.; M. Engr

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2011

Dedication

To my wife, Gitanjali Mathur

And my parents, Oommen Isen and Aleyama Isen

Acknowledgements

In culminating my graduate school life, it gives me great pleasure to thank the many people who have made this thesis and choice of life possible for me.

I would like to thank Prof. Lizy John for her guidance, insight and flexibility all these years. She has believed in me from day one and I am grateful for her unconditional support. Her positive attitude and encouragement were instrumental in keeping me motivated. I appreciate her balanced approach to work, study and life; that has made my doctoral study possible.

I am grateful to my committee members, Prof. Mattan Erez, Prof. Kathryn S. McKinley, Prof. Adan Aziz, Dr. Ravi Bhargava and Prof Paul V. Gratz. Their helpful comments and suggestions greatly improved the quality of my dissertation. I have been greatly inspired by Prof. Kathryn S. McKinley's research into memory managed languages which helped me during the early days of my search for a thesis topic. I have had the pleasure of attending one of Prof. Mattan Erez's courses in which he helped me learn a smart and balanced approach to prove research ideas and the art of thought experiments. Dr. Ravi Bhargava, with whom I have had the pleasure of working for over two summer internships, has been a mentor and a guide. I am grateful for his insight into the post-academic world which helped me stay grounded and excited.

Thanks to Amy Levin, Melanie Gulick, and other administrative staff for their hard work. In addition, I gratefully acknowledge the IBM Corporation for their generous financial support through the PhD fellowship program. I would also like thank Ken Vu and Alex Mericas for helping my candidacy for the PhD fellowship.

I thank all my friends both at UT and outside for making my graduate student life in Austin thoroughly enjoyable. In particular, I would like to thank Dr. Anil Kottam, my dear friend and roommate of several years, for introducing me to UT Austin and making my life in Austin a joy. I am thankful to my fellow researchers in the Laboratory for Computer Architecture (LCA) for our collaborations, discussions and feedback over the years. I would like to thank Dr. Dimitris Kaseridis for his friendship and help all through my PhD. The many great discussions and rants with him on research, life and everything under the sun are a memorable part of my graduate student life. I am grateful to the other members of LCA, Dr. Lloyd Bircher, Dr. Jian Chen, Arun Nair, Dr. Jeff Stuecheli, Karthik Ganesan, Faisal Iqbal, Umar Farooq, YoungTaek Kim, and Jungho Jo for their support and constructive feedback over the years which has helped my research and dissertation.

Finally, I would like to thank my beloved wife Gitanjali Mathur and my parents Mr. Oommen Isen and Mrs. Aleyama Isen. Doctoral study requires a major commitment from family. Doctoral study while getting married and then starting a job requires an incredible commitment and sacrifice from the spouse. I would like to thank my wife, Gitanjali, for her unconditional love and support through academic and personal ups and downs. I would also like to thank my brother Tiji Isen for being such a wonderful and mature friend. I am eternally grateful for my parents unwavering encouragement, support and patience without which this dissertation would not have been possible. Your love and prayers are the rock of my life.

The Use of Memory State Knowledge to Improve Computer Memory System Organization

Publication No._____

Ciji Isen, Ph.D.

The University of Texas at Austin, 2011

Supervisor: Lizy Kurian John

The trends in virtualization as well as multi-core, multiprocessor environments have translated to a massive increase in the amount of main memory each individual system needs to be fitted with, so as to effectively utilize this growing compute capacity. The increasing demand on main memory implies that the main memory devices and their issues are as important a part of system design as the central processors. The primary issues of modern memory are power, energy, and scaling of capacity. Nearly a third of the system power and energy can be from the memory subsystem. At the same time, modern main memory devices are limited by technology in their future ability to scale and keep pace with the modern program demands thereby requiring exploration of alternatives to main memory storage technology. This dissertation exploits dynamic knowledge of memory state and memory data value to improve memory performance and reduce memory energy consumption.

A cross-boundary approach to communicate information about dynamic memory management state (allocated and deallocated memory) between software and hardware

memory subsystem through a combination of ISA support and hardware structures is proposed in this research. These mechanisms help identify memory operations to regions of memory that have no impact on the correct execution of the program because they were either freshly allocated or deallocated. This inference about the impact stems from the fact that, data in memory regions that have been deallocated are no longer useful to the actual program code and data present in freshly allocated memory is also not useful to the program because the dynamic memory has not been defined by the program. By being cognizant of this, such memory operations are avoided thereby saving energy and improving the usefulness of the main memory. Furthermore, when stores write zeros to memory, the number of stores to the memory is reduced in this research by capturing it as compressed information which is stored along with memory management state information.

Using the methods outlined above, this dissertation harnesses memory management state and data value information to achieve significant savings in energy consumption while extending the endurance limit of memory technologies.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Memory subsystems are power and performance bottlenecks in computer systems. It is important that the memory system caters to modern applications by supporting their growing working set, implying that the memory capacity has to grow along with the working set size to keep up with the applications. Virtualization has become extremely common, primarily with the advent of modern data centers, for reasons of operating efficiency of server infrastructure for cloud and enterprise computing. Therefore, the pressures and the demands on memory are on the rise. The growing use of virtualization coupled with the deployment of multicore processor and multiprocessor environments requires a massive increase in the amount of main memory needed in each individual system so that the growing computing facility may be effectively utilized. Hence, the issues with main memory devices are extremely important problems.

1.1 Dynamic Random Access Memory (DRAM) issues

Dynamic Random Access Memory (DRAM) technology is the corner stone of main memory in computer systems. Main memory built from DRAM technology faces severe limitations in terms of power, scaling, and cost [L03]. According to observations made at a recent International Solid-State Circuits Conference (ISSCC), Sun Microsystems revealed that the power consumption of DRAM in the UltraSPARC T1 ("Niagara") systems running SPECjbb was approximately 60 watts [L06], which amounts to almost as much as the processor core power. Another study by Lefurgy et al. [L03] indicates that about 40% of the system power in a Power 4 based system (IBM p670) comes from the memory subsystem (Table 1.1), which includes the DRAM, the memory controllers, buses, etc. Measurements done by Samsung via sensors installed on several Fujitsu PRIMERGY systems also bills the DRAM memory at 33% (Figure 1.1) factor in

the system power. These observations have shed light on the power and energy consumption of memory, and have made it a first-class design consideration for modern systems.

IBM p670 server	Processor	Memory	I/O and other	Processor and memory fans	I/O fans	Total watts
4 way Power 4	384	318	90	676	144	1,614
16 way Power 4	840	1,223	90	676	144	2,972

Table 1.1 Power consumption breakdown for an IBM p670 in watts [L03]

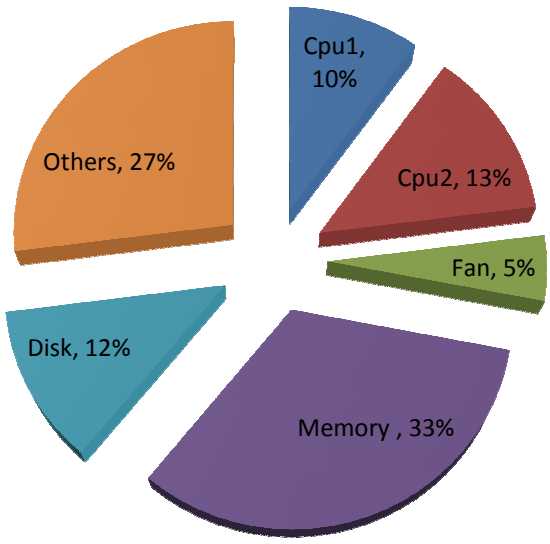


Figure 1.1 Power composition of system components – measurement done via sensors installed on all major hardware components of a Fujitsu PRIMERGY systems using Fujitsu's ServerView application. [BAJR10]

1.2 ISSUES WITH EMERGING MEMORY TECHNOLOGIES

In DRAM technology, DRAM places charge in a storage capacitor and must mitigate sub-threshold charge leakage through the access device. DRAM technology requires capacitors large enough to store charge for reliable sensing and transistors that are large enough to have effective control over the channel – a formidable challenge in manufacturing. In light of these challenges, viable solutions to manufacture DRAM with scaling beyond 36 nm are unknown [ITRS09]. Recent academic and industry research is exploring emerging memory technologies such as Phase-Change Memory (PCM) as alternatives to the main memory. The storage material in a PCM cell can be in more than one degree of partial crystallization allowing for more than one bit to be stored per cell. The different crystalline states and their associated difference in resistance can be exploited to store multiple bits. For example, a recently constructed prototype [B08] stores four states, with two bits stored in the same physical area. However, PCM too has its drawbacks. Currently, PCM writes require energy intensive current injection. More importantly, these writes induce thermal expansion and contraction within the storage element which degrades the injection contacts and limits the endurance of the PCM device to hundreds of millions of writes per cell [ITRS09, FW08]. PCM based memory also suffers from a higher latency compared to DRAMs. The endurance limits as well as the higher latency of PCM relative to DRAM are challenges that need to be addressed before the positive attributes of PCM, such as scaling and density, can be exploited.

Table 1.2 contains a summary of the various memory technologies which are candidates for main memory, i.e., hierarchy level before the disk drive. PCM is a relatively dense technology, 2X-4X denser than DRAMs and yet they have comparable feature sizes. The term “Write Endurance” refers to the maximum number of writes for

each cell while "Data retention" refers to the duration the storage technology can retain data.

Parameter	PCM	NOR Flash	NAND Flash	DRAM
Density	2X-4X	0.25X	4X	1X
Write Endurance	10^6 to 10^8	10^4	10^4	10^{16}
Read Latency	200-300 ns	300 ns	25 us	60 ns
Write Speed	≈ 100 MB/s	0.5 MB/s	2.4 MB/s	≈ 1Gbps
Retention	10 yrs	10 yrs	10 yrs	Refresh

Table 1.2 Comparison of memory technologies [QSR09]

Among flash technologies, NOR based flash shows itself to be an ill fit considering the low density it supports. NAND based flash has a very high density, even higher than DRAM but has high access latency. In addition to the difference in the latency, flash memory has very low write endurance making it a very poor fit for main memory. The reasons for PCM coming up as a viable option is evident from its characteristics which are similar to the NAND flash but with a better read latency and write speed. PCM's write endurance (10^6 to 10^8) is a few orders higher than Flash's (10^4). The drawback of Flash is that it is 200X slower than DRAM and has a limited endurance for the number of writes [ITRS09] it can sustain, making it unsuitable for the main memory. PCM fares much better since it is only 2X – 4X slower than DRAM and can provide up to 4Xx more density than DRAM. Phase change memory (PCM) has the appeal of being a non-volatile storage mechanism amenable to process scaling. Based on

these characteristics PCM and DRAM are the primary technology candidates for main memory.

All these point to the problems with the memory systems of today and the need to focusing on aggressive mechanisms to optimize them.

1.3 MEMORY OCCUPANCY AND STATE OF COMPUTER SYSTEMS

Memory systems treat all regions of memory as important and go to great lengths to conserve their fidelity. In programming languages ranging from C, C++, C#, Java, Python to Ruby, mechanisms exist for explicit or implicit dynamic allocation of memory. When a user's code invokes memory allocation and allocates a region of memory whose existing content is uninitialized, the computer system is normally ignorant of the fact that the current value present in the physical memory location is unused during the correct execution of the program, i.e., it is *inconsequential data*. Similarly, when a program is done with its use of the dynamically allocated memory, it is returned to the memory manager routine which now knows that the program is done using that region of memory. Thus, the memory block that was given up by the program too contains data that is normally considered important by the computer, which thus maintains its fidelity. In reality it is never used during the correct execution of the program i.e., that data too is inconsequential. This dissertation calls regions of the memory that contain data that is unused by the correct execution path of the program *inconsequential memory*.

This dissertation characterizes and exploits inconsequential memory information to optimize memory systems. Most current computer systems operate agnostic of semantic information about memory state and values which are present in the program. They fail to exploit knowledge about inconsequential memory and hence, do not optimize the memory hierarchy to enhance energy, reliability, and performance characteristics of

the memory hierarchy. Although there have been memory hierarchy optimizations such as cache-locking, cache-bypass and prefetching, most of the modern optimizations done in the microarchitecture and memory subsystem tend to be agnostic of the program semantic based memory state. For example, when a program or operating system memory manager allocates or frees up a memory region, the system does not have this information and does not try to act on it.

1.3.1 Key Ideas using Memory State Information

Regions of memory that the program deallocates (gives back to the memory manager) and hence is inconsequential, can still be present in the memory hierarchy. In most cases, such memory will contain modified data that is useless to the program. Usually, when a part of the memory hierarchy containing inconsequential and modified data is evicted, the data is written back to the next level of the hierarchy. However, in the case of inconsequential data this write back is unnecessary. Similarly, when a store operation is issued on a freshly allocated memory region not present in the memory hierarchy, it normally results in a write miss which results in a fetch operation from the next level of the memory. However, write miss operations to inconsequential data are unnecessary and can be avoided. This dissertation proposes to avoid write back and write miss accesses to inconsequential memory. Avoiding write back and write miss access helps in reducing access traffic to the DRAM, which in turn helps in reducing the power and energy consumption of the DRAM based memory. Reducing write backs also helps in reducing the number of write operations that go out to the main memory, which helps in extending the life of PCM (and other EMT) based memory.

Furthermore, when stores write zeros to memory, the number of stores to the memory can be reduced by capturing it as compressed information. This dissertation

proposes to compress and bypass zero value stores to the main memory and thereby reduce the number of write operations that go out to the main memory. This, in turn, extends the life of the EMT based memory.

This dissertation harnesses and exploits memory management state and data value information from the semantics of the program to optimize the main memory system.

1.4 THESIS STATEMENT

By harvesting dynamic memory management state visible to the user code and dynamic memory data value visible to the hardware, we can significantly improve energy consumption and endurance of memory systems.

1.5 CONTRIBUTIONS

This dissertation makes several key contributions:

1) This dissertation evaluates the dynamic memory management based state for all the benchmarks in SPEC CPU2006. Further, this dissertation analyzes the relationship of the benchmark behavior to the behavior of the last level cache and memory level cache.
2) This dissertation presents the concept of inconsequential memory and the use of inconsequential memory to reduce write backs and write misses.
3) This dissertation also presents the concept of zero value stores and ways to reduce data store operations to the memory.
4) This dissertation presents ESKIMO, an architectural mechanism to exploit inconsequential memory based on their memory management state. ESKIMO optimizes the DRAM based memory to avoid write back and write miss accesses as well as refresh operations related to inconsequential memory regions. This dissertation develops and discusses in detail the necessary

hardware structures, ISA changes, their operations and interactions. Further, this dissertation presents a complete evaluation of the efficacy of the DRAM energy optimizations and finds that on an average (volume weighted) 10% of memory subsystem energy is saved.

5) This dissertation presents mFilter, an architectural mechanism to avoid write backs of inconsequential memory and zero value data to reduce store operations to the emerging memory technology based memory. The dissertation develops and presents the necessary hardware structures (mFilter), ISA changes, their operations and interactions. Further, the dissertation presents a complete evaluation of the efficacy of these proposals and their effect on the endurance of such a memory device and finds that on average (volume weighted), 11% of the destructive memory access can be avoided using an mFilter.

1.6 ORGANIZATION

This dissertation is organized as follows:

Chapter 2 provides an overview of the main memory technologies and their problems. Chapter 2 also provides a background into their working and the reason for some of the issues.

Chapter 3 presents a discussion on research work relevant to this dissertation. Various techniques and ideas that are related to this dissertation are discussed in this chapter.

Chapter 4 discusses the benchmarks used in this dissertation as well as details of the simulation methodology used.

Chapter 5 describes the main ideas of inconsequential memory zero value stores presented and used in this dissertation. It details those ideas and their uses.

Chapter 6 presents some preliminary workload analysis to understand how these benchmarks perform in relation to the events targeted.

Chapter 7 describes the application and implementation of these ideas for DRAM memory optimization particularly to reduce its energy consumption. An analysis of the results obtained is also presented.

Chapter 8 describes the application and implementation of these ideas for emerging memory technologies particularly to improve its endurance and lifetime. An analysis of the results obtained is also presented.

Finally Chapter 9 presents conclusions, observations and insights and some future avenues for application of the insights gained.

Chapter 2: Overview of Memory Technologies

2.1 DRAM MEMORY

To better understand efficiencies, strengths and weakness of the DRAM based memory systems, this chapter describes basic circuits and architecture of DRAM devices. This chapter will focus on the high level details and other details that are important to the dissertation.

The organization and structure of a Fast Page Mode (FPM) DRAM device is illustrated in Figure 2.1. Internally, the array of DRAM storage cells in Figure 2.1 is organized as 4096 rows, 1024 columns per row, and 16 bits of data per column. Each time a row is accessed in these devices, the external memory controller supplies a 12 bit address on the address bus and the row address strobe (RAS) is asserted. The 12 bit address is buffered by the row address buffer inside the DRAM device and then sent to the row decoder. The row address decoder in turn uses the 12 bit address and selects one of the 4096 rows of storage cells.

The data stored in the selected row of storage cells is then sensed and cached by the array of sense amplifiers. In the DRAM cells illustrated, each row consists of 1024 columns and each column is 16 bits wide. The 16 bit wide column is the basic addressable unit of the memory in this device. Column accesses that follow the row access would normally read or write 16 bits of data from the same row of DRAM. The FPM DRAM device is structured to allow each 8 bit half of the 16 bit column to be accessed independently through the use of separate column access strobe high (CASH) and column access strobe low (CASL) signals. A column access is engaged similar to the row access. The main memory controller places a 10 bit address on the address bus and

then asserts the appropriate column access strobe (CAS#) signals. Inside the DRAM chip the 10 bit column address is decoded and used to select one of the columns out of 1024 columns. The data present in that column is then placed on the data bus or is overwritten with data from the data bus depending on the status of the write enable (WE) signal.

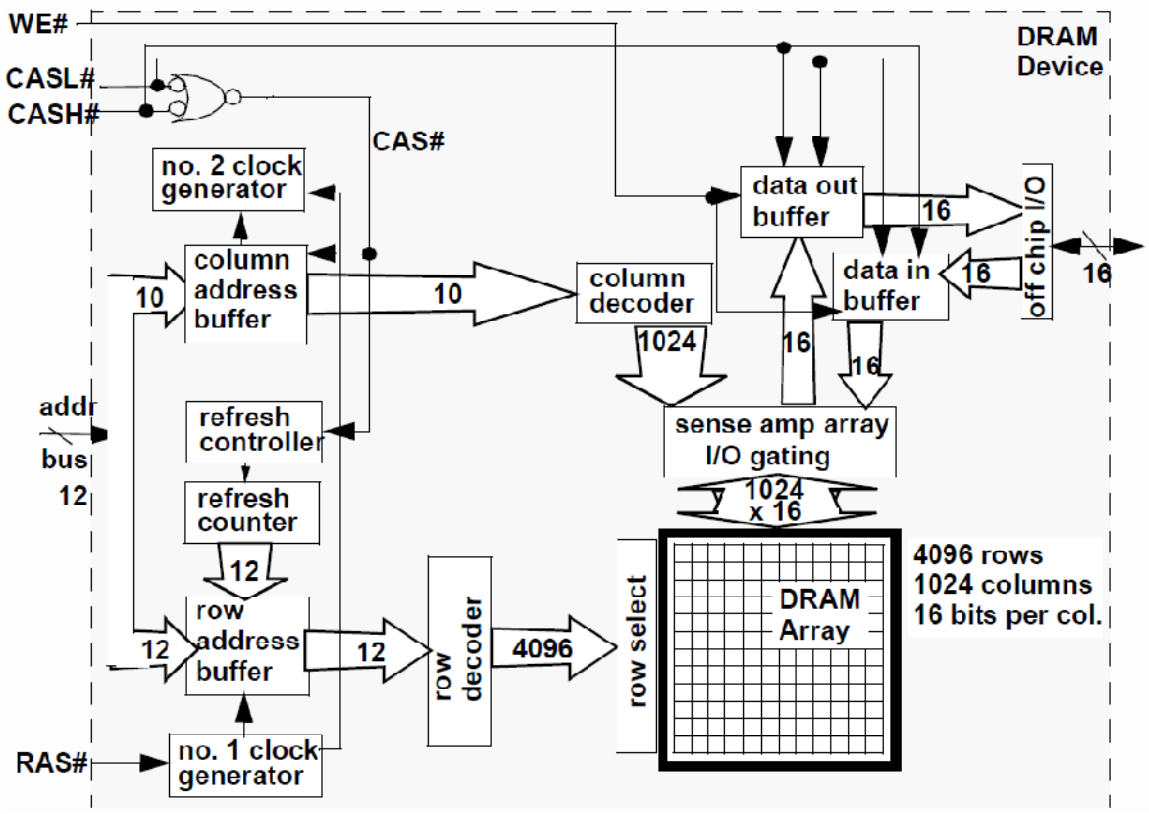


Figure 2.1 64 Mbit fast page mode DRAM device (4096 x 1024 x 16) (borrowed figure) [W05]

All DRAM devices, from the FPM DRAM device to modern DDRx (denotes DDR, DDR2, DDR3) SDRAM devices, have similar basic organizations. The DRAM devices have one or more arrays of DRAM cells which are organized into a mesh of a certain number of rows and columns. The column is the smallest unit of addressable memory on that device. The timing and sequence of how the device operates on all DRAM devices is controlled by logic circuits. For example, in the Figure 2.1 illustrating

an FPM DRAM device, the chip has an internal clock generator as well as a built-in refresh controller. In most cases the DRAM device along with the memory controller controls the relative timing and sequence of events for a particular action. The FPM DRAM also stores the address of the next row that needs to be refreshed. When the memory controller asserts a new refresh command to the DRAM device, the row address can be loaded internally from the refresh counter rather than having to load the row address from an off chip address bus. Historic limitations restrict the number of pins used in DRAM devices. Due to the limitation on pin count DDRx and variants of future DDRx SDRAM devices modern DRAM devices move data onto and off the device through a set of bi-directional input-output pins connected to the system. There are more advanced DRAM devices such as ESDRAM, Direct RDRAM and RLDRAM which have evolved to include more logic circuitry and functionality such as row caches and write buffers to permit read-around-write functionality. The additional integrated logic helps to improve the performance at the expense of die area in the DRAM device. However, due to the additional hardware cost standard DRAM devices do not integrate such logic.

2.1.1 DRAM Storage Cells

The circuit diagram of the basic one transistor, one capacitor (1T1C) cell structure used in modern DRAM devices as the basic storage unit is illustrated in Figure 2.2. In the circuit structure in Figure 2.2, applying a voltage on the gate of the access transistor turns it on, causing a voltage representing the data value to be placed onto the bit line. The voltage on the bit line charges the storage capacitor. The capacitor retains the stored charge for a limited period of time after the voltage on the word line has been removed and the access transistor is turned off. Due to leakage of currents through the access transistor, the charge stored in the capacitor gradually dissipates. As a result, the data

stored in the DRAM cells must be periodically read-out and written back before the stored charge decays to an indistinguishable value; this process is known as refresh. Failing to do a refresh will cause the electrical charge to leak away and cause the stored value to be unresolvable.

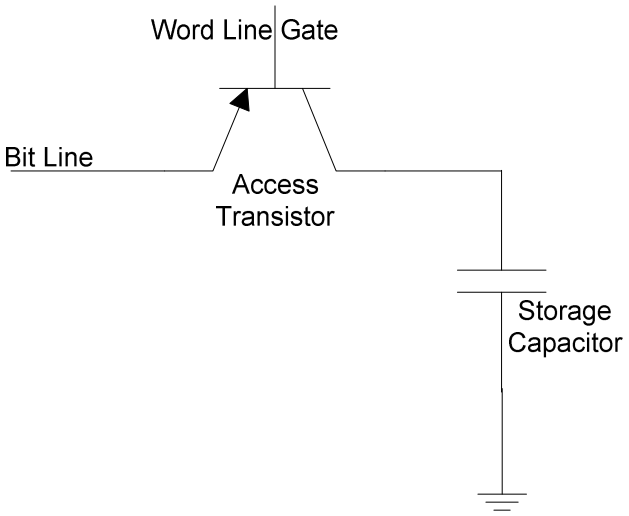


Figure 2.2 Basic 1T1C DRAM cell structure

2.1.1.1 Cell Capacitance, Leakage and Refresh

The capacitance of a typical DRAM storage cell built in the 90 nm process technology is on the order of 30 fF and the leakage current of the DRAM access transistor is on the order of 1 fA [LAPKLLCK03]. With this level of leakage current and capacitance, a typical DRAM cell can retain the state for hundreds of milliseconds. In other words, hundreds of milliseconds after the data write, the differential sense amplifier will resolve the charge stored in the DRAM cell to the stored digital value. All DRAM cells are not built the same and hence some of them can hold the stored charge for much longer, in the order of several seconds. Since memory systems have to be reliable and not loose a single bit, every single DRAM cell must be refreshed at least once before any single bit in the whole device loses its stored charge. Most modern DRAM memory

systems typically refresh the storage cells once every 32 ms or 64 ms or 110 ns depending on the DRAM technology. In cases where the DRAM cells have low capacitance storage capacitors or high leakage currents through the access transistor, the refresh interval (time period between refresh) must be reduced so as to ensure reliable data storage.

2.1.2 DRAM Power and Energy Modes

Figure 2.3 illustrates the different power and energy modes and their transitions in a DRAM.

Active: An active state is where the DRAM stores its data in the sense-amplifier. In the active state the DRAM module is ready to accept row and column address packets. Upon accepting such a packet the module transitions into a read/write state. A read/write state has two components to it; the selection and activation of the correct bank for read/write and the actual read/write of the target row. Both the row and column multiplexer receivers are turned to active in order to enable the arrival of address packets (row or column). The energy consumption of this mode is the highest.

Standby: In the standby mode, the column multiplexers are disabled resulting in significant energy savings compared to the active state. The memory banks are idle with the memory clock set to high. A read or write request causes an ACTIVATE command to be issued to the DRAM module which in turn causes it to transition to the active state.

Power-down: Power-down mode is one more step in energy savings. The DRAM device in this mode shuts down the periodic clock and the synchronization circuitry. When it receives an ACTIVATE command in this state, the DRAM will transition to the active state. Power-down mode is costlier and typically costs several thousand cycles.

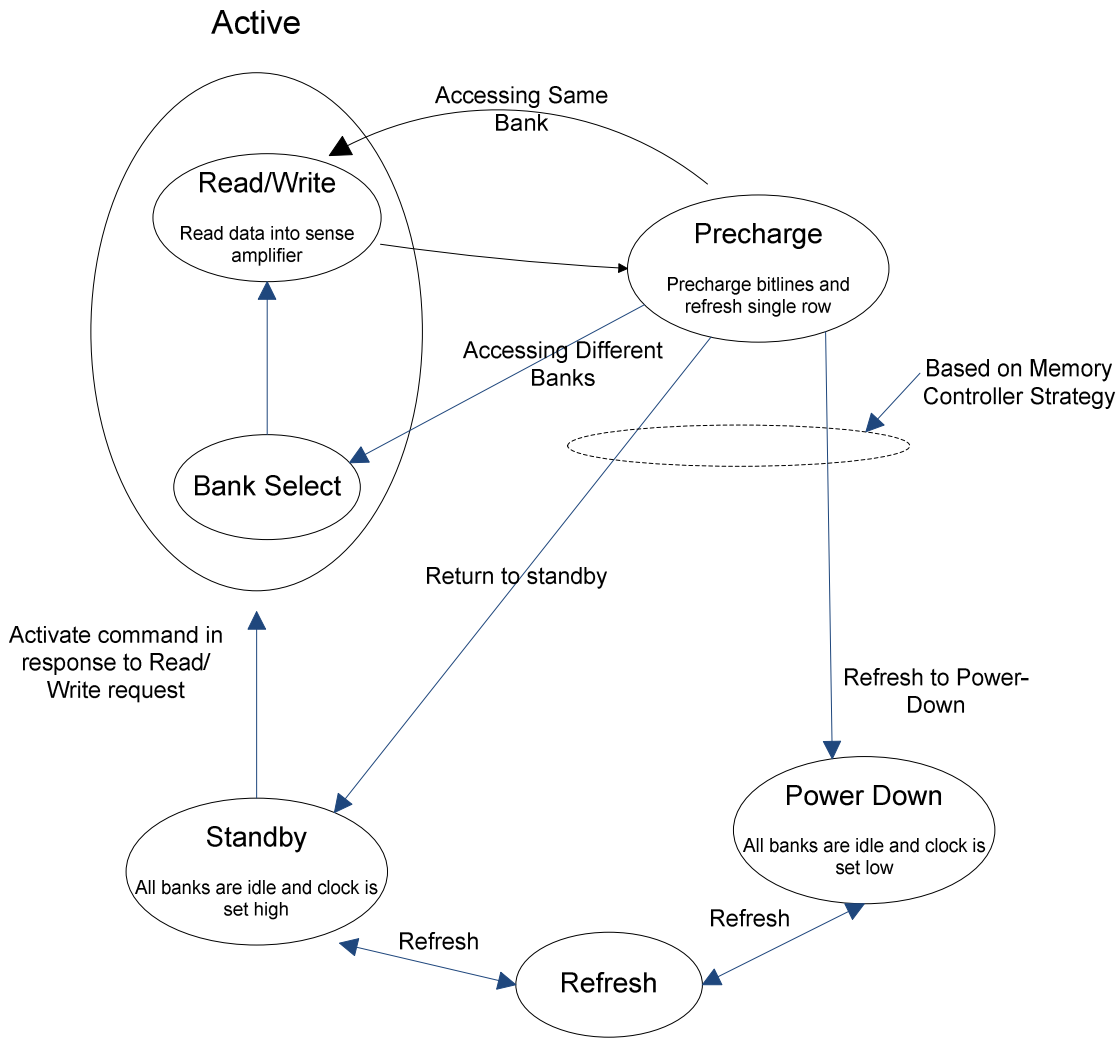


Figure 2.3 State transition diagram in DRAM

Precharge: Accessing the DRAM via the sense-amplifier is a destructive operation. Once the charge corresponding to the data has been stored in the sense-amplifier, the charge is lost from the memory cell and cannot be reconstructed. Hence, the cell contents need to be reconstructed based on the content of the sense amplifiers and is reconstructed using a precharge operation. After every read or write operation the memory controller sets up a precharge operation. After a precharge operation the memory

controller can switch to a read/write state if the same bank is being accessed. If a different bank contains the data then that bank needs to be selected for read/write. If there are no more pending read or write operations following the precharge, the DRAM may be put into standby, power-down or active states. The choice of power saving mode is determined by the policy in the memory controller.

Refresh: Since the memory cells are based on a capacitor, it loses charge over time due to leakage effects. The charges on these cells need to be restored in a periodic fashion. These intervals can range from 64 ms (DDR2) to 110 ns (DDR3) depending on the technology. The operation to restore the data is called refresh operation. Thus, the memory controller has to issue REFRESH commands to the DRAM periodically.

2.1.3 DRAM Refresh Techniques

Due to the dynamic nature of a DRAM cell, periodic refresh operations are required for keeping the data stored. Even in standby mode, such regular refreshes account for large energy consumption in DRAMs. Some studies have shown that even in the lowest power mode, the power needed to keep the DRAM contents is about a third of the total power dissipated. Factors such as the memory vendor and the design technology affect the refresh rate; a typical refresh interval would be 64 ms. This means, a refresh operation takes place every 64 ms. A refresh operation fundamentally involves reading the DRAM cell out and writing back to the same cell. Although a refresh operation consumes power and bandwidth, it is inevitable for the sake of data correctness.

There are two commonly used refresh modes in commercial DRAM designs:

Burst Refresh: In this mode, the entire refresh operation, for all the rows is done one after the other in a burst. The drawback of this scheme is that it increases the peak power consumption of the DRAM. Additionally, it can cause potential performance

degradation during the time of the refresh operations since the DRAM memory module is unable to handle normal access requests.

Distributed Refresh: In distributed refresh mode, refresh operations are spread evenly throughout the refresh interval. The distributed refresh mode has several advantages over burst mode. If the memory controller ensures that a large number of refresh operations are done while the DRAM is idle, then the performance impact will be minimized. Since the refresh cycles are not adjacent to each other it reduces the peak power when compared to the burst mode.

DRAM	Refresh Time	Number of Cycles	Refresh Rate
4 Meg x 1	16 ms	1,024	15.6 µs
256K x 16	8 ms	512	15.6 µs
256K x 16	64 ms	512	125 µs
4 Meg x 4 (2K)	32 ms	2,046	15.6 µs
4 Meg x 4 (4 KB)	64 ms	4,096	15.6 µs

Table 2.1 Refresh specifications for standard DRAMs [MICRON]

2.2 PHASE CHANGE MEMORY

The technology that forms the underpinnings of Phase Change Memory has its roots in research by Ovshinsky [O68] on the properties of a class of amorphous materials. Amorphous materials do not have a definite ordered crystalline structure. It was found, in 1968, that these glasses exhibited a reversible change in resistivity upon changes in phase. Several decades later companies such as Intel, Micron and STMicroelectronics spurred the modern resurgence of this technology.

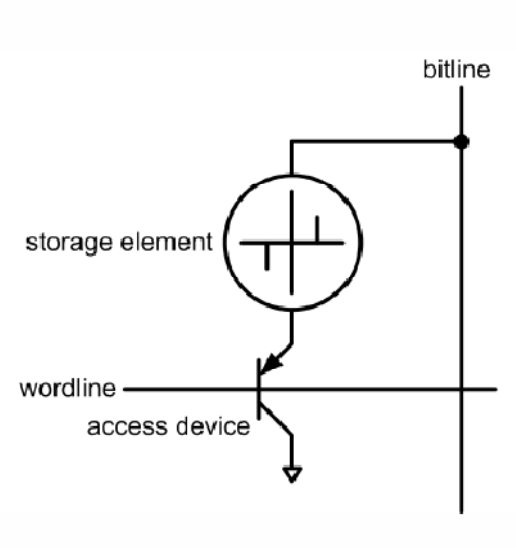


Figure 2.4 Cell structure circuit diagram (borrowed figure) [LIMB09]

2.2.1 The Theory of Operation

Two electrodes separated by a resistor and a phase change material (typically a chalcogenide) are the basic storage elements in the memory. Ge2Sb2Te5 (GST) is the most commonly used chalcogenide. The phase change is induced in the material by injecting current into the base storage element, the resistor-phase change material junction, and heating the chalcogenide to 650 C. In the amorphous phase, the material is highly disordered (absence of regular order to the crystalline lattice). The material has high resistivity and reflectivity in this state. On the other hand, in the polycrystalline phase, the regular crystalline structure exhibits low reflectivity and low resistivity. The programming complexity and latency is lowered by the fact that the current and voltage characteristics of the chalcogenide are identical (regardless of the initial phase) [LIMB09]. The programmed state in the device is controlled by the width and amplitude of the current pulse injected. Phase change memory devices are comprised of an access transistor and the resistive storage element (Figure 2.4 and Figure 2.5). A Field-effect

transistor (FET), bipolar junction transistor (BJT) or a diode is used to control access to the device.

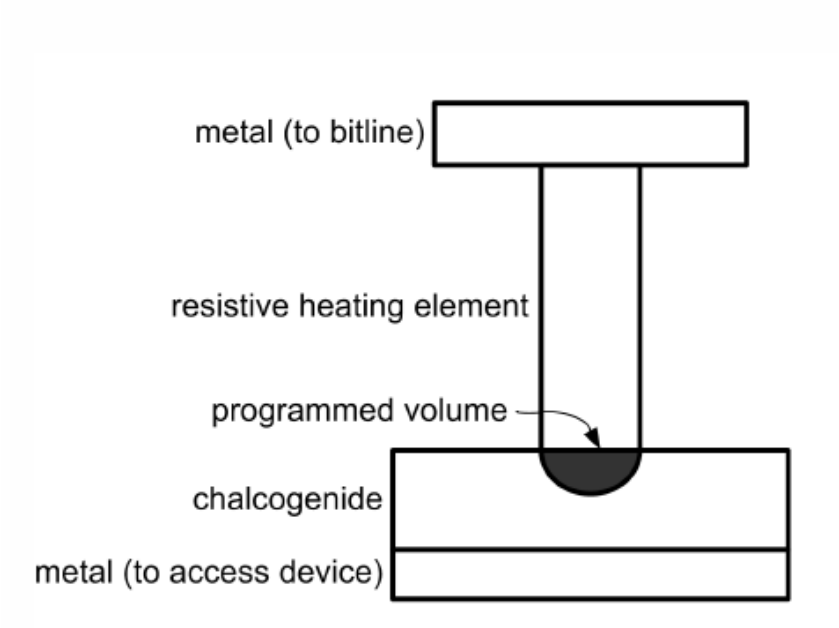


Figure 2.5 Phase change memory: storage element with heating resistor and chalcogenide material between electrodes (borrowed figure) [LIMB09]

2.2.2 Writes

The PCM memory has two states of operations: The SET state and the RESET state. The SET and RESET states refer to the crystalline (low resistance) and amorphous (high-resistance) states of the chalcogenide. The storage element is RESET by a high, short current pulse. The abrupt discontinuation of the current flow due to the short pulse squashes the heat generation and freezes the chalcogenide into the amorphous state it was converted to due to the pulse. The storage element is SET by using a long current pulse of moderate value. The current pulse ramps down over the duration of the write causing gradual cooling of the chalcogenide thereby inducing crystal growth and conversation into the crystalline state.

Since the SET operation is the longer of the two, it determines the write performance. Works by Ahn et al. and Bedeschi et al. [A04, B04] suggests a write latency of 150 ns. Values from this work have been extrapolated by Lee et al. [LIMB09] to arrive at a SET current and voltage of 150 μA and 1.2 V. This implies that SET dissipates 90 μW for 150 ns, consuming approximately 13.5 pJ. The RESET latency is dependent on the write energy. Lee et al. [LIMB09] derived the RESET latency of 40 ns based on the work of Bedeschi et al. [B04] and also suggests that the REST operation requires 300 μA at 1.6V and dissipates 480 μW for 40 ns and consumes approximately 19.2 pJ. Lee at al [LIMB09] arrived at this by extrapolating across process generations using current scaling rules. Shorter SET latencies in the range of 80 ns to 100 ns are being demonstrated by emerging technologies in this field. The longer SET latencies are in the range of 180 ns to 400 ns and it occurs because of the choice of dense devices that are slow in access [K06, L08, O05]. Most PCM prototypes consider the storage elements as a two state devices (i.e. crystalline and amorphous) and hence produce single-level cells (SLC). However recent research has demonstrated additional intermediate states [B08, N08] which can be used to device multi-level cells (MLC). Such multi-level cells store multiple bits by programming the cell to transition into intermediate resistance levels. Essentially a smaller current slope (i.e. slow ramp) produces lower resistance states while larger slopes (i.e. fast ramp down) result in higher resistance states. Thus by varying the slopes, partial phase transitions in size and shape of the amorphous material is caused in the junction area which results in the resistance grades ranging from that of the fully crystalline to fully amorphous chalcogenide. MLC is still a hard problem due to the difficulty in differentiating between a large numbers of resistances.

2.2.3 Write Endurance

The operation of writing to the storage device causes the device to wear and hence is a limiting factor in the endurance of the phase change memory. The injection of current into the storage device causes thermal expansion and contraction which causes the electrode-storage contact to degrade. A degraded contact prevents reliable delivery of current pulses in future which limit the ability to program the storage cell. Variation in the current delivered by the electrodes cause variability in resistance too, which in turn reduces the window, the difference between the minimum and maximum resistance states that can be programmed using the current pulse, to also degrade. Due to this, the number of writes that can be performed before the cells stop being capable of being programmed reliably ranges from 10^4 to 10^9. The write endurance depends on the techniques used for manufacturing. The ITRS roadmap projects a higher endurance of 10^9 writes at 65 nm [ITRS09].

2.2.4 Reads

For the read operation the bit line is precharged to the read voltage before reading the cell. If the selected cell is in a crystalline state, the bit line discharges and the current flows via the storage element. If the cell is in an amorphous state it prevents or limits the bit line current. The work by Bedeschi et al. [B04] suggests a cell read latency of 48 ns. The bit line precharge while using a BJT for access control and current sensing is used to arrive at this latency. Based on the same estimation, the cell requires 40 μA of read current at 1.0 V and it dissipates 40 μW for 48 ns resulting in 2 pJ of energy being consumed. Other research works and prototypes, using FET and diode access devices, have demonstrated read latencies which are higher, in the range of 55 ns to 70 ns.

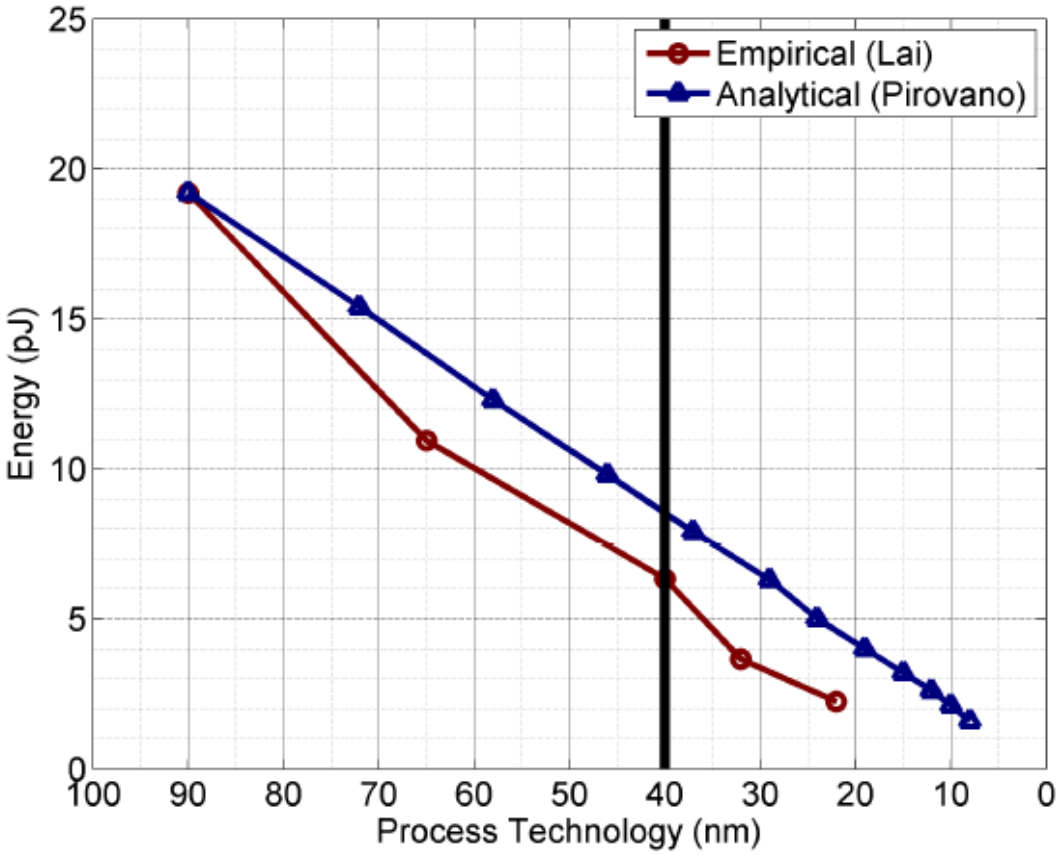


Figure 2.6 PCM RESET energy scaling (borrowed figure) [LIMB09]

2.2.5 Process Scaling

Scaling in PCM reduces the programming current required to be injected via the electrode-storage contact. As scaling reduces the contact area (due to feature size), the thermal resistivity increases. The increase in thermal resistivity means that the volume of phase change material that must be altered to block current flow decreases. This aspect enables the access devices for current injection to be smaller. Figure 2.6 outlines the PCM scaling rules of Pirovano et al. [P03] which was confirmed empirically by Lai [Lai03] in the survey. A reduction in feature size by *k* results in a corresponding

reduction in contact area quadratically ($1/k^2$). The reduced contact area increases the resistivity linearly (k), which in turn causes a linear ($1/k$) reduction in the programming current. These projections are based on the assumption that the SET/RESET voltage does not scale [P03]. At the same time the SET and RESET currents scale together. The SET current is typically 40-80 percent of the RESET current. Both the read and write latencies are not impacted by process scaling. The material used to build the phase change memory determines the write latency.

There are challenges that come with process scaling for PCM technology. The lateral thermal coupling for the decreased contact area, due to scaling, may cause the programming current pulse in one cell to affect the adjacent cell and its state. In the survey done by Lai [Lai03] these effects are shown to be negligible in measurement and simulation. The impact of thermal coupling should be minimal since temperatures fall exponentially with the distance from cell. Another aspect to consider is the effect of increasing resistivity due to the smaller contact area reducing the signal strength because of the smaller difference in resistivity between the crystalline and amorphous states. However, these signal strengths are within the limits of modern memory sense circuits [Lai03].

2.3 OTHER EMERGING MEMORY TECHNOLOGIES

There are several other memory technologies that have also emerged as candidates to replace or work with DRAM based memory. In addition to PCM, Several recent academic works as well as industry research are exploring emerging memory technologies such as MRAM (Magnetic RAM), FeRAM (Ferroelectric RAM) and Flash [ZYZ09, LIMB09, Q09, QSR09].

Flash memory is already being employed as a Solid State Disk (SSD) cache [KRM08] (e.g. Intel® Turbo Memory). The drawback of Flash is that it has very limited endurance for the number of writes [ITRS09] it can sustain (only 10^5) and also has low density at its current and projected nodes relative to other technologies. The limited endurance makes it unsuitable for the main memory. Table 2.2 summarizes the various emerging memory technologies. Most of the emerging memory technologies have the appeal of being a non-volatile storage mechanism (data retention >10 years).

	DRAM	**PCM**	**MRAM**	**FeRAM**	**Flash**
Read Latency vs. DRAM	1x	4x	2x	3x	4x
Write Endurance	-	$>10^8$	$>10^{16}$	$>10^{14}$	10^5
Write Energy (J/bit)	$5x10^{-15}$	$6x10^{-12}$	$1.5x10^{-10}$	$3x10^{-14}$	$1x10^{-14}$
Retention time	64 ms	>10 years	>10 years	>10 years	>10 years
Process Scaling	36 nm	8 nm	65 nm	65 nm	25 nm
Current Process Node	45 nm	45 nm	130 nm	180 nm	90 nm

Table 2.2 Summary of different main memory technologies [ITRS09]

However, these memories have their drawbacks too. For example in MRAM, write operations require very high current [ITRS09, BZE10, Z09], about 5 order higher than DRAM writes, making write operations very expensive in terms of power. FeRAM, PCRAM and other memories also suffer from endurance limits as can be seen in Table 2.2. These emerging memories also suffer from a higher latency compared to DRAMs. The endurance limits, the write cost, as well as the higher latency relative to DRAM are challenges that need to be addressed before the positive attributes of emerging memory technology such as scaling and density can be exploited.

New applications, languages, and design constraints such as process scaling, power, energy consumption, etc. make it essential to optimize the design across architectural boundaries. This dissertation solves some of the problems of Emerging Memory Technology based Main Memory (EMT) by taking advantage of the memory state of the program and using it to reduce read/write access to EMT.

Chapter 3: Related Work

3.1 Identification of Invalid Data

Prior research in making informed decisions based on the block status and future usefulness has been done both in software and hardware. Some techniques have tried to identify block usefulness at the software level [SVMW05, WMRW02] while others have attempted to do so at the hardware level [AGVO05, HKM02, KS08, LFF01]. Most of these techniques try to identify blocks that are not likely to be used in the near future. Software solutions do this by passing hints to the hardware about blocks that are thought to be not likely to be used in the near future to the hardware – based on inferences from profiling or compiler analysis [SVMW05, WMRW02]. Hardware solutions employ predictors to predict those blocks that are not likely to be used in the near future. The predictor does the prediction of future usefulness based on the data address [HKM02] or the program counter (PC) [AGVO05, KS08, LFF01]. All these approaches differ significantly from the approach in this dissertation due to the fact that they predict the likelihood of usage and attempt to use that while this dissertation uses the knowledge from program semantics about validity of a block from the programs perspective. This dissertations approach is different from predicting how likely a block is to be used in the near future and thereby predicting how useful it is to keep a certain block in the cache. The knowledge based on program semantics allows one to optimize the DRAM subsystem to avoid some of the access operations without fear of incorrectness; the same cannot be done with predictive techniques. Lewis et al. [LBL02] explored using program semantic information about allocated space for caches and at cache block granularity to improve performance. Additionally, energy savings were never explored in Lewis et al. [LBL02]. Sartor [S10] extended the ideas of inconsequential write back for Java benchmarks. An upper and lower bound of the free heap was employed in Sator's

research to track Java memory. Such a mechanism would not work on native (unmanaged) benchmarks due to the lack of a garbage collector compacted and well defined contiguous heap. Jouppi [J93] investigated a cache policy, 'write-validate", which does word-level sub-blocking [C96]. In the 'write-validate', policy data for the write is not fetched but rather written directly to the cache line with the valid bits turned off for all but the data being written. Thus, write-validate could potentially eliminate all write misses; but the implementation overhead of this scheme is significant. Wulf and McKee [WM95] proposed having a "first write" instruction to bypass cache stall due to write miss. The PowerPC instruction set has an instruction *dcbz* geared towards this end. This dissertation proposes a few instructions, the application of which transcends write misses and helps to track several different artifacts of data values and reap benefits from them.

3.2 DEAD CACHE BLOCK LIFETIME PREDICTIONS

The application of memory state knowledge has some similarity to existing uses of block lifetime prediction. Some of the uses explored in prior research related to block lifetime prediction are prefetching [HKM02, LFF01], replacement [KS08], bypassing [GAV95, JS03, JCMH99, RTDF98, TFMP95], coherence protocol optimizations [LW95, LF00, SWHKAF04] and to a limited extend power reduction [KHM01, AGVO05]. Works by Lai et al. [LFF01], Hu et al. [HKM02], Ferdman and Falsafi [FF07] use the predictions of the block lifetime to trigger prefetches; Lai et al. [LFF01], Hu et al. [HKM02], prefetched into the L1 data cache while Ferdman and Falsafi did the same from off-chip to on-chip memory. Kharbutli and Solihin [KS08] used the knowledge of block lifetime to improve the LRU algorithm by replacing the dead blocks first and then bypassing the cache. Cache coherence protocols have also been tuned to take advantage

of block lifetime prediction to maintain or avoid status updates. Lebeck and Wood [LW95] proposed a reduction in cache coherence protocol overhead by invalidation some of the shared cache blocks early. Lai and Falsafi [LF00] employed a predictor based on program counter (PC), to predict last-touch and decide when blocks should be invalidated. PC-traces are used to identify last stores to a cache block in Somogyi et al.'s work [SWHKAF04]. There were also proposals for power saving techniques based on block lifetime prediction work by turning off (Kaxiras et al. [KHM01]) or gating (drowsy caches [FKMBM02]) transistors. The ideas in this dissertation can help most of these usage models while complementing the already existing ideas.

3.3 Power and Energy DRAM Optimization

Venkatesan et al. in [VHR06] introduced a retention-aware placement algorithm which tried to reduce the refresh operations by experimentally identifying that, different rows require different refresh times. Mrinmoy et al. [GL07] suggested a technique to identify rows that were refreshed by a memory access and avoid refreshing those rows when possible. Murakami [OKM98] presents the benefits of selective DRAM refreshing using OS or compiler, however they do not describe how exactly this is done. It is a limit study evaluating the benefits of capturing all condition where refresh can be avoided. In this dissertation there are descriptions and evaluation of mechanisms to achieve part of the benefits.

3.4 Emerging Memory Technology

Work related to emerging memory technologies such as MRAM, PCM, FeRAM etc, is relatively new in computer architecture. The published works have focused on analyzing the prospects of PCM as well as techniques to improve its life time and hide/tolerate its latency. Qureshi et al. [QSR09] proposes a wear leveling technique to

shift cache lines within a page which makes the wear more uniform over all lines in the page. A more refined technique is proposed by Qureshi et al. recently [Q09]. The work by Zhou et al. [ZYZ09] proposed shifting bits in a line, shifting lines in a segment, and segment swapping; all of which are various layers of wear leveling. Another approach to reducing the wear of PCM is to reduce the write traffic to PCM. Lee et al. [LIMB09] proposed partial write buffers which allow only modified data to be written to the PCM thereby avoiding unnecessary writes. Line level write back was proposed by Qureshi et al. [QSR09] which attempts to write to the PCM at a line level granularity. A line level write back technique could be complex since it requires the write buffer mechanisms to operate at a much finer granularity than the typical memory controller. Qureshi et al. [QSR09] also proposed lazy write which is essentially a cache for the PCM memory and use this to hide some of the latency of PCM. The idea of silent store removal [LBL01] exploits the fact that some of the stores tend to write the same data over and over again which is unnecessary and can be avoided. Zhou et al. [ZYZ09] proposed to reduce the write energy spent on MRAM by avoiding redundant writes, i.e. when the value written back is the same as the old value. They modify the read/write logic and perform a read operation parallel to the write operation and abort the write operation if the read value is the same as the value to be written. There have been several fine-grained approaches that try to reduce unnecessary writes to the EMT memory, including data comparison write (DCW) [YLK07], Flip-N-Write [SL09], and many others [ZYZ09, ZL09, YND10]. These fine-grained ideas utilize read-before-write to detect modified data and potentially selectively invert bits.

Another class of proposals, known as wear-leveling, improve lifetime by distributing writes equally to all cells in the device. Wear-leveling is commonly used with FLASH memory. Researchers have adapted ideas from FLASH memory to perform

wear-leveling in EMT's by performing row shifting [ZYZ09, SLSB10], word shifting [ZL09], randomized address mapping [Q09,QSR09, SWL10] and data remapping [SLSB10, YMC11].

These schemes are complementary to ideas in this dissertation and contribute towards making emerging memory technologies much more viable an option.

3.5 VALUE LOCALITY AND CACHE FILTERS

Prior work focusing on data value has been centered on value locality. For example, Lipasti et al. [LWS96] pointed out that load instructions exhibit value locality and proposed that there is potential for prediction. Last-value predictors, stride predictors, context predictors, and hybrid predictors have been proposed to predict load values [BZ02, CR00, LWS96]. In the large body of work related to value and stride prediction, the focus has been on performance and hence attempts to exploit value locality in memory operations has been focused on loads since loads are performance critical and on the critical path of program execution. What this dissertation presents is based on a similar intuition but the focus is on store operations.

Kin and Magione-Smith [KMS97] introduced the filter cache, a small cache placed between the CPU and L1, to achieve power reduction. This works well for embedded application but not so well for larger workloads especially modern workloads with a large appetite for heap memory. The Frequent Value Cache (FVC) was proposed by Yang and Gupta [YG02] and works by encoding frequently used values in a compressed format. FVC saves time in accessing frequently used values but suffers performance loss for non-frequently used values. Islam and Stenstrom proposed Zero-Value Caches [IS09] to filter out loads that load data whose values are zero. Their idea

works by placing a small cache dedicated to storing zero value data in a small separate cache directly accessible by the CPU.

The optimizations in this dissertation rely on semantic information available from the program allowing for the system to act without fear for correctness. The techniques can work in a complementary fashion with most of the previous power saving techniques. It could also be applied to other areas were block lifetime prediction has been put to use to but the converse is not true since it requires accurate information.

Chapter 4: Experimental Methodology and Benchmarks

This chapter reviews the benchmarks and methodology used for experiments and measurements.

4.1 BENCHMARKS

Languages such as Java, C++ and C# have become the languages of choice in many domains due to their object oriented nature. Object oriented programs are rich with features that reduce the programmer's effort while increasing the manageability of code by encouraging modularity. The object oriented nature of programs has resulted in an increase in dynamic nature of control flow as well as memory management, an aspect that is targeted in this research.

Based on the suggestions of Phansalkar et al. [PJJ07], a subset of SPEC CPU2006 benchmark inputs were used to evaluate the optimizations. Since the experiments of interest require tracking explicit memory allocation, a subset of the SPEC CPU2006 benchmark suite was picked to avoid benchmarks which did not have explicit dynamic memory allocation or whose memory allocation could not be tracked (particularly FORTRAN code).

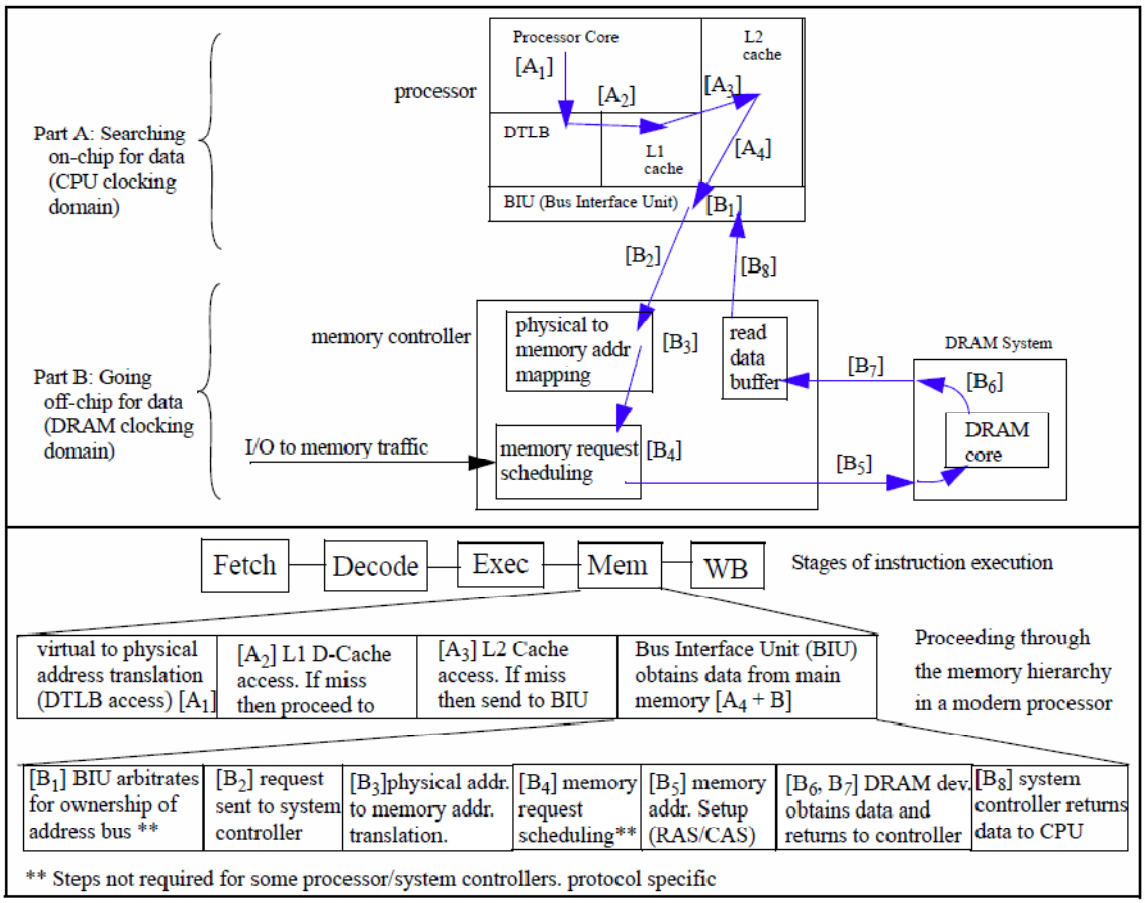


Figure 4.1 Data flow for a load instruction in a processor-memory system (borrowed Figure) [WGTBJJ05]

4.2 EXPERIMENTAL METHODOLOGY

The Figure 4.1 shows the topology and data flow of a processor-memory system. There are three separate and distinct parts of the system that interact during the life of a memory operation. These are the processor(s), memory controller(s), and DRAM memory system(s). The experimental methodology uses a cache/memory simulator built on top of PIN [PIN, PIN05] to handle the first two parts of the system. The PIN tool tracks both memory access operations (loads and stores) as well as memory management operations (*malloc*, *free, calloc, valloc, realloc, mmap, munmap* etc). This allows the

emulation of the necessary state changes corresponding to memory state i.e. inconsequential or not. For the DRAM optimizations (third part of the system) a DRAM simulator, DRAMsim [WGTBJJ05] that models both power and latency is used. DRAMsim is a hardware-validated, public-domain DRAM system simulation code that was developed by members of the Systems and Computer Architecture Lab (SCAL) in the Department of Electrical and Computer Engineering at the University of Maryland. It is a detailed and highly-configurable C-based memory system simulator which implements detailed timing models for a variety of existing memories including SDRAM, DDR, DDR2, DRDRAM and FBDIMM. It also models the power consumption of SDRAM and its derivatives. The DRAM simulator is used to model DRAM energy and is tied into the PIN memory model using the hooks on it. The simulator built, uses x86 binaries of the SPEC CPU2006 benchmarks and can simulate the allocation and de-allocation behavior in C and C++ benchmarks.

For modeling the PCM based system the PIN based model is extended to use a PCM based memory hierarchy. The PCM based memory hierarchy employs a DRAM based cache similar to Qureshi et al. [QSR09]. This model does not model energy and power but models basic scheduling as well as latency for the PCM based memory hierarchy.

All the benchmarks were run for 10%-20% of their run length for detailed memory simulation using DRAMsim. In cases where general behavior statistics were collected the benchmarks ran for the full length, if applicable. The benchmarks used in this dissertation are summarized in Table 4.1. Statistical sampling approaches such as SIMPOINT [PHBSC03] are not suited for this dissertation because all the memory allocation calls as well as their effect on the memory hierarchy has to be tracked completely to ensure correct estimation of memory state. Sampling instruction ranges

will result in loss of information regarding memory allocation and memory reads and writes which are very important in estimating the usefulness of memory ranges.

The memory system configuration assumed for the DRAM based memory is presented in Table 4.2. The memory system configuration assumed for the emerging memory technology based main memory is presented in Table 4.3.

Benchmark	Instruction Count(Billions)		Benchmark	Instruction Count(Billions)	
	Total	Simulation		Benchmark	Simulation
astar	869	87	*milc*	1178	118
bzip2	341	34	*namd*	2333	233
dealII	2007	201	*omnetpp*	611	61
gcc	57	6	*perlbench*	674	67
gobmk	328	33	*povray*	1013	101
h264ref	2595	259	*sjeng*	2319	232
hmmer	2004	200	*soplex*	381	38
lbm	136	14	*sphinx3*	3101	310
libquantum	2242	224	*xalancbmk*	1131	113
mcf	372	37			

Table 4.1 SPEC CPU2006 benchmarks used in this dissertation

Parameter	Value	Parameter	Value
Type	DDR2	Number of Columns	1024
Size	1 GB	Data Width	72 bits (64 data + 8 ECC)
Rows	16384	Refresh Interval	32ms
Frequency	667 MHz	L2 cache size	1-8 MB
Number of Banks	8	L2 cache way	8way (64 byte line)
Number of Ranks	2	L1 cache	64 KB, 2 way, 64byte line

Table 4.2 Memory subsystem configuration for DRAM based memory

L1	64 KB, 64 byte line, 2 way		DRAM cache	64 MB, 320 cycle latency
L2	2 MB, 64 byte line, 16 way		EMT(PCM)	4 GB, 1280 cycle latency

Table 4.3 Memory subsystem configuration for emerging memory technology based memory

Chapter 5: Memory State Knowledge Based Optimizations

Computer systems tend to operate agnostic of semantic information about memory state and the value which is present in the program and fail to exploit this knowledge to optimize the memory hierarchy to enhance energy, reliability and performance characteristics of the memory hierarchy. Although there have been memory hierarchy optimizations such as cache-locking, cache-bypass, prefetching etc, most of the modern optimizations done in the microarchitecture and memory subsystem tend to be agnostic of program semantics. This dissertation harnesses memory state and value information from the semantics of the program and exploits it to optimize the main memory system.

5.1 Concept of Inconsequential Memory

When a program or operating systems memory manager allocates or frees up a memory region, this program semantic information is used by the architecture to optimize the working of the memory system. The idea of inconsequential memory is explained with some examples in this section. Most program languages provide means for dynamic memory allocation (implicitly or explicitly). Considering its wide use and understanding, the constructs and assumption of C language will be used for the purpose of examples. For example, malloc, calloc, or realloc etc allocate a region of memory whose existing content is uninitialized. The uninitialized memory is inconsequential until actual data is written to it by the program. The malloc function prototype is

*void *malloc (size_t size);*

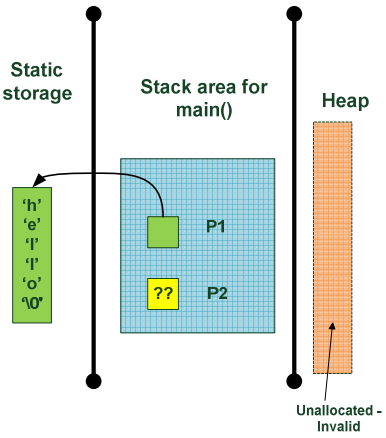
this prototype allocates *size* bytes of memory. If the allocation is successful, a pointer to the block of memory is returned. If it fails, a null pointer is returned. The pointer returned by malloc is a void pointer (void *), indicating the lack of any known data type. This

pointer can be cast to the necessary type and assigned to a pointer variable. The memory allocated via malloc is persistent, i.e. it will continue to exist until it gets explicitly deallocated by the programmer (in the code) or the program terminates. The explicit deallocation (freeing the block of memory) is done with the help of the function “free”. Its prototype is

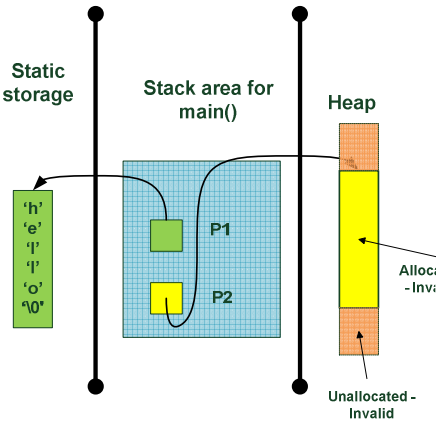
*void free (void *pointer);*

and this prototype releases the block of memory pointed to by *pointer*. The address, *pointer* must have been returned previously by memory allocation functions such as malloc, calloc, or realloc etc. Once the address is *freed* any access to this memory location will be erroneous and its behavior undefined. Hence, the programmer will not be using this location after it is freed and many modern programming languages have mechanisms to prevent such erroneous accesses. Consider the following example program:

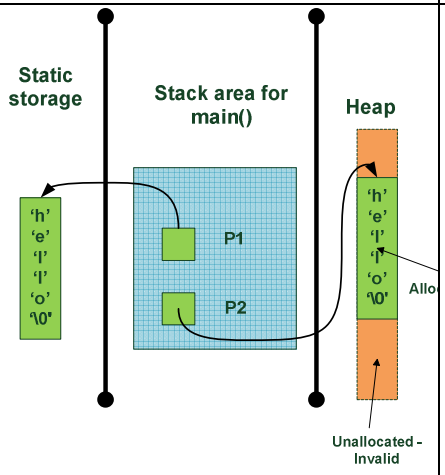
```
1.    int main (void) {
2.          const char *p1 = "hello";
3.           char *p2;
4.          p2 = malloc (strlen (p1) + 1);
5           strcpy (p2, p1);
6.          free (p2);
7.          return 0;       }
```



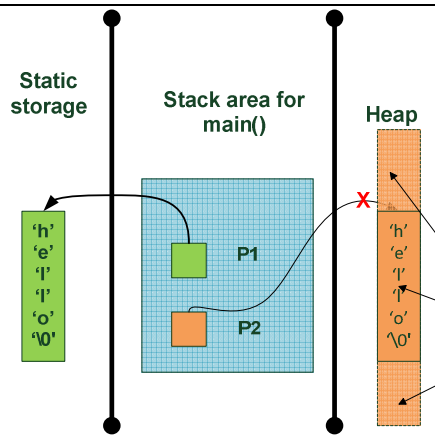
(a) No data region is reserved or in use in the heap; i.e. all the heap is unallocated



(b) Part of the heap is reserved for use by pointer p2, but the reserved region still contains no useful data.



(c) Memory region is used for storing data.



(d) Memory reserved for pointer p2 is now unallocated and hence it is in the same state as the rest of the heap.

Figure 5.1 Memory state during different stages of memory allocated, in use and unallocated

The diagrams (Figure 5.1 (a) – (d)) show the situation at line 3, line 4, line 5 and line 6. At line 3, the pointer p2 has no memory allocated to it. At line 4, the pointer p2 has memory allocated to it but has no useful data occupying the allocated space and at line 5 the pointer p2 has data copied into the allocated space. Note that at line 6 the pointer variable p2 still contains the address of a byte in the free store but there is no longer an allocated block at that address. It would be a serious error to actually use that address. Thus, according to program semantics, the address pointed to by p2 is not expected to contain useful data after line 6. Similarly, if pointer p2 was read before the strcpy function (line 5) the result would be an error or some random data. Thus before strcpy function writes data to the allocated memory it contains random data and the program does not read from it without writing to it first. In both these cases the data present is invalid, hence inconsequential to the program execution.

Therefore, some memory accesses are inconsequential memory accesses, i.e. the transfer of data between the caches, the memory and the disk drive are those that have not been initialized by the program, or have already been released by the program. The data is transferred by the hardware based on demand, regardless of the memory state. During a typical program execution all structures such as the stack and heap contain inconsequential data until they are allocated to some data structures and are initialized. Figure 5.2 illustrates the different states a memory location can be in as a result of memory management. An unallocated-invalid memory location becomes allocated-invalid after allocation. It remains in this state until the memory location is written to, causing it to transition to the allocated-valid state. From this point on, that memory location cannot be considered inconsequential. Eventually, when the program is done with the memory location, a free operation causes it to be returned to the heap; i.e. the location transition from an allocated-valid state to an unallocated-invalid state.

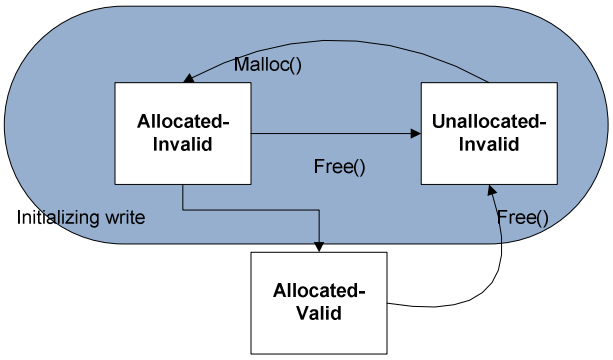


Figure 5.2 Inconsequent memory state transitions

5.2 ZERO VALUE DATA

Another important memory state this dissertation focuses on relates to the sparse nature of data and its value in stores. In most programming languages, data structures are often initialized to zeros. Furthermore, very often, large data sets contain arrays and matrices which are sparse matrices heavily populated with zero. It is found that stores too have value sparseness, particularly for zero data value.

5.3 KEY MICROARCHITECTURAL IDEAS

There are three major insights that come from the memory state:

a) Inconsequential write backs.

b) Inconsequential write miss.

c) Zero Value Stores.

The details of how this is applied and implemented along with its results are presented in the following chapters. The remainder of the present chapter explains the key microarchitectural ideas of this dissertation.

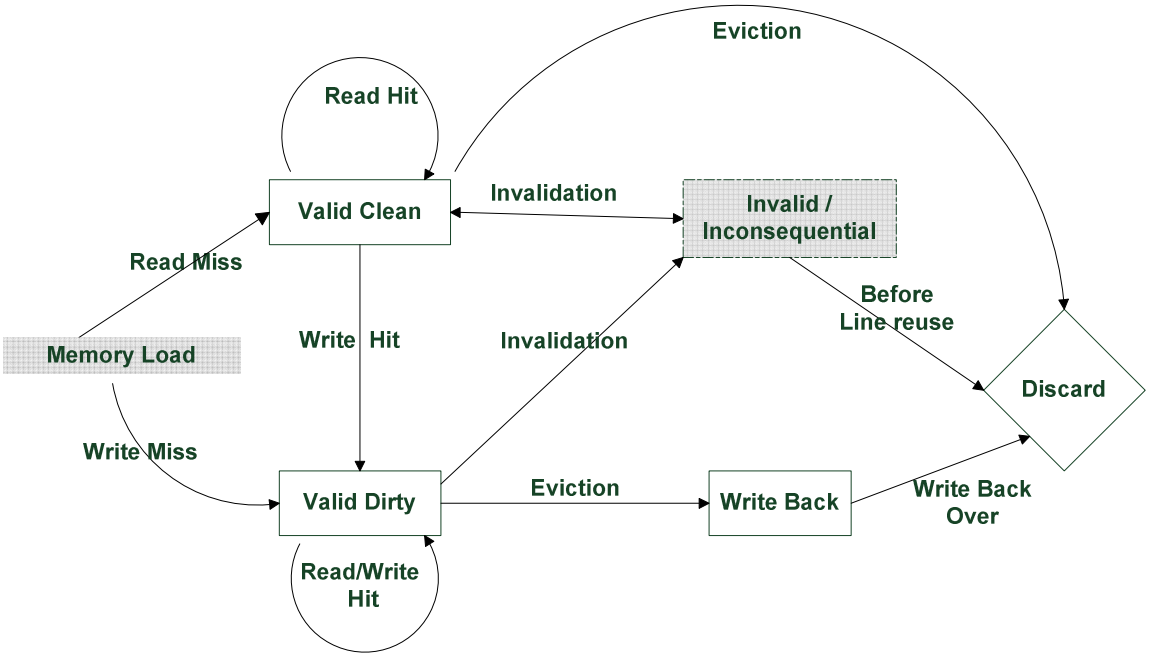


Figure 5.3 Cache line states (write back and write allocate cache)

5.3.1 Inconsequent Write Backs (IWB)

The cache hierarchy of a processor operates with a write through or a write back policy. Most modern caches tend to be write back. Figure 5.3 is an illustration of the different states of a cache line. In a write back cache, writes are not immediately stored into the memory. Instead, the cache keeps track of the locations that have been written over (marks them as dirty). The data in the modified locations are written back to the memory when the data is evicted from the cache due to the cache replacement policy. Hence a read miss in a write back cache, i.e. replacing a block with another, will require two memory accesses to be serviced if the replacement candidate is in modified state –

one to fetch the necessary data for the read and one to write the replaced data from the cache to the store.

Programs typically allocate memory regions to compute and store data that may persist across different parts of the program in somewhat temporal fashion. Modified data stored in an allocated region will appear as dirty data, but once the temporary use of the memory region is over (marked by a call to free), the correct execution of the program does not use the data stored in these memory addresses. The data is in inconsequent state since the data stored in a location that has been freed is of no consequence to the correct execution of the program. The inconsequential memory region information can be stored in the cache line using the invalid bit that already exists to mark the cache line as inconsequential.

When a modified and inconsequential line is evicted, one can avoid writing the replaced data to the next level of the memory. Furthermore, marking an inconsequential line as invalid makes it the next candidate for replacement which can improve the efficiency of the cache and thereby reduce load latency. Doing so would also reduce the number of writes performed on the memory device.

Sartor [S10] in her dissertation extends my idea of inconsequential write back for Java benchmarks. An upper and lower bound of the free heap was employed to track Java. Such a mechanism would not work on native (unmanaged) benchmarks due to the lack of a garbage collector compacted and well defined contiguous heap but lends credit to the applicability and expandability of my proposal.

5.3.2 Inconsequential Write Miss Servicing (IWM)

A write miss occurs in a cache when a store attempts to store data and none of the cache ways contain the necessary cache line. In a cache that has write allocate policy, a

write miss would result in a corresponding fetch from memory and store in a cache line as shown in Figure 5.3. The necessary cache line is fetched from memory and brought to the cache so that the write to the cache can proceed. Fetching the data from the memory hierarchy during a write miss is important for correctness, particularly for partial writes (writes a fraction of the cache line in size). If the write miss's target data is not brought from the main memory and the data write executed, the data will be written to the cache line with the rest of the cache line filled up with random data. Since the write miss candidate cache line will be marked as dirty after a write operation, a future eviction to this cache line would result in the whole cache line (including the random data) being written back to the main memory, resulting in loss of data. To avoid such data corruption, a write miss requires a corresponding read to memory.

The memory management library allocates and frees data for the program and these operations on the heap are done on a contiguous memory range. A write to a newly allocated memory space resulting in a write miss gains no useful data by fetching the data from memory since the data present in this address range is ”inconsequent”.

Lewis et al. [LBL02] explored using program semantic information about allocated space for caches by tracking limited allocation ranges in the cache to improve performance. Lewis's approach suffers when there is very active dynamic memory use because it causes the allocated ranges to become very fragmented. Jouppi [J93] investigated a cache policy, 'write-validate”, which does word-level sub-blocking [C96]. In Jouppi's policy, data for the write is not fetched but rather written directly to the cache line with the valid bits turned off for all but the data being written. 'Write-validate' could potentially eliminate all write misses; but the implementation overhead (one bit per word in caches) of this scheme is significant.

5.3.3 Zero-Value Stores (ZVS)

In the past, data patterns were explored in loads by works related to Value Prediction. Store operations arising from the program too can have certain data patterns. This dissertation focuses the intuition of value locality on to stores since data structures are often initialized to zeros. Initializing to zero is also often the case with the heap/generational regions of managed languages. Furthermore, very often large data sets contain arrays and matrices which are sparse matrices heavily populated with zero. Past studies, on the data distribution of zero value data stored in the memory done by Ekman and Stenstrom [ES05], demonstrated that a significant amount of memory contains zero value. When stores write zeros to memory, this dissertation attempts to reduce the number of stores to the memory by capturing it as compressed information. This can help condense and bypass stores to the memory.

Prior work focusing on data value was centered on value locality. Last-value predictors, stride predictors, context predictors, and hybrid predictors have been proposed to predict load values [BZ02, CR00, LWS96]. A Frequent Value Cache (FVC) was proposed by Yang and Gupta [YG02] which works by encoding frequently used values in a compressed format. FVC saves time in accessing frequently used values but suffers performance loss for non-frequently used values. Islam and Stenstrom proposed Zero-Value Cache [IS09] to filter out the loads that load data values of zero. Zero-Value cache works by placing a small cache dedicated to storing zero value data in a small separate cache directly accessible by the CPU. In the past, the focus has been on performance and hence attempts to exploit value locality in memory operations has been focused on loads. Loads were targeted because loads are performance critical and on the critical path of program execution. What this dissertation presents is based on a similar intuition but the focus is on store operations.

5.3.4 Application to DRAM and EMT memory

In Chapter 7 and Chapter 8 the ideas of IWB, IWM and ZVS are exploited to optimize DRAM energy consumption and the lifetime of emerging memory technology (EMT) based memory.

In Chapter 7, ESKIMO uses the concept of IWB and IWM and reduces such access by storing information regarding inconsequential memory in the TLB and page table. ESKIMO helps reduce activity to the DRAM by avoiding access that are IWM or IWB. In addition, the inconsequential memory blocks in DRAM's are identified and avoided during refresh operations since the data fidelity of blocks known to be inconsequential is not important.

Chapter 8 presents mFilter, an augmentation to an emerging memory technology based memory hierarchy. This filter stores information regarding allocated and deallocated memory as well as memory regions containing zero value data. This way the mFilter helps avoid IWB and ZVS related write operations as well as IWM related load operations.

Chapter 6: Workload Characterization

Before perusing mechanisms to exploit inconsequential data, it would be valuable to profile the workloads to understand the amount of such data in them. In this chapter, some preliminary profiling of the benchmarks is performed to get an insight into their inconsequential state as well as zero data value.

6.1 Dynamic Memory Allocation and Free Pattern

Any microarchitectural technique that aims to exploit the temporal existence of inconsequential memory has to track as well as signal memory ranges based on allocation and free patterns. For this reason, it is important to look at the benchmarks with an interest in their data allocation and data free behavior. In Table 6.1 and 6.2 we have the allocation call behavior of the SPEC CPU2006 benchmarks used in this dissertation. The calls to allocation memory were recorded and categorized according to their granularity. In general, the smaller the granularity, the higher is the tracking overhead. It is desired that most of the dynamic memory allocation and free related behavior are of large size granularity thereby enabling easier tracking of inconsequential state arising from it. The reasoning behind this is that smaller granularity of state leads to fragmented memory which requires a larger number of state storage memories. Implementing state storage, related to tracking, in the hardware can be a significant cost. In Table 6.1 we see that benchmarks such as *dealII, gcc, gobmk, libquantum, povray, sphinx3* and *xalancbmk* have majority of the allocation calls for issuing memory that is of any size smaller than 64 byte, the size of a cache line assumed in this dissertation. On average the top 3 granularity of allocation for allocation calls are less than 64 byte, 256 B to 1 KB and 4 KB to 256 KB.

Benchmark[1]	% of Allocation Calls per Size Bucket								Total Calls (1K)
	<64	<256	<1K	<4K	<256K	<1M	<16M	>16M	
*astar**	21.7	0.8	74.1	3.0	0.5	0	0	0	1117
bzip2	0	0	0	0	46.4	10.7	32.1	10.7	<1
*dealII**	92.5	6.5	0.9	0.1	0	0	0	0	153873
*gcc**	64.8	21.0	2.0	4.3	7.8	0	0	0	2920
gobmk	73.6	0	0.2	24.3	1.8	0	0	0	119
*h264ref**	4.8	1.4	87.5	0.8	4.8	0.6	0.1	0	105
*hmmer**	3.1	14.8	74.7	7.3	0	0	0	0	1000
lbm	0	0	50	0	0	0	0	50	<1
libquantum	28.9	2.2	2.2	2.2	8.3	16.7	23.3	16.1	<1
mcf	0	0	40	0	0	0	40	20	<1
*milc**	0.1	0.1	0	0	0	0.2	61.6	38.0	7
*namd**	0.2	0.3	41.5	0.3	56.4	0.2	1.1	0	1
*omnetpp**	18.8	81.2	0	0	0	0	0	0	267065
*perlbench**	18.1	63.5	0.3	0.3	15.0	2.8	0	0	22917
*povray**	96.3	2.2	1.4	0	0.1	0	0	0	2462
sjeng	0	0	20	0	0	0	20	60	<1
*soplex**	1.3	0.4	0.9	0.8	89.7	5.3	1.3	0.3	9
*sphinx3**	65.5	0.9	5.7	27.0	0.8	0	0	0	14225
*xalancbmk**	67.5	5.1	12.1	13.8	1.5	0	0	0	135184
AVG	**29.33**	**10.55**	**21.76**	**4.43**	**12.27**	**1.92**	**9.45**	**10.27**	**31632**

Table 6.1 Granularity of allocation calls (allocation calls per size)

[1] Benchmarks that are marked by an asterisk symbol are the ones that demonstrate better response to Inconsequential Write Miss

Benchmark[2]	% of Allocated Bytes per Size Bucket								Total Allocated Bytes(MB)
	<64	<256	<1K	<4K	<256K	<1M	<16M	>16M	
*astar**	1.1	0.1	79.4	7.2	4.6	1.8	5.8	0	997
bzip2	0	0	0	0	0.1	0.1	3.8	96.0	628
*dealII**	49.2	13.9	6.6	3.1	4.8	2.6	5.1	14.6	10819
*gcc**	0.8	0.8	0.5	5.0	89.5	2.4	0.9	0	6634
gobmk	1.2	0	0.1	71.4	8.5	0.8	18.1	0	123
*h264ref**	0	0	4.3	0.1	22.9	28.8	43.9	0	1026
*hmmer**	0.2	4.4	77.6	15.1	0.2	2.5	0	0	545
lbm	0	0	0	0	0	0	0	100	409
libquantum	0	0	0	0	0.1	1.8	37.0	61.2	1486
mcf	0	0	0	0	0	0	0.5	99.5	1676
*milc**	0	0	0	0	0	0	34.8	65.2	84226
*namd**	0	0	1.2	0	28.2	1.6	69.0	0	45
*omnetpp**	2.7	97.0	0	0	0.2	0	0	0	42503
*perlbench**	0	0.2	0	0	68.0	31.7	0	0	59254
*povray**	42.5	6.1	14.5	0.2	36.7	0	0	0	114
sjeng	0	0	0	0	0	0	6.7	93.3	172
*soplex**	0	0	0	0	9.0	5.1	17.4	68.5	3186
*sphinx3**	1.3	0.1	1.8	77.9	8.5	10.3	0.2	0	15398
*xalancbmk**	2.8	1.6	15.8	60.7	18.9	0.1	0	0	59352
AVG	**5.36**	**6.54**	**10.62**	**12.67**	**15.80**	**4.72**	**12.80**	**31.49**	**15189**

Table 6.2 Distribution of allocated bytes (allocated bytes per size)

[2] Benchmarks that are marked by an asterisk symbol are the ones that demonstrate better response to Inconsequential Write Miss

In Table 6.2 we have the amount of memory allocated in bytes recorded per size class i.e., a distribution of the bytes allocated more than the allocation function call count. The amount of dynamic memory allocated is more important than the number of calls belonging to a class of allocation granularity. With the exception of *dealII* and *povray*, all the benchmarks allocate data larger than 64 bytes. In fact, on an average, the top 3 size classes of dynamic memory allocated are larger than 16 MB, 4 KB to 256 KB and 1 MB to 16 MB. The benchmarks that are marked by an asterisk symbol are the ones that demonstrate better response to IWM according to results shown is the later chapters. This is encouraging because even *dealII* and *povray* tracking mechanisms, with a minimum granularity of a cache line, gave reasonable results. In fact, by constructing a cumulative distribution from Table 6.2 we see that a tracking size of 64 bytes or greater can account for 95% of the allocated bytes while 256 bytes, 1 KB and 4 KB size tracking can account for 88%, 78% and 65% of the total allocated bytes.

In Table 6.3 and 6.4, the free call behavior of the SPEC CPU2006 benchmarks used in this dissertation are presented. The calls to deallocate memory were recorded and categorized according to their granularity. The reasoning for granularity of free sizes is similar to the previous rational about allocation. In Table 6.3 we see that in benchmarks such as *dealII, gcc, gobmk, libquantum, povray, sphinx3* and *xalancbmk,* the majority of the free calls issue memory that is smaller than 64 byte, the size of a cache lines assumed in this dissertation. The pattern for free calls is similar to the observation for allocation pattern before. On an average the top 3 granularity of allocation for allocation calls are less than 64 bytes, 256 to 1 KB and 4 KB to 256 KB.

Benchmark[3]	% of Deallocation (*Free*) Calls per Size Bucket								Total Calls (1K)
	<64	<256	<1K	<4K	<256K	<1M	<16M	>16M	
*astar**	21.7	0.8	74.1	3.0	0.5	0	0	0	1117
bzip2	0	0	0	0	50	12.5	37.5	0	<1
*dealII**	92.5	6.5	0.9	0.1	0	0	0	0	153873
*gcc**	65.1	20.9	1.9	4.3	7.8	0	0	0	2876
gobmk	71.9	0	0.2	26.0	1.8	0	0	0	104
*h264ref**	4.8	1.3	87.5	0.8	4.8	0.6	0.1	0	105
*hmmer**	3.1	14.8	74.8	7.3	0	0	0	0	1000
lbm	0	0	50	0	0	0	0	50	<1
libquantum	39.1	1.6	1.6	1.6	7.8	21.9	4.7	21.9	<1
mcf	0	0	40	0	0	0	40	20	<1
milc	0	0	0	0	0	0	61.7	38.3	6
namd	0.2	0.3	41.5	0.3	56.5	0.1	1.1	0	1
*omnetpp**	18.8	81.2	0	0	0	0	0	0	266999
*perlbench**	19.7	77.2	0.3	0.2	2.5	0	0	0	18646
*povray**	97.6	2.0	0.3	0	0	0	0	0	2427
sjeng	0	0	100	0	0	0	0	0	<1
soplex	3.1	0.5	1.9	1.5	84.1	7.4	1.5	0.2	4
sphinx3	65.0	0.9	5.8	27.4	0.8	0	0	0	14024
*xalancbmk**	67.5	5.1	12.1	13.8	1.5	0	0	0	135184
AVG	**30.01**	**11.22**	**25.94**	**4.54**	**11.48**	**2.24**	**7.72**	**6.86**	**31388**

Table 6.3 Granularity of deallocation calls (free call per size)

[3] Benchmarks that are marked by an asterisk symbol are the ones that demonstrate better response to Inconsequential Write Miss

In Table 6.4 we have the amount of memory freed in bytes recorded per size class i.e., a distribution of the bytes allocated. Now we see that with the exception of *dealII* and *povray*, all the benchmarks allocate data larger than 64 bytes. In fact, on an average the top 3 size classes of dynamic memory mostly allocated are larger than 16 MB, 4 KB to 256 KB and 256 KB to 1 KB. The benchmarks that are marked by an asterisk symbol are the ones that demonstrate better response to IWB in results shown in later chapters. This is encouraging because even in *dealII* and *povray* tracking mechanisms with a minimum granularity of a cache line gave reasonable results. In fact, by construction a cumulative distribution from Table 6.4, we see that a tracking size of 64 bytes or greater can account for 93% of the allocated bytes while 256 bytes, 1 KB and 4 KB size tracking can account for 86.4%, 71% and 57% of the total allocated bytes.

In addition to *malloc* and *free* operations there are larger system level functions to allocate memory such as *mmap* and *munmap*. Both these functions (*mmap* and *munmap*), are used to allocate large chunks of memory in granularities of the page size. They are typically used by the memory manager or allocator to allocate memory used to construct, replenish, scale down or empty the dynamic heap, a reservoir of memory space. The heap is then used to allocate dynamic memory requests for memory over the course of the program. Thus *mmap* and *munmap* are the calls that fill up and empty the heap, which is, in turn, used by the memory manager or allocator to provide for on demand dynamic memory quickly. Table 6.5 presents the ratios of memory allocated via *malloc* calls to the amount of memory allocated via *mmap* i.e. a representation of the amount of dynamic memory recycling. This is also an indication of the rate of allocation and free operations because, the higher the rate of allocations followed by free, the more capable it is in reusing the heap space. Thus, programs with higher rate of dynamic memory recycling

require a smaller amount of heap space to satisfy the dynamic allocation. If most of the heap fits in the cache, it is unlikely for IWB and IWM to help.

Benchmark[4]	% of Deallocated (*Freed*) Bytes per Size Bucket								Total Bytes (MB)
	<64	<256	<1K	<4K	<256K	<1M	<16M	>16M	
*astar**	1.1	0.1	79.4	7.2	4.6	1.8	5.8	0	997
bzip2	0	0	0	0	1.5	3.0	95.5	0	25
*dealII**	49.2	13.9	6.6	3.1	4.8	2.6	5.1	14.6	10819
*gcc**	0.8	0.8	0.5	5.0	89.5	2.5	0.9	0	6521
gobmk	1.5	0	0.1	88.5	9.9	0	0	0	93
*h264ref**	0	0	4.3	0.1	22.9	28.8	43.9	0	1026
*hmmer**	0.2	4.4	78.1	15.2	0.2	1.9	0	0	542
lbm	0	0	0	0	0	0	0	100	409
libquantum	0	0	0	0	0.1	2.6	4.9	92.4	951
mcf	0	0	0	0	0	0	0.5	99.5	1676
milc	0	0	0	0	0	0	34.9	65.1	83613
namd	0	0	1.2	0	28.5	0.8	69.5	0	45
*omnetpp**	2.7	97.0	0	0	0.2	0	0	0	42499
*perlbench**	0.3	5.3	0.1	0.3	82.0	12.0	0	0	23383
*povray**	66.5	8.1	5.4	0.1	20	0	0	0	73
sjeng	0	0	100	0	0	0	0	0	<1
soplex	0	0	0	0	14.6	8.1	23.3	54.0	1160
sphinx3	1.3	0.1	1.8	78.1	8.4	10.3	0	0	15358
*xalancbmk**	2.8	1.6	15.8	60.7	18.9	0.1	0	0	59352
AVG	**6.65**	**6.91**	**15.44**	**13.59**	**16.11**	**3.92**	**14.96**	**22.40**	**13081**

Table 6.4 Distribution of deallocated bytes (freed bytes per size)

[4] Benchmarks that are marked by an asterisk symbol are the ones that demonstrate better response to Inconsequential Write Miss

Benchmark	**Allocated/ *Mmap'd***	***Freed/ munmap'd***	**Benchmark**	**Allocated/ *Mmap'd***	***Freed/ munmap'd***
astar	24	33	*milc*	154	3801
bzip2	1	2	*namd*	1	1
dealII	7	7	*omnetpp*	2237	5312
gcc	332	502	*perlbench*	871	866
gobmk	4	93	*povray*	10	0
h264ref	27	32	*sjeng*	1	0
hmmer	78	181	*soplex*	86	2
lbm	1	1	*sphinx3*	497	1920
libquantum	2	1	*Xalancbmk*	1915	2968
mcf	1	1	**AVG**	**329**	**827**

Table 6.5 Allocated to *mmap'd* and freed to *munmap'd* memory ratios

Tables 6.6 to 6.9 present data similar to 6.1 to 6.4 but for *mmap* and *munmap*. These calls have a minimum granularity of a page size because it is the granularity at which the OS issues memory to the requester via *mmap*. Such a large granularity could simplify tracking significantly; but typical dynamic memory allocation and free behavior operates by cycling through such pages of heap memory allocated. This means the amount of memory allocated via *mmap* would be used by the memory manager or allocator as its repository of space that it issues and reclaims based on *malloc* and *free* calls. This also means that the amount of memory allocated and freed using *mmap* and *munmap* will typically be a fraction of what is requested by *malloc* and *free*.

Benchmark	% of *mmap* Calls per Size Bucket								Total *mmap* Calls
	<4K	=4K	<8K	<64K	<256K	<1M	<16M	>16M	
astar	0	24.1	3.4	10.3	6.9	17.2	37.9	0	29
bzip2	0	21.1	0	10.5	10.5	5.3	36.8	15.8	19
dealII	0	11.9	1.7	5.1	3.4	1.7	23.7	52.5	59
gcc	0	21.7	4.3	8.7	17.4	8.7	39.1	0	23
gobmk	0	91.5	0.8	1.7	0.8	0.8	4.2	0	118
h264ref	0	29.6	3.7	7.4	7.4	22.2	29.6	0	27
hmmer	0	43.8	6.3	12.5	25.0	0	12.5	0	16
lbm	0	46.7	6.7	13.3	6.7	0	13.3	13.3	15
libquantum	0	11.4	2.3	4.5	2.3	6.8	11.4	61.4	44
mcf	0	43.8	6.3	12.5	6.3	0	25.0	6.3	16
milc	0	12.5	2.1	4.2	2.1	33.3	37.5	8.3	48
namd	0	24.3	2.7	8.1	8.1	5.4	51.4	0	37
omnetpp	0	14.3	1.6	4.8	71.4	1.6	6.3	0	63
perlbench	0	8.2	1.6	3.3	13.1	27.9	45.9	0	61
povray	0	72.2	2.8	8.3	5.6	0	11.1	0	36
sjeng	0	33.3	0	16.7	8.3	0	16.7	25.0	12
soplex	0	23.3	3.3	10	23.3	10	30	0	30
sphinx3	0	90.7	0.7	1.3	2.0	0.7	4.7	0	150
xalancbmk	0	10.2	1.0	3.1	51.0	29.6	5.1	0	98
AVG	**0**	**33.40**	**2.70**	**7.70**	**14.29**	**9.01**	**23.27**	**9.61**	**47**

Table 6.6 Granularity of allocation calls (*Mmap* calls per size)

Benchmark	% of *mmap'd* Bytes per Size Bucket								Total Bytes (MB)
	<4K	=4K	<8K	<64K	<256K	<1M	<16M	>16M	
astar	0	0.1	0	0.2	0.4	6.3	93.0	0	41
bzip2	0	0	0	0	0	0	3.1	96.8	623
dealII	0	0	0	0	0	0	4.5	95.5	1575
gcc	0	0.1	0	0.2	3.3	6.3	90.1	0	20
gobmk	0	1.4	0	0.1	0.4	3.2	94.8	0	30
h264ref	0	0.1	0	0.1	0.7	7.7	91.4	0	38
hmmer	0	0.4	0.1	0.5	10.7	0	88.3	0	7
lbm	0	0	0	0	0	0	1.4	98.5	415
libquantum	0	0	0	0	0	0.1	2.0	97.9	866
mcf	0	0	0	0	0	0	0.8	99.2	1682
milc	0	0	0	0	0	1.8	4.7	93.5	546
namd	0	0.1	0	0.2	0.9	1.6	97.2	0	43
omnetpp	0	0.2	0	0.4	40.6	1.3	57.5	0	19
perlbench	0	0	0	0.1	1.8	9.7	88.4	0	68
povray	0	0.9	0.1	0.6	1.6	0	96.8	0	11
sjeng	0	0	0	0	0.1	0	8.4	91.5	175
soplex	0	0.1	0	0.2	2.6	3.8	93.3	0	37
sphinx3	0	1.7	0	0.1	1.8	2.5	93.9	0	31
xalancbmk	0	0.1	0	0.2	29.5	29.1	41.0	0	31
AVG	**0**	**0.27**	**0.01**	**0.15**	**4.97**	**3.86**	**55.29**	**35.42**	**329**

Table 6.7 Distribution of allocated bytes (*Mmap'd* bytes per size)

Benchmark	% of *munmap* Calls per Size Bucket								Total
	<4K	=4K	<8K	<64K	<256K	<1M	<16M	>16M	Calls
astar	0	7.1	0	0	7.1	35.7	50	0	14
bzip2	0	0	0	0	12.5	12.5	75.0	0	8
dealII	0	0	0	0	2.3	2.3	23.3	72.1	43
gcc	0	8.3	0	0	25.0	16.7	50	0	12
gobmk	0	99.0	0	0	1.0	0	0	0	104
h264ref	0	17.6	0	0	11.8	35.3	35.3	0	17
hmmer	0	33.3	0	0	16.7	50	0	0	6
lbm	0	40	0	0	20	0	0	40	5
libquantum	0	0	0	0	2.9	5.9	8.8	82.4	34
mcf	0	33.3	0	0	16.7	0	33.3	16.7	6
milc	0	0	0	0	50	0	0	50	2
namd	0	5.3	0	0	10.5	5.3	78.9	0	19
omnetpp	0	4.3	0	0	93.6	2.1	0	0	47
perlbench	0	2.6	0	0	15.8	42.1	39.5	0	38
povray	0	95.2	0	0	4.8	0	0	0	21
sjeng	0	50	0	0	50	0	0	0	2
soplex	0	0	0	0	21.4	14.3	35.7	28.6	14
sphinx3	0	97.0	0	0	0.8	0.8	1.5	0	132
xalancbmk	0	3.7	0	0	59.8	35.4	1.2	0	82
AVG	**0**	**26.14**	**0.00**	**0.00**	**22.25**	**13.60**	**22.76**	**15.25**	**32**

Table 6.8 Granularity of deallocation calls (*Munmap* per size)

Benchmark	% of *munmap'd* Bytes per Size Bucket								Total Bytes (MB)
	<4K	=4K	<8K	<64K	<256K	<1M	<16M	>16M	
astar	0	0	0	0	0.4	8.6	91.0	0	30
bzip2	0	0	0	0	0.7	1.5	97.8	0	16
dealII	0	0	0	0	0	0	3.8	96.2	1564
gcc	0	0	0	0	4.0	9.8	86.1	0	13
gobmk	0	78.6	0	0	21.4	0	0	0	1
h264ref	0	0	0	0	0.8	9.1	90	0	32
hmmer	0	0.3	0	0	3.9	95.8	0	0	3
lbm	0	0	0	0	0	0	0	100	409
libquantum	0	0	0	0	0	0.1	1.3	98.6	891
mcf	0	0	0	0	0	0	0.5	99.5	1676
milc	0	0	0	0	0.5	0	0	99.5	22
namd	0	0	0	0	1.0	1.1	97.9	0	32
omnetpp	0	0.1	0	0	96.7	3.2	0	0	8
perlbench	0	0	0	0	2.8	23.3	73.9	0	27
povray	0	41.6	0	0	58.4	0	0	0	<1
sjeng	0	3.4	0	0	96.6	0	0	0	<1
soplex	0	0	0	0	0.1	0.2	5.5	94.2	533
sphinx3	0	6.3	0	0	1.4	9.7	82.6	0	8
xalancbmk	0	0.1	0	0	45.3	45.0	9.7	0	20
AVG	**0**	**6.86**	**0**	**0**	**17.58**	**10.92**	**33.69**	**30.95**	**278**

Table 6.9 Distribution of deallocated bytes (*Munmap'd* bytes per size)

6.2 LIFETIME DISTRIBUTION OF DYNAMIC MEMORY ALLOCATION AND FREE PATTERN

The ideas of inconsequential write back (IWB) and inconsequential write miss (IWM) are also affected by the rate of dynamic memory allocation and deallocation. For example, programs that allocate all the memory at the start of the program and then release them at the very end are typically not good candidates for optimizing inconsequential write backs. It would be too late in the execution cycle to identify memory as free and inconsequential. Furthermore, the longer it takes for blocks of data to be identified as free and hence as an inconsequential block, the less likely is it for those blocks to be present in the cache. The inconsequential blocks could also be replaced by newer data thereby denying IWB a chance to have any impact. Thus, a program that waits until the very end of its life to free up its dynamic memory is less likely to contribute to inconsequential write backs. On the other hand, programs that allocate memory at the start could make tracking of inconsequential write miss candidates easier because such blocks can be coalesced and identified as large blocks of freshly allocated data allowing it to be represented at larger granularity.

Given these effects and intuitions, this section presents data that charts out the dynamic memory allocation and deallocation pattern through the lifetime of the benchmark. Figures 6.1 (a) to 6.1 (j) plot two figures per benchmark where one figure shows the timeline distribution of memory allocated and the other shows the timeline distribution of memory being deallocated. Each point in the graph shows the percentage of total allocated or freed memory that was allocated or freed at that particular point in time. The x-axis represents time measured in instruction count (using multiples of 100 million). The y-axis represents the percentage of the total allocated or freed memory.

The benchmark data that is presented in the following set of paragraphs and figures can be summarized as follows:

1) Benchmark *astar* could benefit from IWB but would require a last level cache size large enough to hold the data long enough to be identified by a free operation.

2) Benchmark *bzip2's* limited dynamic memory use limits the gains from IWB but it could benefit from IWB.

3) Both *gcc* and *dealII* have good potential to benefit from IWB as well as IWM due to their distributed allocation behavior.

4) Benchmark *gobmk's* allocation and free behavior could benefit from IWB and IWM but is limited by the overall light use of dynamic memory in the program.

5) Benchmark *h264* has an even distribution of free and memory allocation, suggesting that it would benefit well from IWB and IWM.

6) Benchmark *hmmer* also has a reasonably distributed allocation and free pattern which suggests it could benefit from IWM and IWB.

7) Benchmark *lbm* releases its dynamic memory only at the very end making it impossible to benefit from IWB.

8) Benchmark *milc* has a moderate chance to benefit from IWM and can also benefit from IWB with the right cache size.

9) Benchmark *namd* gives up all its dynamic allocated memory only at the very end making it impossible to benefit from IWB. The lump sum allocation of memory at the start also makes it hard for *namd* to benefit from IWM.

10) Benchmark *perlbench* has a strong prospect of benefiting from IWB and IWM.

11) The memory deallocation pattern of *povray* suggests that it is a very good candidate to benefit from IWB.

12) Benchmark *soplex* seems to be a good candidate for IWM but not so much for IWB.

13) Benchmarks *xalancbmk, sphinx* and *omnetpp* are promising candidates for IWB.

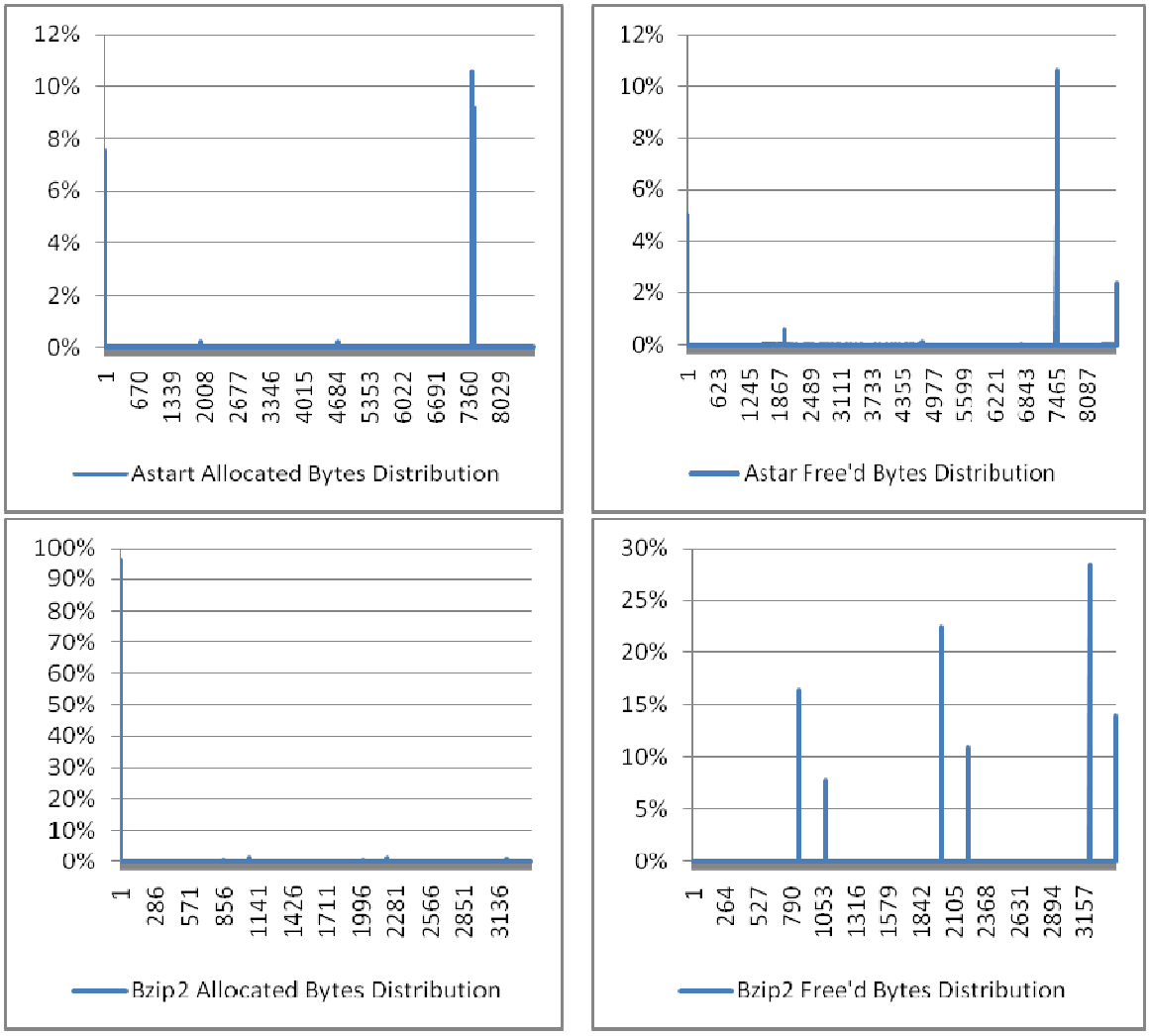


Figure 6.1 (a) Byte allocation and deallocation distribution timeline for *astar* and *bzip2*

Figure 6.1 (a) clearly illustrates that the benchmark astar has a spike in allocation and free very early on followed by a relatively steady state behavior. The early spikes

followed by steady allocation rate suggest that *astar* is a potential candidate for IWM. Its early spike and steady data deallocation rate suggest that *astar* has potential for IWM. The spikes of free operations in the range of 4%, 11% etc suggest that *astar* requires a last level cache size large enough to capture the memory free operations. If the cache is not large enough, potential candidate lines could be evicted without benefiting from IWB.

The bottom part of Figure 6.1 (a) shows the pattern for the *bzip2* benchmark. The *Bzip2* benchmark allocates almost all its dynamic data early on in the program lifetime and sporadically clears it up via free operations. One can see spikes of data free at 79 billion, 105 billion, 190 billion, 220 billion etc. This could suggest that *bzip2* has good potential for early identification of freshly allocated dynamic memory. From Table 6.3 we can see that almost all of the 0.63 GB of allocated memory has a granularity larger than 16 MB. Even though these signs are positive, the total freed dynamic memory is only 25 MB (Table 6.1) suggests that *bzip2* might not benefit from IWB but it could benefit from IWM. The allocation to *mmap* ratio of 1 in Table 6.5 demonstrates that *bzip2* needs as much heap space as the total dynamic memory allocated because it recycles very little data. This concurs with the time line behavior in Figure 6.1 (a).

Figure 6.1 (b) presents the byte allocation and deallocation behavior for *dealII* and *gcc*. In both these benchmarks we see that the allocation as well as free behavior is fairly distributed over time. In the case of *dealII,* there are a lot more spikes in its allocation and free behavior while *gcc* is mostly uniform except in the middle where it has some spikes. This suggests that both these benchmarks are good candidates to show positive impact for both IWB and IWM optimization. The *gcc* benchmark has a very high allocation to *mmap* ratio (332) in Table 6.5 which is supported well by the very distributed allocation pattern exhibited in the time line chart here.

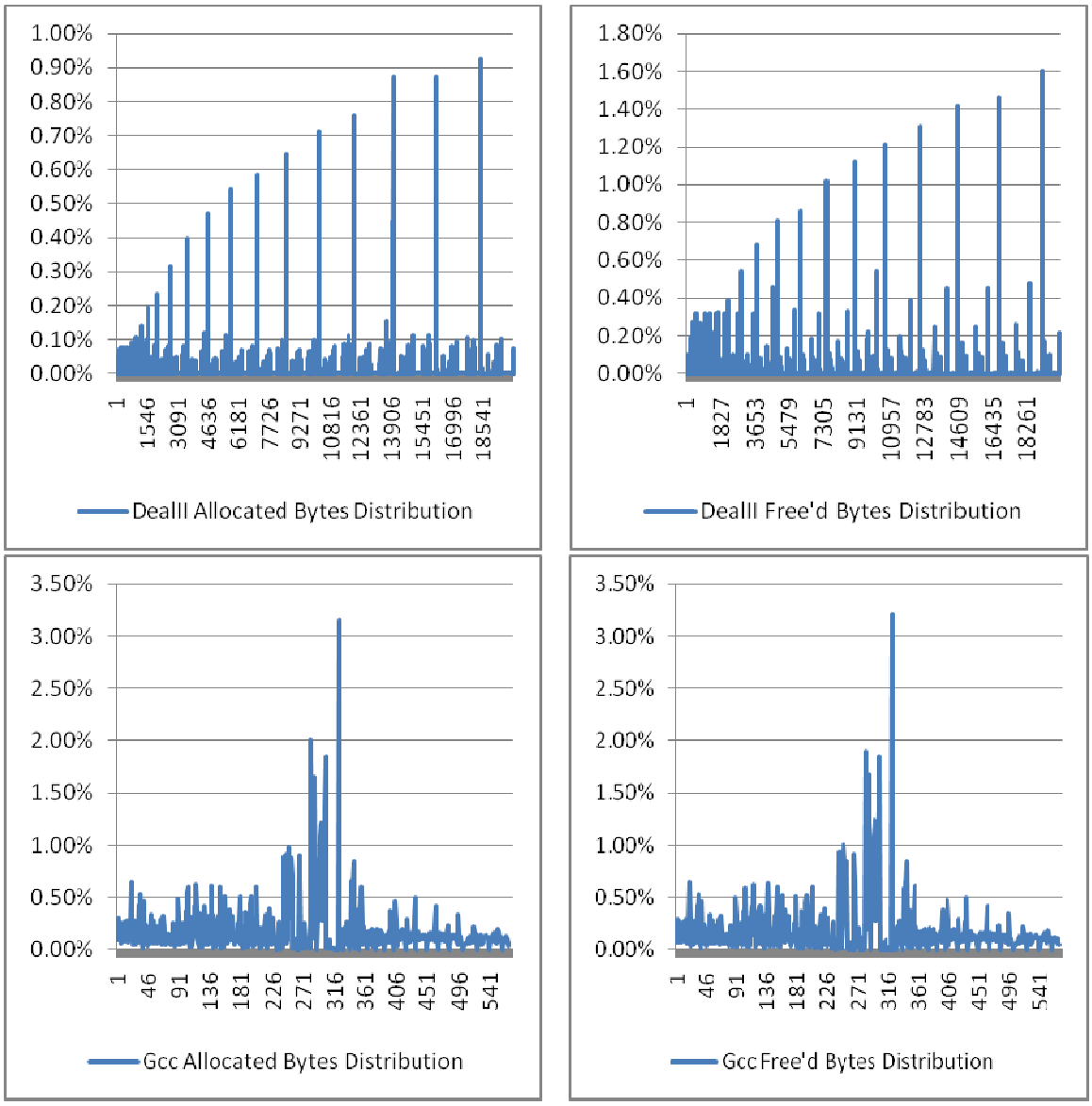


Figure 6.1 (b) Byte allocation and deallocation distribution timeline for *dealII* and *gcc*

The top section of Figure 6.1 (c) presents the behavior of *gobmk*. It has a high spike (18%) in allocation very early on followed by a slow steady state behavior. Its data deallocation behavior is quite the opposite to this, with a fairly distributed rate of free operations. Both these factors could play well for *gobmk* with respect to IWB and IWM, but the light use of dynamic memory by *gobmk* reduces suggests a reduced impact for its

characteristics. Tables 6.2 and 6.4 show the light use of dynamic memory employed by *gobmk*; where 123 MB of total dynamic allocation and 93 MB of dynamic memory freed are being tracked. The *h264ref* benchmark has a fairly distributed behavior for memory allocation and data free with an exception at the very end of the lifetime when a larger chunk (6%) of dynamic memory is freed. The evenly distributed nature of data free suggests that *h264ref* could be a good candidate for both IWB and IWM.

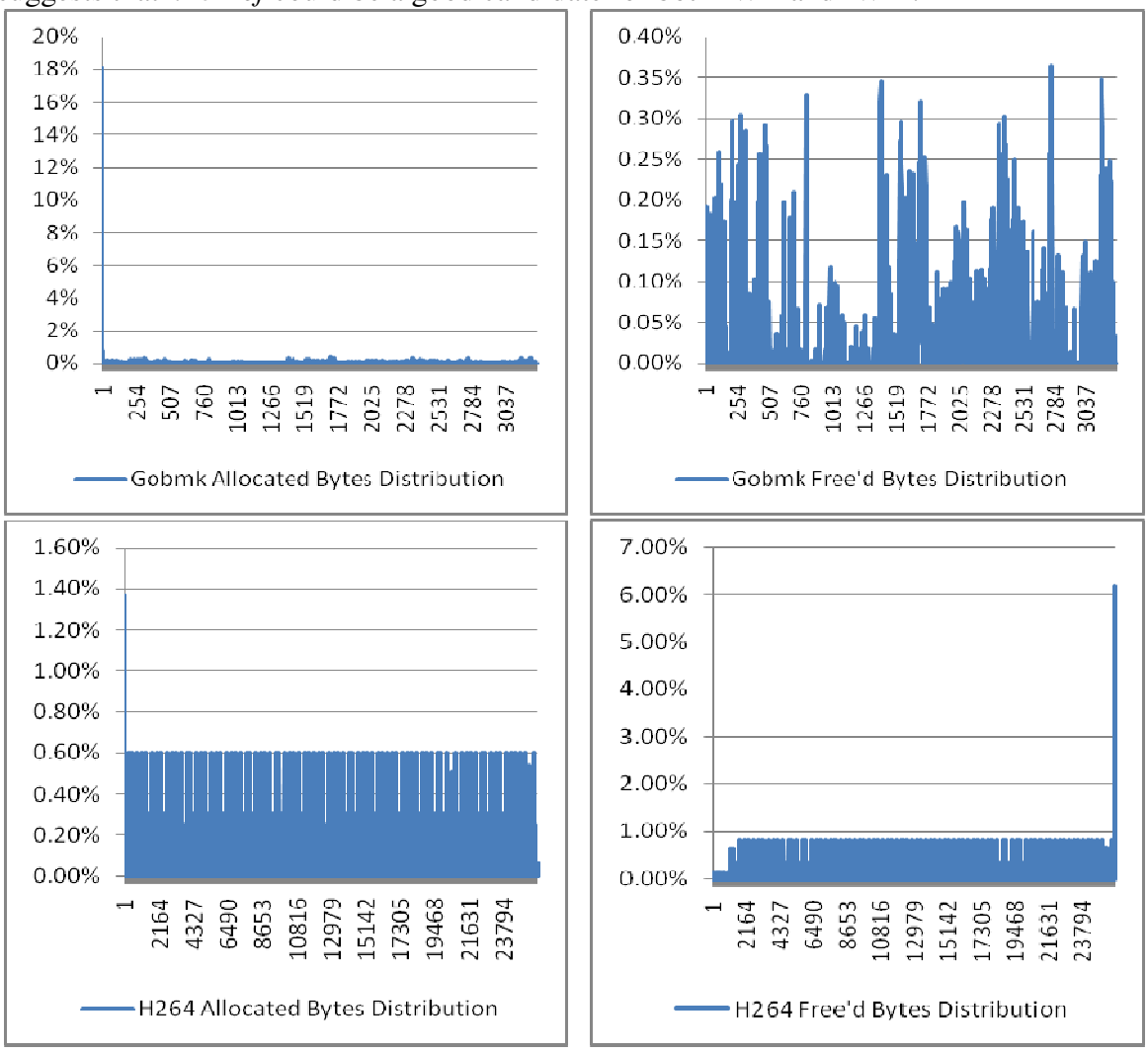


Figure 6.1 (c) Byte allocation and deallocation distribution timeline for *gobmk* and *h264ref*

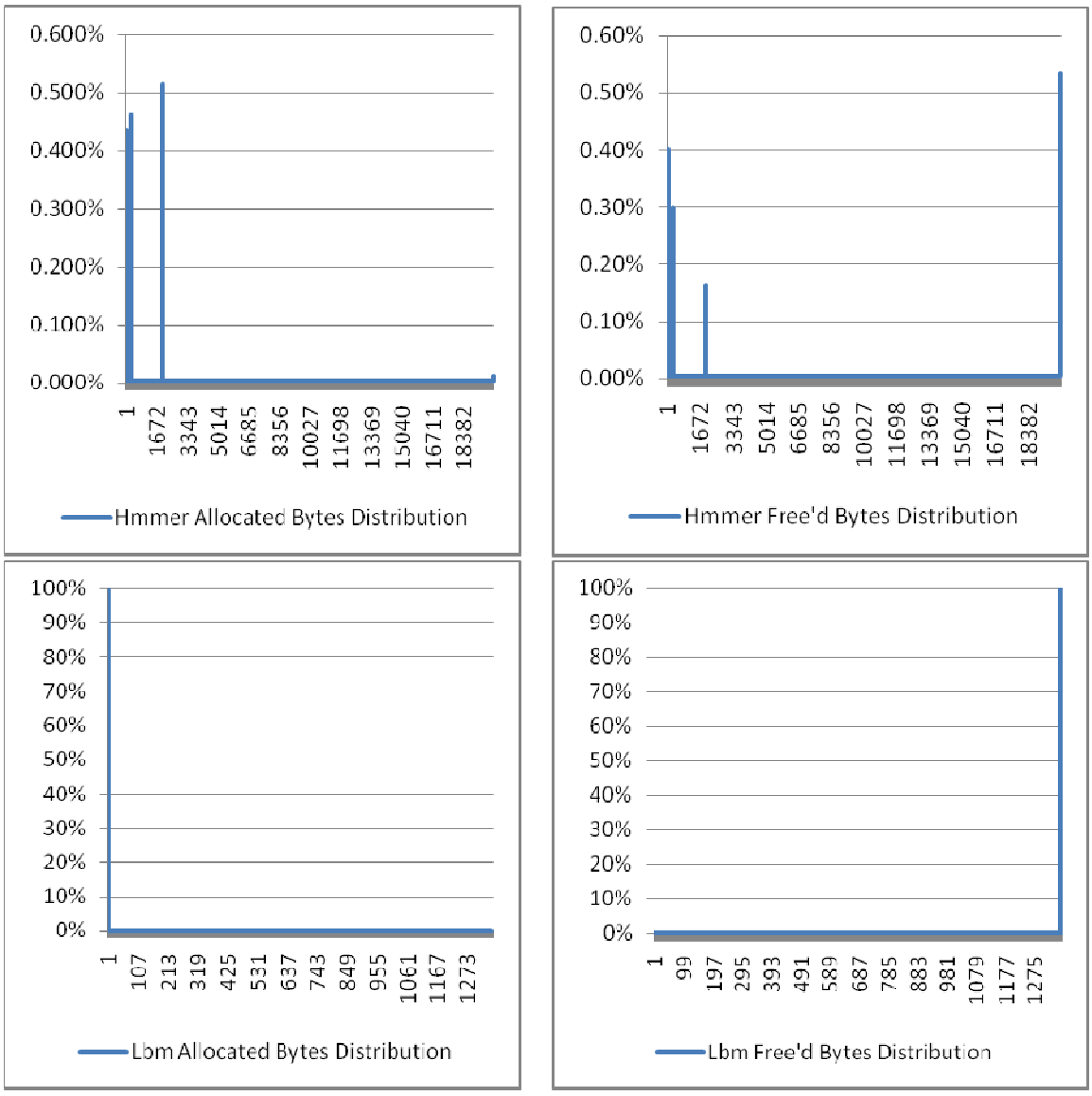


Figure 6.1 (d) Byte allocation and deallocation distribution timeline for *hmmer* and *lbm*

The *hmmer* benchmark has a non uniform behavior of allocation and free without being too biased to the beginning and end of the program lifetime. It allocated data at a very light rate and evenly for most of the program lifetime except for a few points early on. This suggests that it will be a good candidate for IWM early on and possibly for the rest of its life. Its data deallocation operations also have very light distributed rate with 3

major spike points, two early on and one at the very end of the program life. Since only 0.5% of the memory is freed at the very end, it suggests that there is a reasonable chance for finding IWB candidates throughout the program lifetime unlike benchmarks which free most of its memory at the very end such as *lbm* (bottom portion of Figure 6.1 (d)). The bias in allocation and free for *lbm* (all of the allocation at the start and all of its data free operation at the end of the life time *lbm)* makes *lbm* unappealing for IWB and IWM.

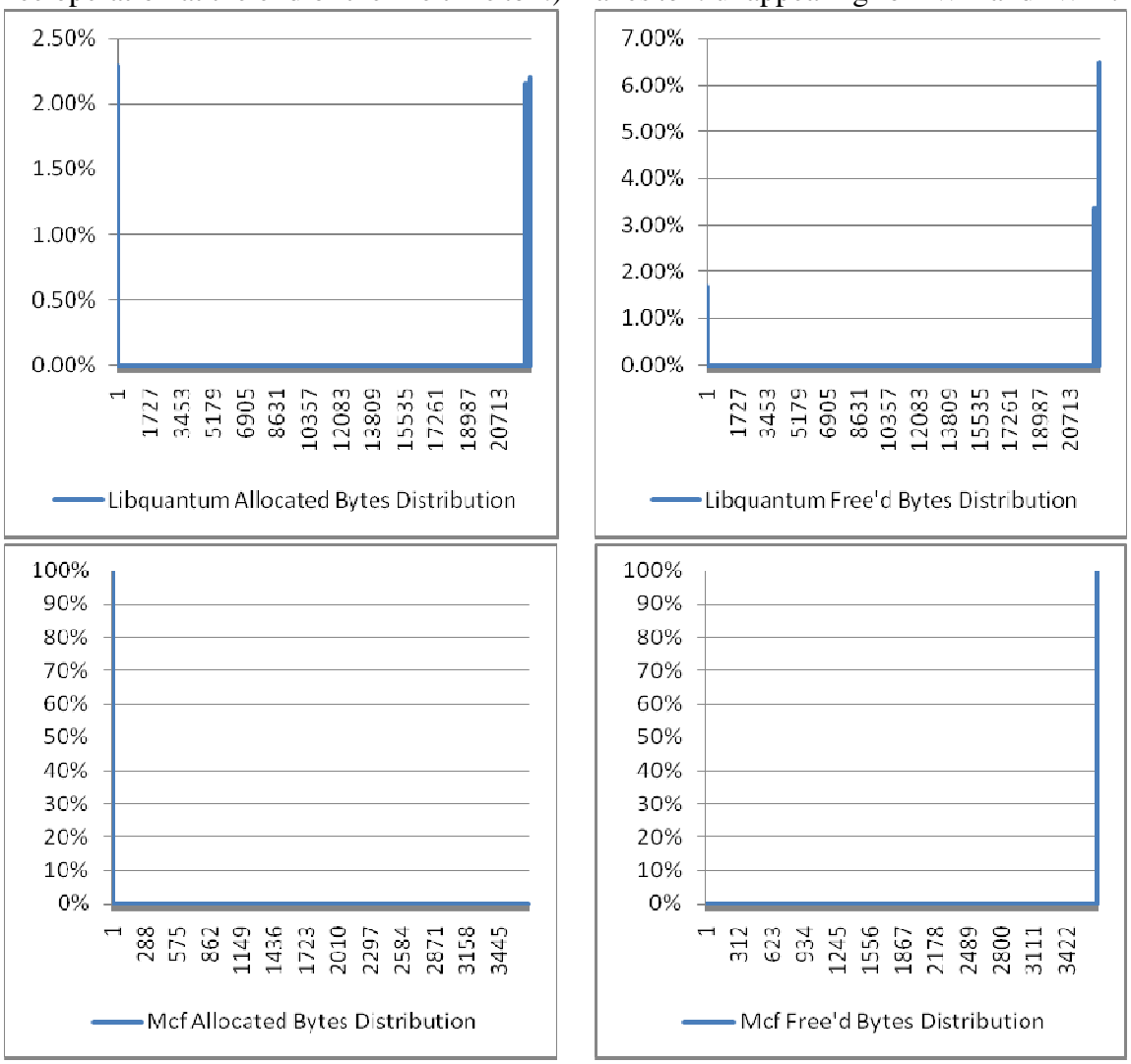


Figure 6.1 (e) Byte allocation and deallocation distribution timeline for *libquantum* and *mcf*

Both *libquantum* and *mcf* are presented in Figure 6.1 (e). For *mcf*, almost all of its allocation is done early on and most of its free operations performed at the end (similar to *lbm*). This makes them very bad candidates for IWB because all the free operations happen at the end, i.e., at a point which is too late to be of use. Both these benchmarks allocate large chunks of memory in granularities too large to even fit in last level caches (1 MB to 16 MB and larger). Most of the miss might arise from capacity miss which makes the effect of IWM harder.

In Figure 6.1 (f), the top portion represents *milc* benchmark which has a small allocation spike of 0.6% at the start but in general has a relatively even pattern of allocation through its life. This makes it an average candidate for IWM. Its memory free operations on the other hand are very evenly distributed. This does raise the expectation of *milc* to perform well for IWB provided the cache can capture and hold IWB candidates long enough to be identified as IWB. Given the large granularity of its object sizes (1 MB - 16 MB and larger) this could be hard to achieve in last level caches (LLC). Benchmark *namd* on the other hand has a pattern we saw in the case of both *mcf* and *libquantum*. From Table 6.2 we do see that smaller but reasonable portion of its dynamic memory belongs to granularities that are small enough to fit in LLC. Thus, *namd* can be expected to have a reasonable positive contribution for IWM. The same does not hold true for IWB because of the focus of all free operations at the end of the program lifetime.

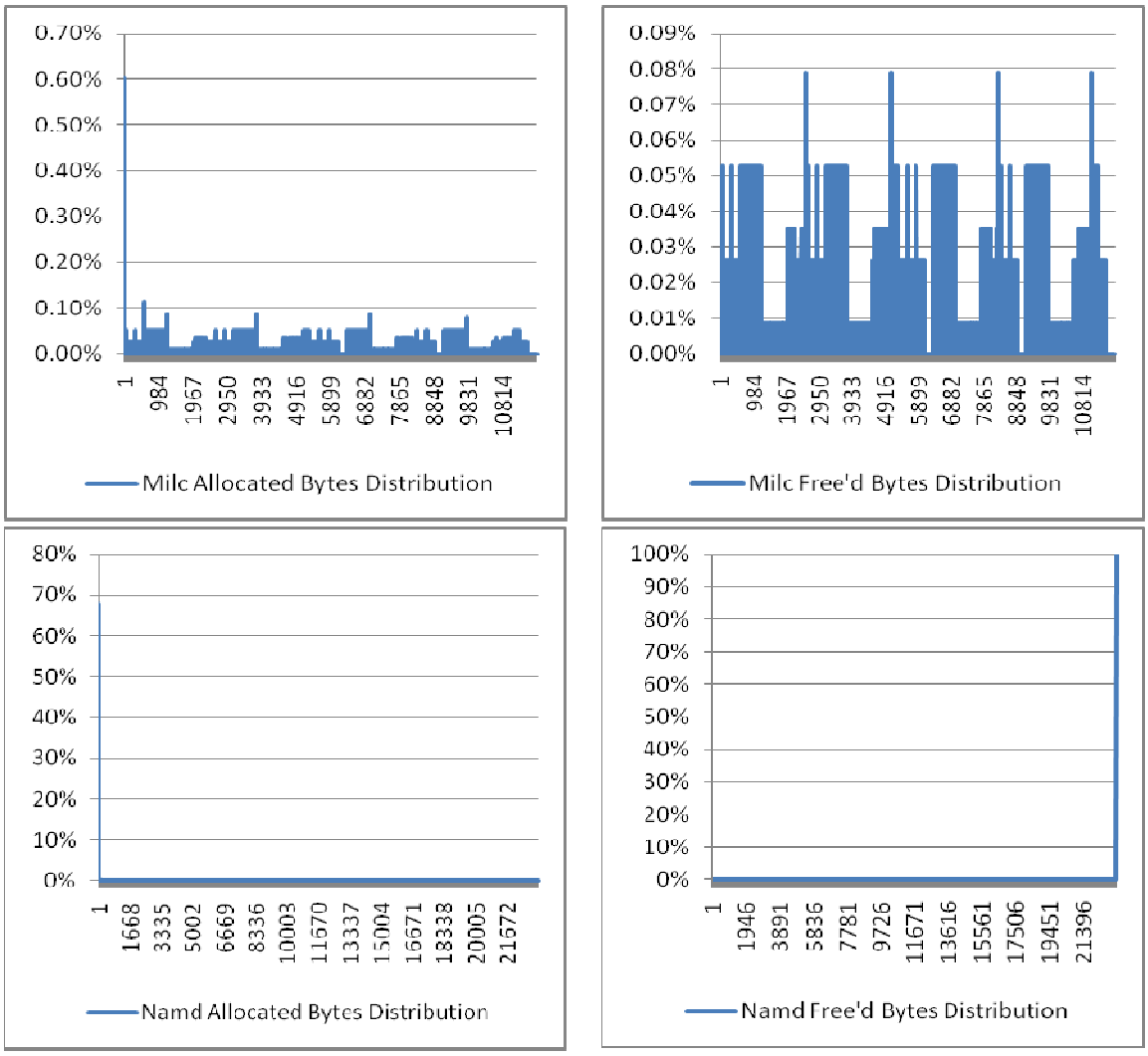


Figure 6.1 (f) Byte allocation and deallocation distribution timeline for *milc* and *namd*

Figure 6.1 (g) presents the dynamic memory consumption pattern for *omnetpp* and *perlbench*. Both these benchmarks have very dynamic and fairly uniform distribution of allocation and data free operations. In the case of *omnetpp,* the dynamic memory behavior seems to be very scattered with allocation and deallocation operations at each point contributing only a very small portion of the total i.e., in the 0.02% range. This might be correlated to the fact that most of the dynamic memory allocations have a size

granularity (Table 6.2 and Table 6.4) in the 64 byte to 256 byte range. Thus, it does not give us a clear expectation on the effects of IWB and IWM. On the other hand, *perlbench* has granularities of allocation and free operations (Table 6.2 and 6.4) that are in the 4 KB to 256 KB as well as the 256 KB to 1 MB range. Each point tends to allocate and free a larger amount of the total dynamic memory (0.1% range) compared to *omnetpp*. This, combined with the strong dynamic memory usage, could make *perlbench* a good candidate for IWB and IWM.

The top portion of Figure 6.1 (h) has the dynamic memory pattern for *povray*. There is a large spike in allocation (35%) early in the life of *povray* with the rest of the pattern being flat and even. Its deallocation operations also have a similar spike at the start with roughly 15% of the free operations occurring very early on followed by a flat even distribution until the end where a small spike in data deallocation occurs. The early spike in memory allocation plays well for IWM because it permits early identification of allocated memory which is consumed over time. The early spike in memory deallocation means that, as the execution progresses, *povray* has a good chance that dirty data in the cache is identified as IWB candidates. Future instructions could evict and cause write backs which could be saved by IWB, thus making IWB based gains more likely for *povray*.

The bottom portion of Figure 6.1 (h) shows the dynamic memory behavior of *sjeng*. Its behavior is very similar to that of *mcf, libquantum, namd* etc; i.e., heavy allocation at the start (almost 100%) and heavy amount of free operation at the very end. This could play well for IWM optimization since most of the allocation is very early on and this dynamic memory gets consumed over time. Table 6.2 shows that *sjeng* has a heavy bias in its allocation granularity to data sets larger than 16 MB. This could make it very hard for data to fit in LLC and hence could shadow the impact of IWM unless the

memory hierarchy can encompass the data requested. Since almost all the memory free operations for *sjeng* happen at the very end, it occurs too late to be of use in identifying useful IWB candidates that are still present in the memory hierarchy.

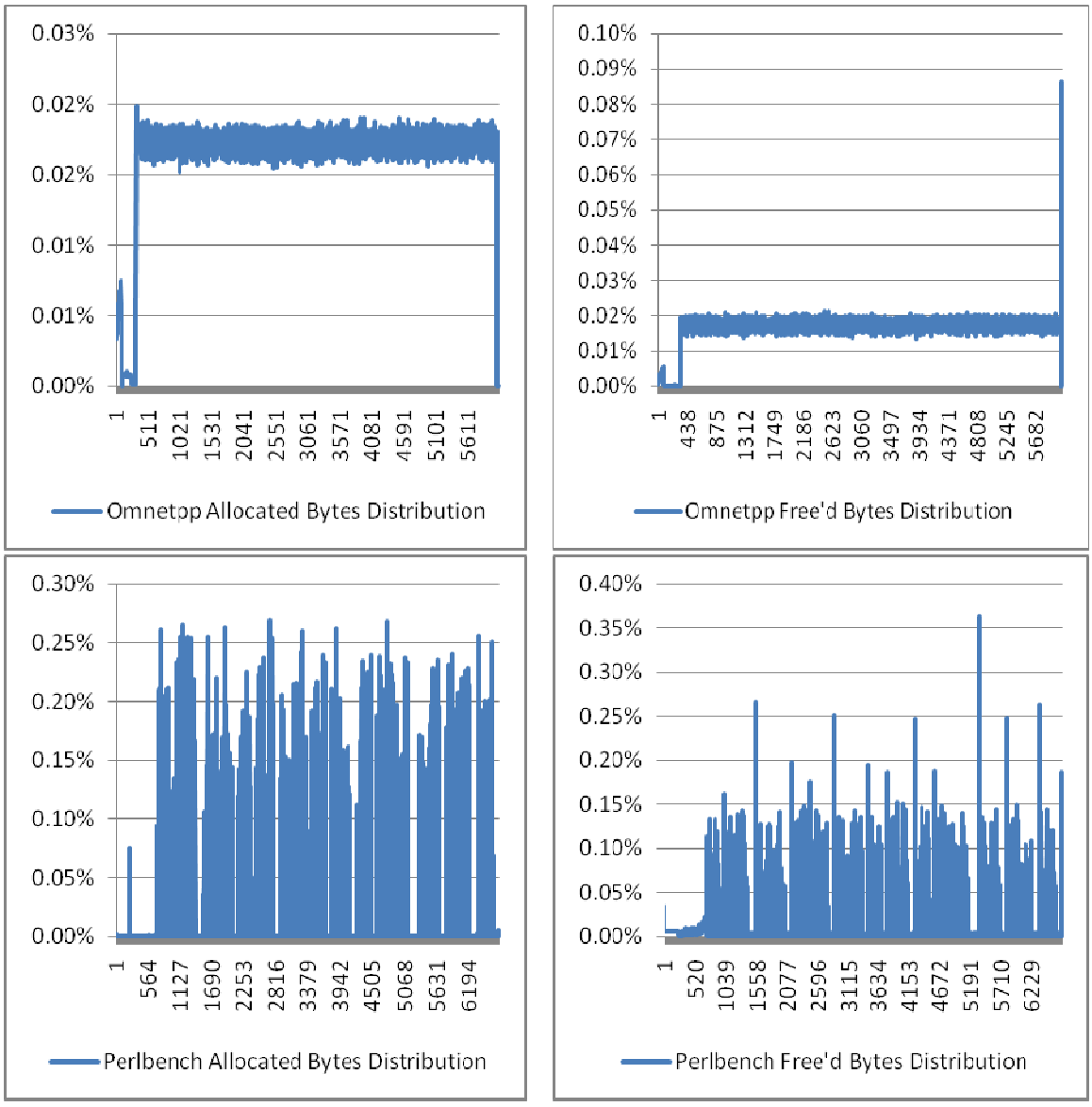


Figure 6.1 (g) Byte allocation and deallocation distribution timeline for *omnetpp* and *perlbench*

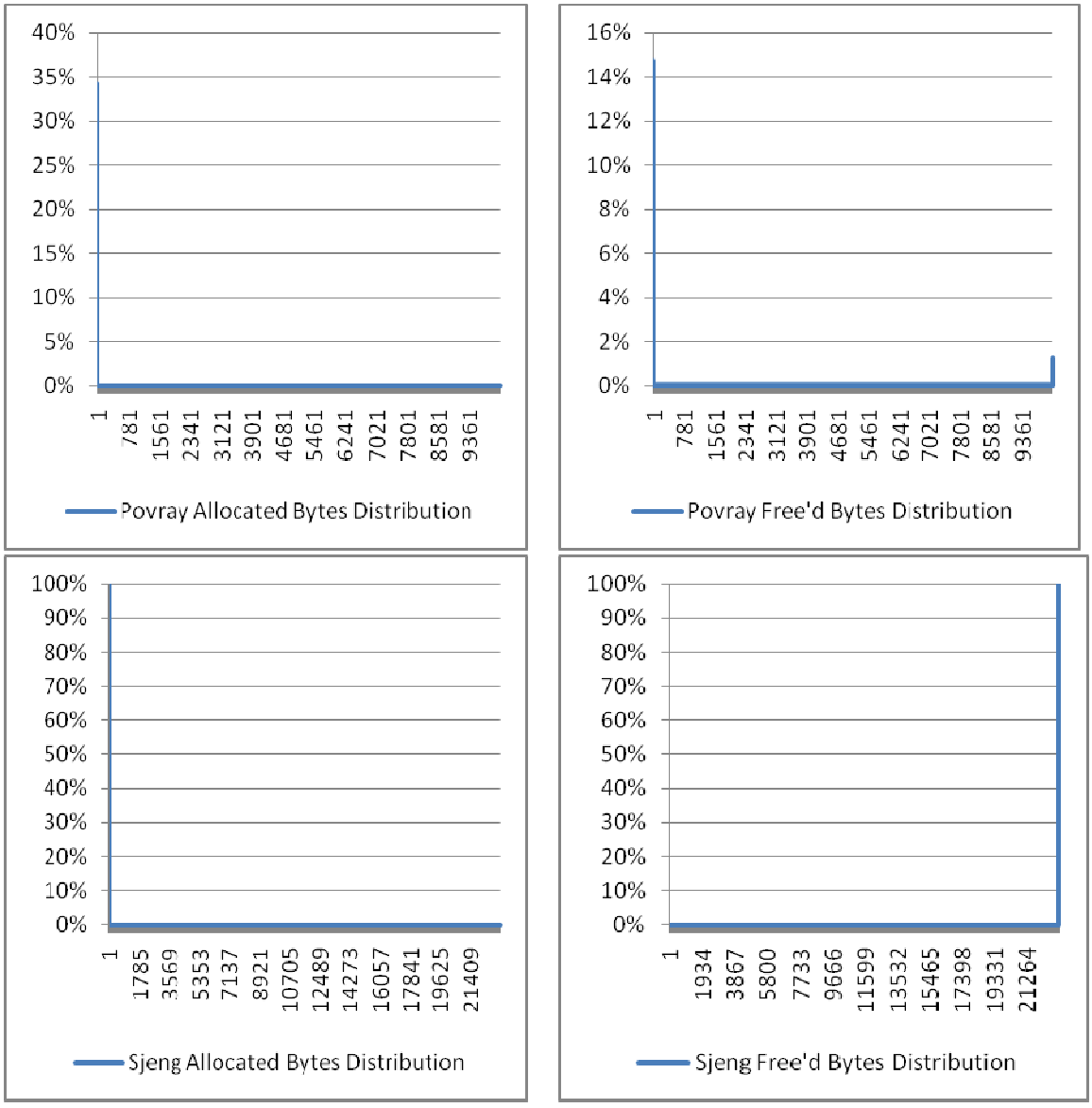


Figure 6.1 (h) Byte allocation and deallocation distribution timeline for *povray* and *sjeng*

Figure 6.1 (i) Byte allocation and deallocation distribution timeline for *soplex* and *sphinx3*

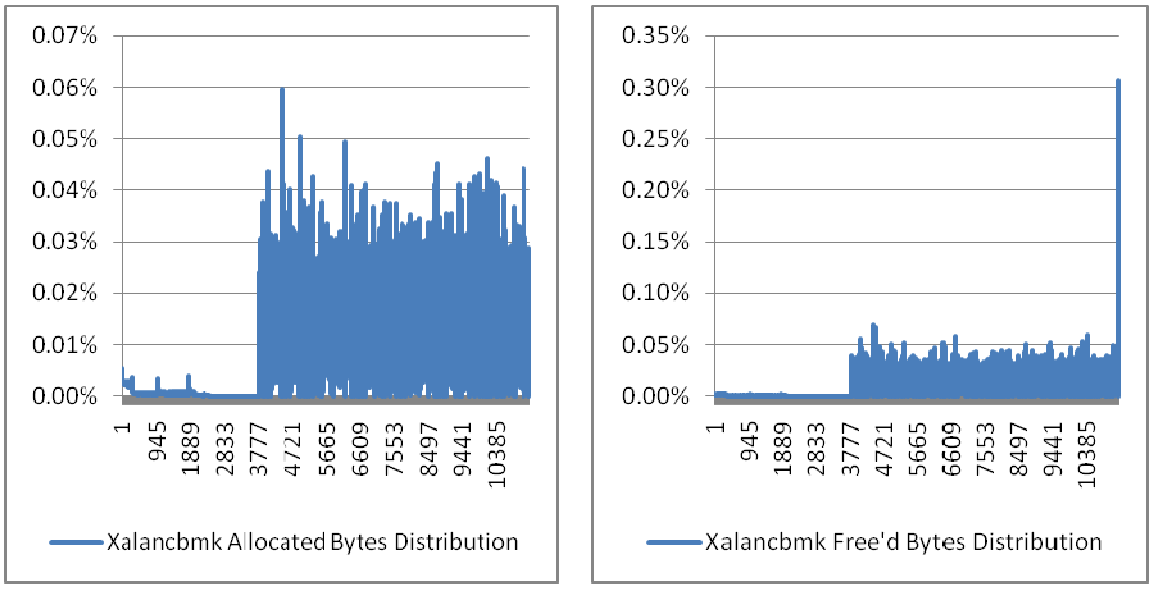


Figure 6.1 (j) Byte allocation and deallocation distribution timeline for *xalancbmk*

The allocation pattern of *soplex* (top-left of Figure 6.1 (i)) shows strong spikes very early on followed by low and steady activity through the rest of its life. This makes it a very good candidate for IWM. On the other hand, *soplex* deallocates more than 50% of its data at the very end, which could make it harder for it to do well with IWB. The small spikes in data free operation early on (10%) can still give some benefit to IWB, provided that the memory hierarchy being considered can encompass the object sizes used by *soplex*; object sizes mostly larger than 16 MB and many others in the 1 MB to 16 MB range. The patterns for *sphinx* (Figure 6.1 (i)) on the other hand are very uniform which suggests a mixed expectation for impact for IWM and a better impact for IWB. Since the distribution of allocation and free seems to be very fragmented and scattered over time, the impact on IWB and IWM due to its temporal nature is hard to estimate.

In Figure 6.1 (j), we have the dynamic memory consumption and release pattern for *xalancbmk*. This program makes heavy use of dynamic memory, but its allocation patterns as well as data free operations pick up almost at the 30% mark. This constant

activity from the 30% mark with dynamic memory object granularities in the range of 1 KB to 4 KB makes *xalancbmk* a possible candidate but it is harder to estimate its impact on IWB and IWM.

The dynamic memory behaviors of these benchmarks are very varied and diverse. This gives strength to the use of these benchmarks for the studies in this dissertation. More importantly, the behavior of these benchmarks gives some insight into the scope for IWB and IWM for each of them.

6.3 INVALIDATION HIT RATE

In this section, this dissertation looks into the amount of free calls and freed blocks that are present in the cache. It is of interest to find out what percentage of the memory free operations find the address it frees in the cache hierarchy. The more the free function call finds the freed address in the cache the more are the chances for it to be a future candidate for an IWB or a candidate for replacement. The last level cache here is a 2 MB L2, the base line used in this dissertation. Table 6.10 presents the free line hit rate, defined as the percentage of "free" function calls that find at least one cache line worth of freed block in the cache. Table 6.10 also presents the free block hit rate, defined as the percentage calls whose whole freed block is found in the L2 cache. The same data is also collected in Table 6.10 for munmap operations. In almost all cases, we find at least one block from the freed area is still residing in the L2 cache. The chances of the whole block residing in the cache are significantly lower.

On average only 50% of the free calls find its freed data block in the L2 cache. In the case of munmap operation this falls to 35.52%. Benchmarks such as *gcc, hmmer, perlbench, povray* and *sphinx3* have 90% or more of the freed blocks present in the L2 cache. This can translate into a good amount of IWB provided dirty data caused by

dynamic memory is a significant percentage of the dirty data in the cache. The effect of IWB can be lower if the freed block gets reused by a future allocation, which is likely given that memory allocators typically work on a LIFO (last in first out) policy for reuse of dynamic memory blocks.

	Memory Deallocation (*Free*)		Large Scale Deallocation (*Munmap*)	
Benchmark	Line Hit	Block Hit	Line Hit	Block Hit
astar	~100	25.6	~100	62.3
bzip2	~100	76.5	~100	82.1
dealII	~100	63.4	~100	5.8
gcc	~100	89.1	~100	37.9
gobmk	~100	56.4	~100	30.8
h264ref	~100	20.7	~100	0.6
hmmer	~100	~100	~100	1.4
lbm	~100	2	~100	2
libquantum	~100	15.8	~100	13.5
mcf	~100	0	~100	0.5
milc	~100	26.8	~100	27.9
namd	~100	23.1	~100	8.4
omnetpp	~100	78.5	~100	93.8
perlbench	~100	96.6	~100	92.8
povray	~100	94.9	~100	37.6
sjeng	~100	0	~100	0.8
soplex	~100	17.4	~100	1.4
sphinx3	~100	92.8	~100	87
xalancbmk	~100	75.7	~100	88.3
AVG	~100	50.28	~100	35.52

Table 6.10 Invalidation call hit rate in L2 cache

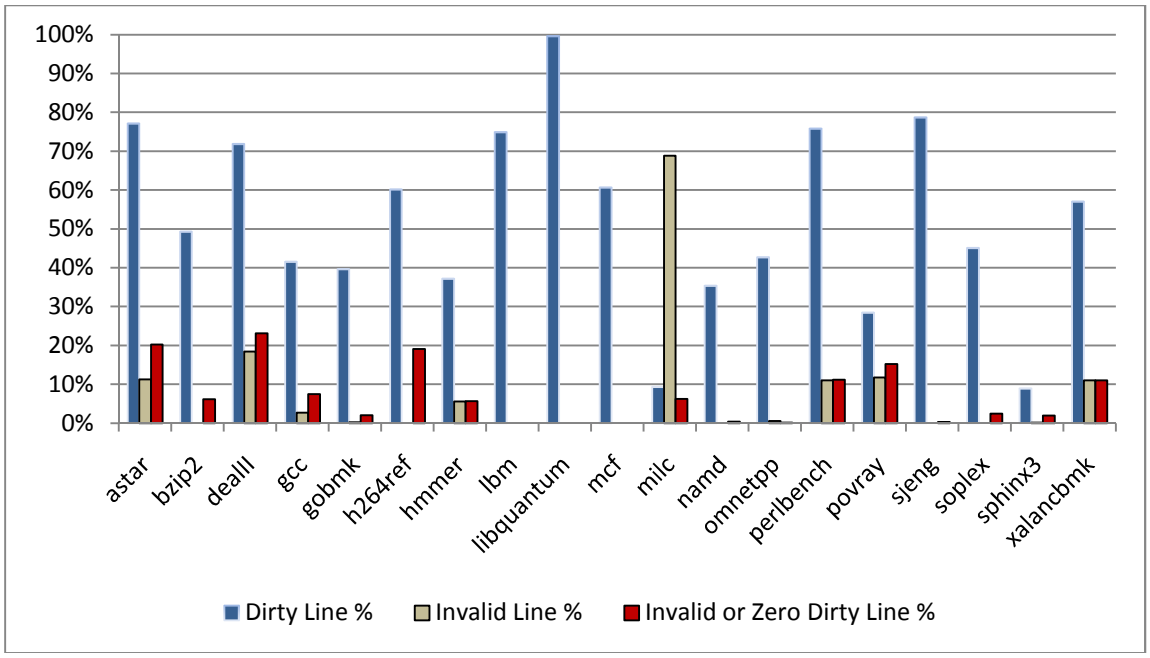


Figure 6.2 Last level cache occupancy based on snapshot at end of simulation - % of dirty line, % invalid lines and % lines that are both dirty and invalid

6.4 Last Level Cache Occupancy

The previous sections studied the benchmarks memory management behavior to understand the allocation and free pattern and estimate the benchmarks that are more likely to benefit from IWB and IWM. In this section, this dissertation analyzes the actual last level cache occupancy. The data in Figure 6.2 presents the occupancy level of the L2 cache measured at the end of the simulation used for energy estimation i.e., the benchmarks were run for the simulation length used to estimate energy in the next chapter i.e., 10% of the execution length. Hence, this is a snapshot of the cache state. The first bar represents the percentage of the total cache lines that are currently in dirty state. The snapshot of cache occupancy at the end of simulation is an indication of future impending write back for that benchmark. Figure 6.2 shows that most of the benchmarks

have about 40% of the cache lines in dirty state. A few clear exceptions are *sphinx3* and *milc* which has less than 10% of the cache lines in dirty state. What is of more interest is the amount of cache lines that are dirty but are also invalid or are dirty and contain zero value data. Both of these are indicators of the potential for IWB and ZVS in those benchmarks. The percentage of cache lines that are in the dirty and invalid or zero data state for *astar, bzip2, dealII, gcc, h264ref, hmmer, milc, perlbench, povray, soplex* and *xalncbmk* demonstrate signs of having potential for IWB and ZVS optimizations.

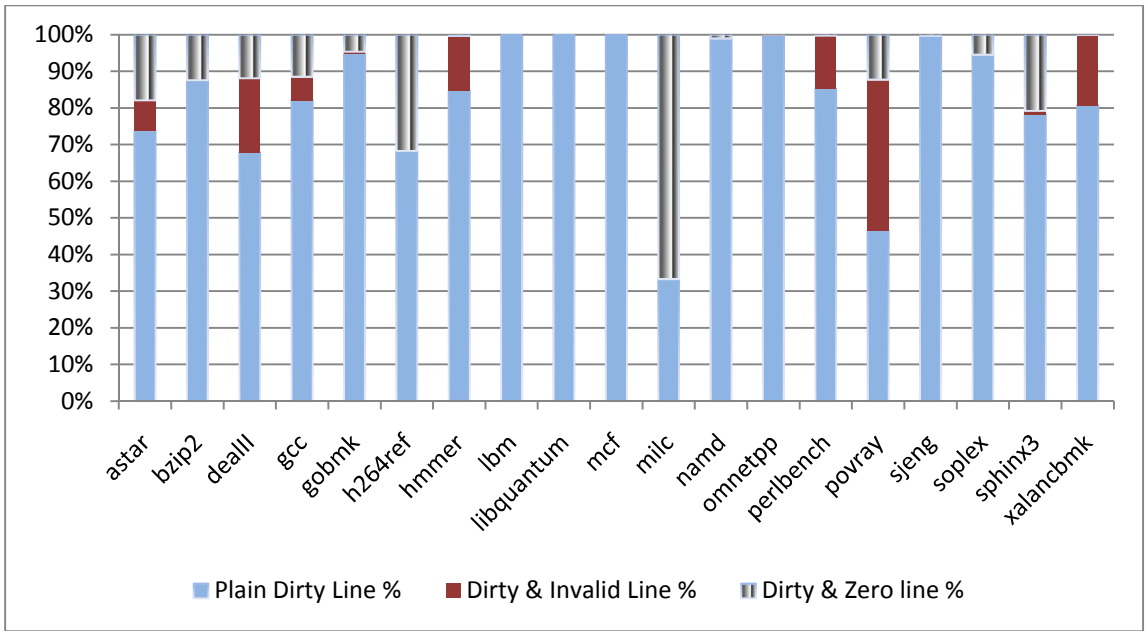


Figure 6.3 Last level cache dirty line occupancy composition based on snapshot at end of simulation

Focusing on the write backs, Figure 6.3 shows the split of the dirty lines i.e., the percentage of dirty lines that are invalid and the percentage of dirty lines that contain zero data. Here we see that *astar, dealII, gcc, hmmer, perlbench, povray* and *xalancbmk* contain dirty lines that have been marked as invalid i.e., inconsequential. Benchmark *povray* has the maximum amount of such cache lines, while *dealII, hmmer, perlbench*

and *xalancbmk* have 15-20% of the dirty lines in inconsequential state because they have been marked invalid by either a free or *munmap* operation.

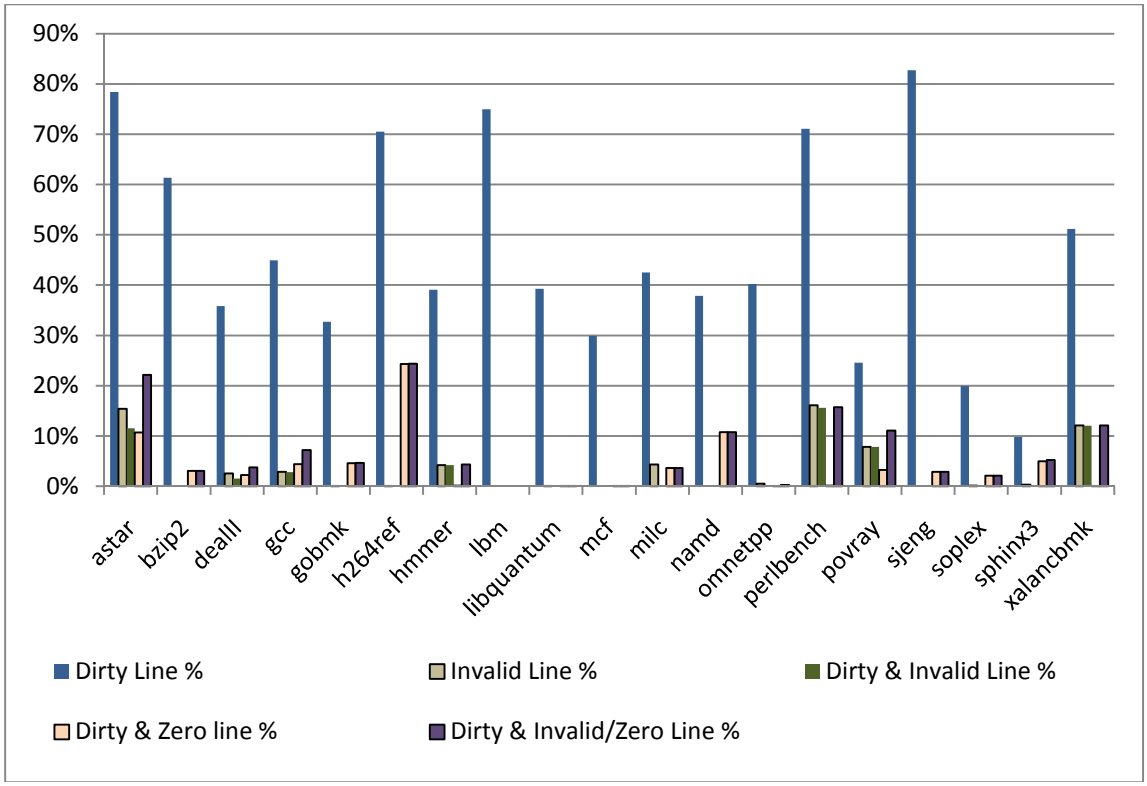


Figure 6.4 Last level cache occupancy composition based on average behavior

The data in the previous charts represent only a snapshot of time. In the interest of better coverage and better understanding the true nature of these benchmarks, this dissertation inspects the benchmarks over its whole length and records the average occupancy level. Figures 6.4 and 6.5 are counterparts to Figures 6.2 and 6.3 but present the average program behavior sampled at a million instruction interval. The average cache occupancy for the amount of lines that are dirty stays roughly the same for most benchmarks. The benchmarks that deviate significantly i.e., more than 10%, are *dealII, libquantum, mcf* and *milc*. Except for *milc* the amount of dirty lines present in the whole

run reduces. Although this does not directly correspond to the amount of invalid and dirty lines as well as amount of write backs, it does reduce the amount of dirty lines that could be candidates for identification as inconsequential. It also means that, for these benchmarks, the dirty lines decrease over time, possibly because more of its data writes are concentrated earlier in its execution. For certain other benchmarks such as *bzip2*, *h264rerf* and *milc* the trend is reversed i.e., the amount of dirty lines increase over time suggesting that more of the data writes happen in the latter half of the program execution.

In Figure 6.5 the composition of dirty cache lines, i.e., the split between invalid and zero lines that are dirty, is presented. It is encouraging to see that both *dealII* and *povray* have increased the percentage of the dirty lines that are marked as invalid and hence inconsequential compared to the observation based on the snapshot shown in Figure 6.3. In the case of *milc*, which had an increase in the percentage of dirty lines for the full program run, the amount of zero data dirty lines decrease, thereby reducing the scope of ZVS gained due to the increase in the percentage of dirty lines. In general it is observed that *astar, dealII, gcc, hmmer, perlbench, povray* and *xalancbmk* are good candidates for IWB. Benchmarks *astar, bzip2, dealII, gcc, gobmk, h264ref, milc, povray, soplex* and *sphinx3* are good candidates for ZVS.

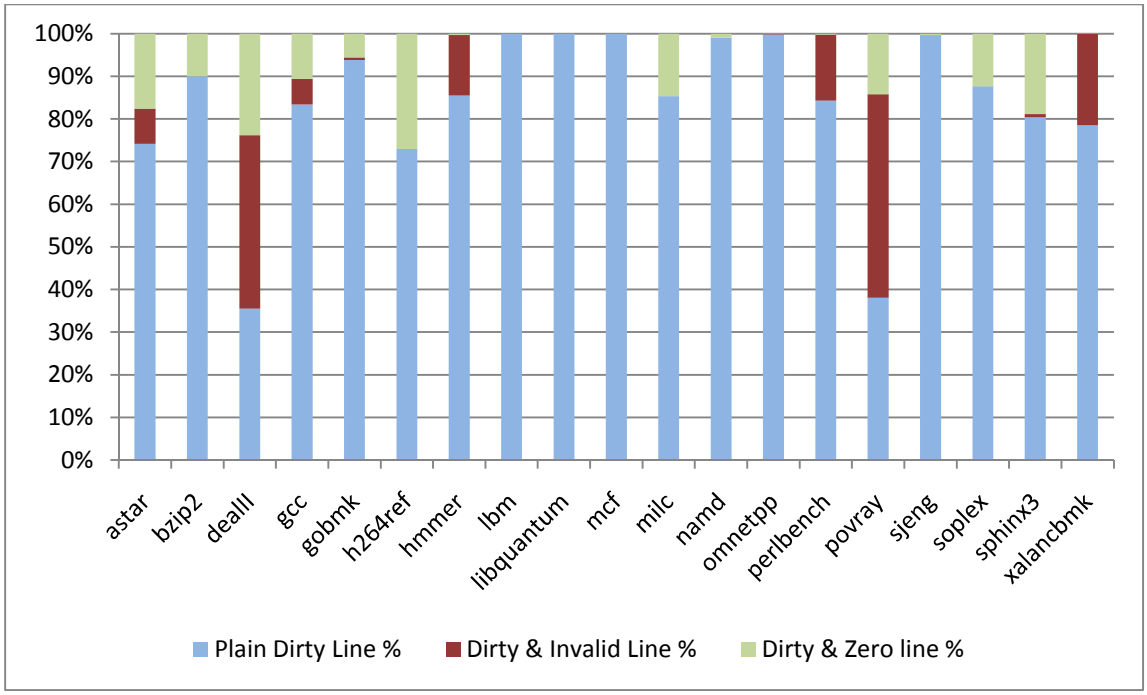


Figure 6.5 Last Level cache dirty line occupancy composition based on average behavior

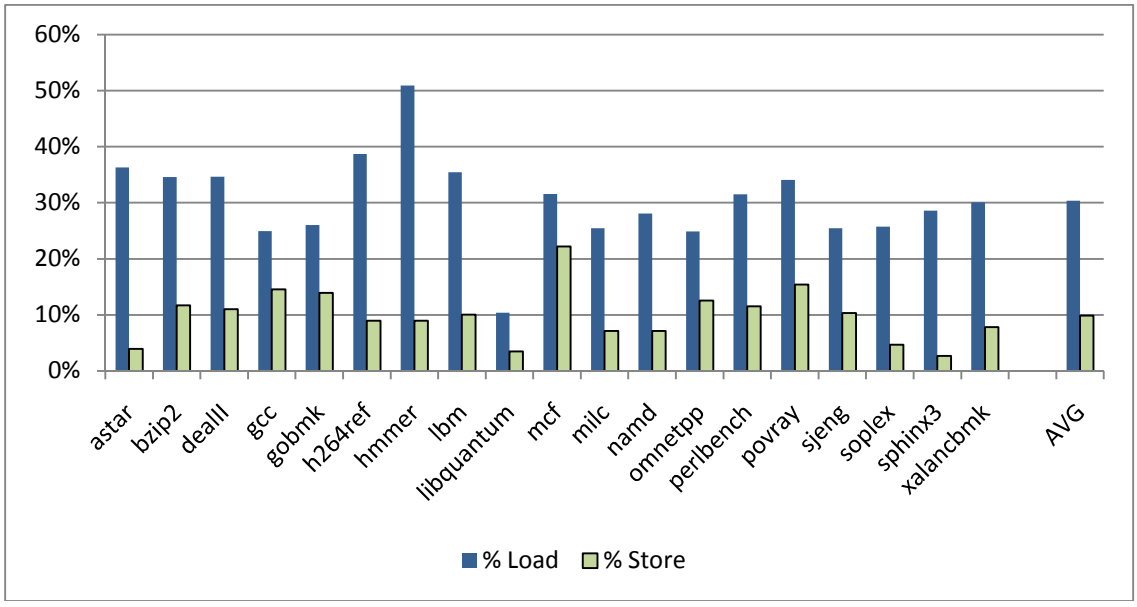


Figure 6.6 Load-Store instructions as a percentage of total instruction

6.5 ZERO VALUE LOAD/STORE DISTRIBUTION

This section inspects these benchmarks to better understand their data value characteristics. Figure 6.6 is a representation of the load-store instruction mix. As expected, loads are more common than stores. On an average loads are three times more common that stores. In benchmarks such as *gcc, gobmk, mcf, omnetpp* and *povray* have a higher amount of stores; about half as much stores as loads.

In Figure 6.7 all the stores issued by the benchmarks are analyzed and the percentages of the stores that store zero value are presented based on the granularity. There are four granularities that are recorded. From left to right they are: (1) stores that have all the data in one instruction storing zero value, (2) stores that store zero value data and have the whole target cache line containing zero value, (3) stores that store zero value data and have 1 KB of aligned memory range around the target memory as zero and (4) stores that store zero value data and have 1 page (4 KB) of aligned memory range around the target memory containing all zeros. As the granularity is increased the amount of stores that meet the criterion tends to decrease. For example, *astar* has a large amount of stores which stores all of its data as zeros but at higher granularity of aligned memory ranges this percentage falls to almost zero. Benchmarks such as g*cc, gobmk, h264ref, milc, namd, omnetpp, povray, sjeng* and *sphinx3* have some amount of zero valued stores which stores to a cache line which contains zero value data. On expanding the granularity to 1 KB, only *gcc* demonstrates a significant amount of zero value stores. This in turn gives the insight that tracking zero value data ought to be done at least at a cache line to generate some reasonable impact. Thus, zero value stores exist in quite a few of the benchmarks with 10 out of the 19 benchmarks used in the study having almost 20% or more of the stores belonging to this category. On an average, 20% of the stores attempt to store zero value while about 8% of stores store zeros into a cache line which also contains

zeros.

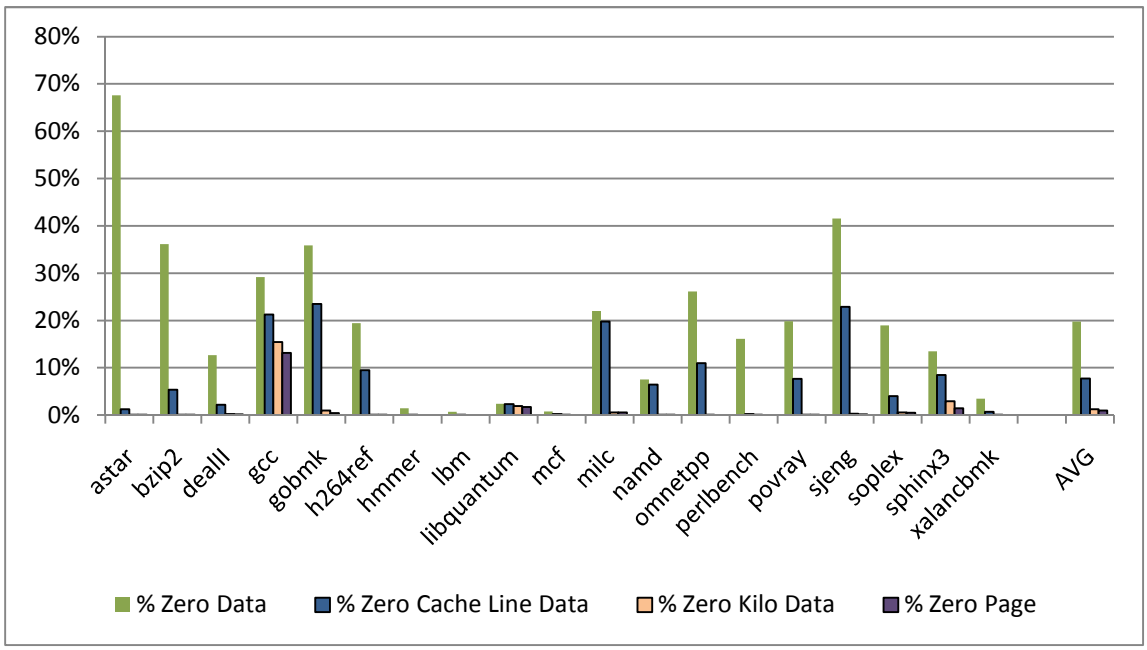


Figure 6.7 Zero data stores as percentage of total stores

In Figure 6.8 all the loads issued by the benchmarks are analyzed and the percentages of the loads that load zero value are presented based on the granularity. There are four granularities that are recorded. From left to right they are: (1) loads that have all the data in one instruction loading zero value, (2) loads that load zero value data and have the whole target cache line containing zero value, (3) loads that load zero value data and have 1 KB of aligned memory range around the target memory as zeros and (4) loads that load zero value data and have 1 page (4 KB) of aligned memory range around the target memory containing all zeros. As with stores, when the granularity is increased the amount of loads that meet the criterion tends to decrease. For example, *astar* has a large amount of loads which load all of its data as zeros. But at a higher granularity of aligned memory ranges this percentage falls to almost zero. In benchmarks such as g*cc, gobmk, milc, povray, sjeng* and *sphinx3* a noticeable amount of zero valued loads also have their target cache line containing zero. However, on expanding the granularity to 1

KB none of the benchmarks demonstrate a significant amount of zero value loads. This in turn gives the insight that tracking loads with zero value data target ought to be done at least at a cache line to generate some reasonable impact. This implies mechanisms to store this information too have to exist at a cache line granularity even if the level in the memory hierarchy data is being fetched from has a larger granularity. Unless the zero value loads target previous zero value stores, it becomes harder to identify loads that will result in zero value data loads. For example, if a memory page is the target one would need 64 bits per page to identify zero value data in a page at a cache line granularity.

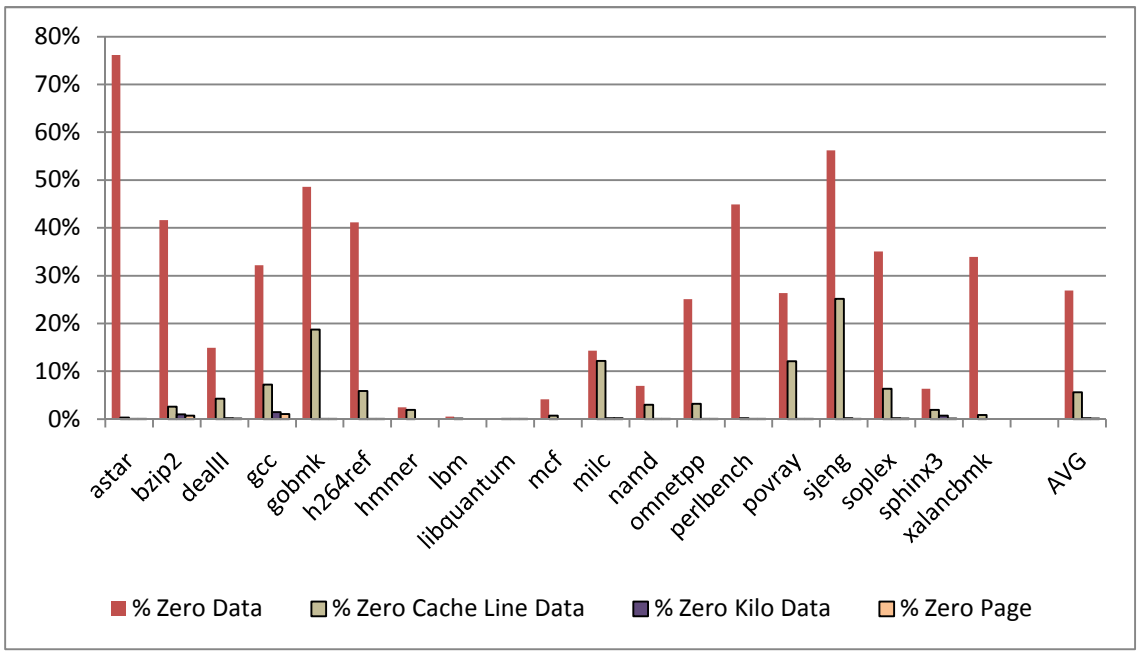


Figure 6.8 Zero data loads as a percentage of total loads

Irrespective of how IWB, IWM and ZVS are employed there are a few essential common high level aspects to its implementation.

1) One has to be able to detect the transition of normal memory into inconsequential state. This requires co-operation between the memory manager and the hardware.

2) One has to be able to store inconsequential memory ranges so that this information is available when needed. A mechanism is needed to encode information regarding inconsequential memory and allow update and retrieval of this information.
3) One has to be able to detect the transition of inconsequential memory into normal state. Detecting this transition will require co-operation between the memory manager as well as the hardware.
4) One has to be able to use the stored information for the optimizations such as IWB, IWM and ZVS.

An additional aspect involved in this determination is the granularity at which inconsequential memory information is detected and stored. The discussions related to the last few figures make the importance of granularity a lot more obvious. Larger the granularity lesser are the chances for IWB, IWM and ZVS. The next few chapters go over the details of how such mechanisms are implemented for DRAM energy savings and improving the endurance and lifetime of an emerging memory technology based memory.

Chapter 7: ESKIMO – Saving Energy in DRAM based Memory

Dynamic Random Access Memory (DRAM) is used as the main memory in most computing systems. Observations about the memory power being comparable to that of the core power in large scale systems, points to the need to focus our attention on the energy consumed by the memory subsystem. With the need to reduce the energy consumption of the DRAM subsystem of modern systems in mind, this dissertation presents ESKIMO which uses a few techniques based on memory state to save energy and power in the DRAM based memory subsystem. It leverages insights discussed in the previous chapters and use those to optimize the system architecture.

7.1 ADAPTATIONS FOR DRAM ENERGY SAVINGS

7.1.1 Semantics Aware DRAM Refresh

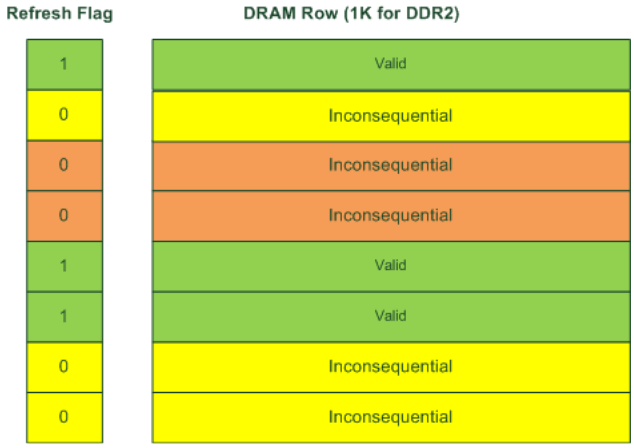


Figure 7.1 DRAM refresh optimization

Due to the dynamic nature of a DRAM cell, periodic refresh operations are required for keeping the data stored. Even in standby mode, such regular refreshes account for a large energy consumption in DRAMs. Some studies have shown that even in the lowest power mode, the power needed to keep the DRAM contents is about a third

of the total power dissipated. Factors such as the memory vendor and the design technology affect the refresh rate. A refresh interval of 110 ns means, a refresh operation takes place every 110 ns. A refresh operation fundamentally involves reading the DRAM cell out and writing back to the same cell. Although this refresh operation consumes, power and bandwidth, it is inevitable for the sake of data correctness.

This dissertation presents a technique to eliminate unnecessary refreshes by using the program semantics information. The regions of memory that are marked as free (inconsequential) or freshly allocated (inconsequential) have no concern related to correctness since the data stored is of no consequence to the correct execution of the program. Thus, not refreshing the data stored in these regions will not be a concern. Hence, this dissertation avoids refreshing the rows that are known to be inconsequential from the program semantics. There are several technologies proposed in literature [OKM98] as well as patents [SPS1] [SPS2] that allow selective refresh of DRAM rows. Any of these technologies could be used along with knowledge of memory state to save power and energy.

7.1.2 Inconsequential Write Back Optimization (IWB)

The concept of inconsequential memory and inconsequential write back was explained in Chapter 5. The inconsequential state of the cache lines are stored using the validity flag present with each cache line. When a cache lines that has been marked dirty becomes free, and hence inconsequential, its validity bit is reset. When a read miss occurs requiring an eviction of this cache line, one can avoid writing the replaced data to the next level if the validity bit is not set. Hence, it has just one access to the next level case as opposed to two as in regular caches. Doing this reduces the number of access to the DRAM resulting in the reduction of the number of pre-charge discharge cycles as well as

more opportunity for the DRAM to enter into a power saving mode. Hence one can reduce power while also saving on energy. IWB is not applicable to all the cache lines that need a write back since one can only avoid the write backs to cache lines that are inconsequential. The data in these cache lines are not required by future state of the application since it was deallocated by the program. Thus, not reflecting the modified value in the main memory does not affect the program. Since this assumption is based on the program semantic, the system is not introducing any additional error.

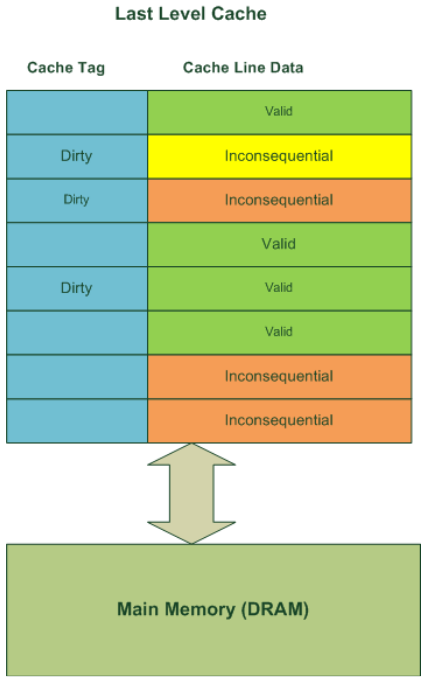


Figure 7.2 DRAM optimization for inconsequential write backs

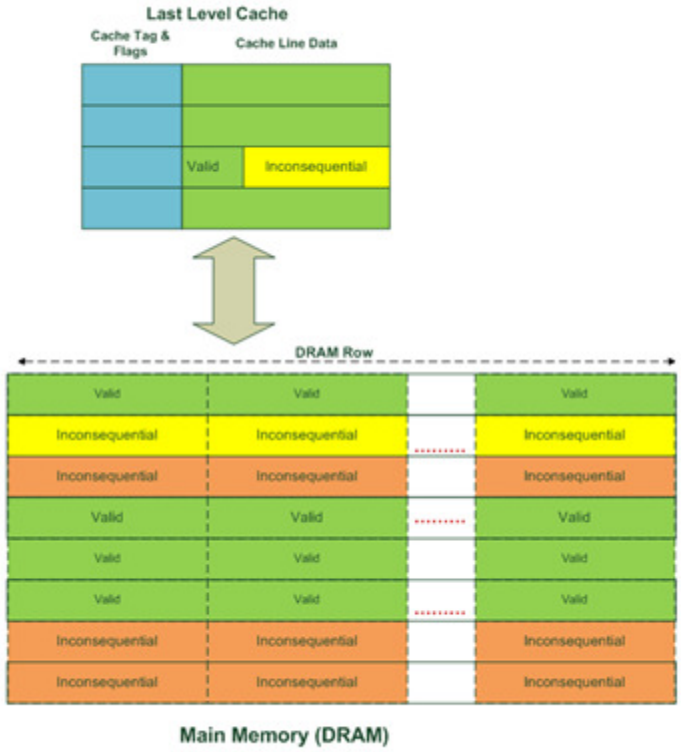


Figure 7.3 DRAM optimization for inconsequential write miss

7.1.3 Inconsequential Write Miss Servicing (IWM)

The concept of inconsequential memory and inconsequential write miss was explained in Chapter 5. The inconsequential state of the cache lines is stored using the TLB; details are explained in the coming sections.

When a write miss occurs, a load is issued to the memory to fill the cache line. If the fetch is from an inconsequential memory segment, one can reduce the number of accesses to the DRAM. Reducing DRAM access results in reduction in the number of precharge discharge cycles as well as more opportunity for the DRAM to enter power saving mode. Lewis et al. [LBL02] explored the use of program semantic information about allocated space for caches to improve performance. Lewis et al. proposes an

allocation range cache and a dedicated instruction to track memory allocation. The allocation range cache is a fully associative 64 entry cache structure that holds allocated memory ranges represented as an address pair (start address and end address). The dedicated instruction is invoked by a modified allocator code during memory allocation. The instruction adds the allocated memory range in the allocation range cache as an address pair entry. Write operations to newly allocated memory are checked by a fully associative search in the allocation range cache. When an address match is found in the allocation range cache, the load issued by a write miss is squashed. Write operations also update the allocation range cache and change the upper or lower bounds represented in the allocation range cache.

Lewis et al. assumes that allocated memory can be tracked using a very small set of memory ranges (64 ranges). Unfortunately the increase in dynamic memory usage and the fragmentation of allocated ranges caused by frequent allocation makes a small subset of ranges inadequate to provide enough coverage. The range based tracking does not perform well with workloads which does not have a strictly incremental allocation pattern. Furthermore, the need to do range check and range adjustment during each store operation is very costly. Using a TLB based structure as proposed in this dissertation provides better scalability since TLB can use the page tables as caches. Furthermore, the TLB based structure can scale memory allocation as well as deallocation. The research presented in this dissertation is compared against an implementation of the ideas of Lewis et al. applied to DRAM at DRAM row granularity. The results for these are represented as Lewis ++ in the relevant charts.

7.2 IMPLEMENTATION DETAILS OF ESKIMO

ESKIMO takes advantage of IWB and IWM for optimizing DRAMs, but ESKIMO needs a few essential common high level aspects for its implementation.

1) One has to be able to detect the transition of normal memory into inconsequential state. This requires co-operation between the memory manager and the hardware.
2) One has to be able to store inconsequential memory ranges so that this information is available when needed. A mechanism is needed to encode information regarding inconsequential memory and allow update and retrieval of this information.
3) One has to be able to detect the transition of inconsequential memory into normal state. Detecting this transition will require co-operation between the memory manager as well as the hardware.
4) One has to be able to use the stored information for the optimizations such as IWB and IWM.

An additional aspect involved in this determination is the granularity at which this information is detected and stored. The next few sections will deal with the details of the implementation mechanisms which address these aspects.

7.2.1 Storage of Inconsequential Memory Status

The inconsequential status of memory ranges need to be stored to facilitate the identification of memory ranges as inconsequential during a write back or write miss to get the advantage of IWB or IWM. In this dissertation, this data is stored as an augmentation to the TLB. A single additional bit per entry in the TLB, i.e., a bit per page (4 KB) stores the inconsequential status of that particular page. By extension, the same bit is swapped out on a TLB replacement into the page table by leveraging existing page table management policies and design. The addition of this bit does not affect the existing

TLB-Page Table policies. The new status bit rather benefits from existing page table policies for managing TLB capacity, TLB entry replacement and TLB coherence. The additional flag bit representing inconsequential state will now on be referred to as the INQ flag bit.

7.2.2 Detection

To communicate the inconsequential memory state, this dissertation presents two instructions similar in format to a few existing memory instructions. For example, the x86 ISA has the INVD (invalidate data cache) and WBINVD (write back and invalidate data cache) instructions while PowerPC ISA has DCBI (data cache block invalidate) and ICBI (instruction cache block invalidate) instructions. The INVD instruction invalidates the whole data cache rather than specific lines while the WBINVD instruction invalidates a cache line but causes a write back if the data contained is dirty. Both the PowerPC instructions unfortunately exist in privileged mode but provide part of the benefit required. The x86 instructions too only exist in privileged mode making all these instructions a good model but not suitable for the desired purpose. To be able to help in the optimizations presented, the instructions devised needs to exist in user mode because they are triggered by the memory management library which exists in user mode. If the instruction is privileged, the cost of transitioning from user to privileged mode during each memory state change i.e. possibly during each allocation or free operation and that would be too costly. The instructions also need to aid in invalidating a cache line irrespective of its modified flag. Additionally they serve to communicate the state of memory ranges to the bookkeeping storage used to store the state of memory ranges. To this end two instructions are discussed here.

INQCL addr, size: This is an instruction meant to be invoked by deallocation operation of at least cache line size. It tells the processor, which in turn communicates to the TLB based storage, that the address range starting at *addr* of size equal to *size* cache lines (i.e. *size**64 bytes) has either been freed by the memory manager or the operating system. The system can now safely assume this address range to contain inconsequential data by setting the corresponding cache lines as invalid using the validity bit. The insertion of INQCL is assumed to be part of the modified memory management library and hence part of the code without explicit invocation by the programmer. The primary purpose of this instruction is to mark cache lines as inconsequential. The maximum size of *size* is one page, thus 6 bits are required for this part of the instruction, if page size is 4 KB and line size is 64 bytes.

INQPG addr, size: This is an instruction meant to be invoked by free or malloc of at least a page. It tells the processor, which in turn communicates to the augmented TLB storage, that the address range starting at *addr* of size equal to *size* pages (i.e. *size**4 KB) has either been freed or freshly allocated by the memory manager or the operating system. The system can now safely assume this address range is inconsequential and this information can be stored in the TLB entry by setting the INQ bit. The insertion of INQPG is also assumed to be part of the modified memory management library or operating system and hence part of the code without explicit invocation by the programmer. The primary purpose of this instruction is to mark TLB entries as inconsequential. The same instruction doubles up to reset the INQ flag bit. INQPG can be made to act as a reset instruction for INQ flag by encoding this information into the flags in the instruction.

7.2.3 Allocator based Book Keeping - HOARD

The previous two instructions track allocations and free operations of memory ranges at two different granularities, one at a cache line granularity and the other at page granularity. The allocation and deallocation of memory happens at various granularity levels which are not necessarily multiples of a page size or page aligned. This makes it essential that some system be responsible for aggregating fragmented allocations and grouping it together to be identified as a whole range worth of inconsequential memory. Even allocations that are larger than a page size can lead to fragmentation. Ignoring those fragments will not only cause loss of opportunity but also complicate the tracking of memory ranges. Consider the following sequence of allocations:

A = Malloc (6 KB);

B= Malloc (1 KB);

C= Malloc (4 KB);

Here, the first 4 KB of the A takes up a whole page which is easy to identify and mark the INQ flag bit the TLB, but the next 2 KB of A takes up only half a page. The 1 KB given to B adds to this fragmentation of the second memory page. This offsets the address range of C to start a little further from middle of the second page causing both the second page as well as the third page to be fragmented even though C allocated a whole page worth of memory. In most memory allocators, memory is allocated in size groups i.e. allocations of a certain size range are all allocated from the same large segment of memory. So requests for 64 bytes and 1 KB will not be allocated from the same physical memory space. Even then, fragmentation exists and is a common issue dealt with by memory managers. In addition to fragmentation, since the assumption of inconsequential nature of data is very hard and strict, it is important to estimate it correctly and conservatively. In this dissertation, this job is piggy backed on the memory manager itself

since it has to perform this job anyway. This allows for the book keeping of allocations and free operations as well as their sizes to be done via software and without needing additional support for hardware. Memory mangers are ideal candidates for this because this task is performed by default by them and they can also take care of ensuring safe estimating of memory state i.e. about it being allocated or free.

This research targets the HOARD [BMBW00] allocator as the sample allocator. This allocator has the benefit of attempting to aggregate recently freed and allocated memory into segments of pages which plays well for the needs of this dissertation. HOARD is chosen because it is a modern allocator compared to the standard *glibc* allocator and it also leads to less fragmentation. Below is the pseudo-code for HOARD allocation and free operations quoted from Berger et al. [BMBW00]:

"**malloc (sz)**
1. If sz > *S*/2, allocate the superblock from the OS and *return* it. Here S is the size of a super block.
2. $i \leftarrow$ hash (the current thread).
3. Lock heap *i*.
4. Scan heap *i*'s list of superblocks from most full to least (for the size class corresponding to sz).
5. If there is no superblock with free space,
6. Check heap 0 (the global heap) for a superblock.
7. If there is none,
8. Allocate *S* bytes as superblock *s* and set the owner to heap *i*.
9. Else,
10. Transfer the superblock *s* to heap *i*.
11. $u_0 \leftarrow u_0 - s.u$
12. $u_i \leftarrow u_i + s.u$
13. $a_0 \leftarrow a_0 - S$
14. $a_i \leftarrow a_i + S$
15. $u_i \leftarrow ui + sz.$
16. $s.u \leftarrow s.u + sz.$
17. Unlock heap *i*.
18. Return a block from the superblock."

Figure 7.4 (a) Memory allocation algorithm in HOARD

"**free (ptr)**
1. If the block is "large",
2. Free the superblock to the operating system and *return*.
3. Find the superblock *s* this block comes from and lock it.
4. Lock heap *i*, the superblock's owner.
5. Deallocate the block from the superblock.
6. $u_i \leftarrow u_i$ - block size.
7. $s.u \leftarrow s.u$ - block size.
8. If $i = 0$, unlock heap *i* and the superblock and *return*.
9. If $u_i < a_i - K * S$ and $u_i < (1 - f) * ai$,
10. Transfer a mostly-empty superblock $s1$ to heap 0 (the global heap).
11. $u_0 \leftarrow u_0 + s1.u$, $u_i \leftarrow u_i - s1.u$
12. $a_0 \leftarrow a_0 + S$, $a_i \leftarrow a_i - S$
13. Unlock heap *i* and the superblock."

Figure 7.4 (b) Memory deallocation algorithm in HOARD

In both the allocation (*malloc*) and deallocation (*free*) operations, there are several steps that track the size of the super block as well as the heap space; lines 15 and 16 in malloc and 6 and 7 in free in particular. These steps are used to insert the correct call to the ISA to convey information about memory management state i.e. information about allocated or deallocated memory region.

The new pseudo-code for the allocator with the new instructions inserted appropriately looks as follows:

"**malloc (sz)**
1. If sz > *S*/2, allocate the superblock from the OS and *return* it. Here S is the size of a super block.
2. $i \leftarrow$ hash(the current thread).
3. Lock heap *i*.
4. Scan heap *i*'s list of superblocks from most full to least (for the size class corresponding to sz).
5. If there is no superblock with free space,
6. Check heap 0 (the global heap) for a superblock.
7. If there is none,

8. Allocate *S* bytes as superblock *s* and set the owner to heap *i*.
9. Else,
10. Transfer the superblock *s* to heap *i*.
11. $u_0 \leftarrow u_0 - s.u$
12. $u_i \leftarrow u_i + s.u$
13. $a_0 \leftarrow a_0 - S$
14. $a_i \leftarrow a_i + S$
15. $u_i \leftarrow ui$ + sz.
16. $s.u \leftarrow s.u$ + sz."
17. If (s.u multiple of Page_Size)
18. INQPG s.u, (s.u/Page_Size)
"19. Unlock heap *i*.
20. Return a block from the superblock."

Figure 7.4 (c) Memory allocation algorithm modified to use INQPG instruction

"**free (ptr)**
1. If the block is "large",
2. Free the superblock to the operating system and *return*.
3. Find the superblock *s* this block comes from and lock it.
4. Lock heap *i*, the superblock's owner.
5. Deallocate the block from the superblock.
6. $u_i \leftarrow u_i$ - block size.
7. $s.u \leftarrow s.u$ - block size."
8. If (s.u/Page_Size)
9. INQPG s.u, (s.u/Page_Size)
10. INQCL 0xAB14, (S/Line_Size)
"11. If $i = 0$, unlock heap *i* and the superblock and *return*.
12. If $u_i < a_i - K * S$ and $u_i < (1 - f) * ai$,
13. Transfer a mostly-empty superblock $s1$ to heap 0 (the global heap).
14. $u_0 \leftarrow u_0 + s1.u$, $u_i \leftarrow u_i - s1.u$
15. $a_0 \leftarrow a_0 + S$, $a_i \leftarrow a_i - S$
16. Unlock heap *i* and the superblock."

Figure 7.4 (d) Memory allocation algorithm modified to use INQPG and INQCL instructions

As seen from the pseudo code, after a *malloc* operation, when the super block size reaches a multiple of the page size an INQPG instruction is invoked with the start of the page address as its argument (lines 17 and 18). INQPG conveys to the TLB that this data

block has been freshly allocated and this information is stored. In the case of the *free* operation, when the super block becomes empty, those whole super block whose size is a multiple of the page size; can be marked as inconsequential using INQPG as well as INQCL. INQPG marks the line as free in the TLB while INQCL marks it as free in the caches.

7.3 OPERATION

This section will explain the operation of the additional structures for DRAM optimization for each of the following memory events.

7.3.1 Memory Allocation

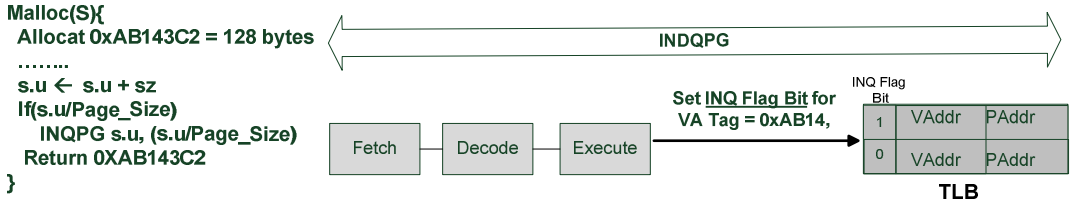


Figure 7.5 (a) Events during allocation of memory

As stated before, the memory manager library will be modified to take advantage of the augmented TLB via the INQPG instruction, since memory management libraries are shared libraries, it saves the normal programmer from having to make changes to his code. As illustrated in Figure 7.5 (a), the library (HOARD in this case) communicates the allocation of an address range using the INQPG instruction which tells the processor that the address range starting at address *A* of size *S* has been freshly allocated by the memory allocator. The instruction informs the TLB, which then looks up the corresponding TLB entry using the virtual address tag (*0xAB14*) corresponding to the address (*0xAB143C2*). If a tag is present then the INQ flag bit corresponding to address is set to one. Note that

the size S is represented at the page granularity. If the TLB does not have a tag corresponding to the current segment, then branding it inconsequential in that attempt is discarded. If the size spans multiple pages then multiple TLB lines will need to be updated.

7.3.2 Memory Free

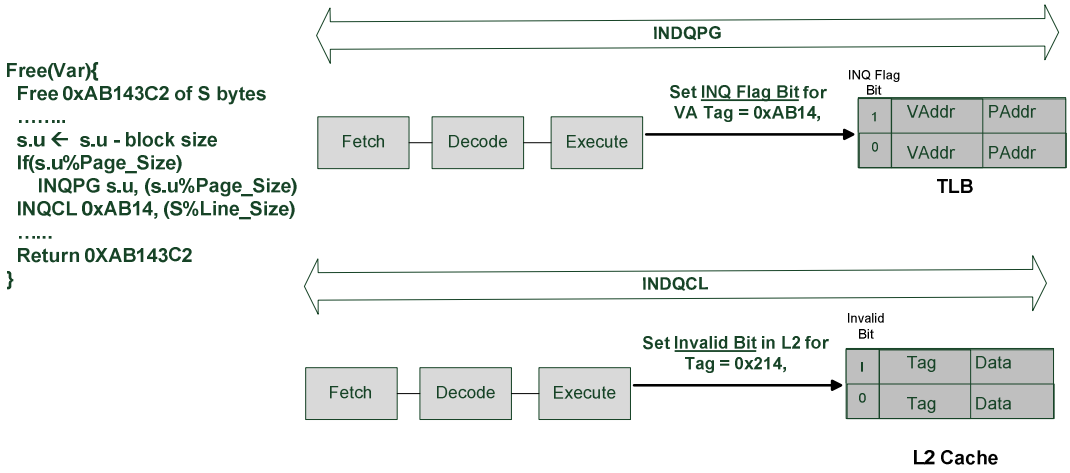


Figure 7.5 (b) Events during free operation

When the memory manger frees a region of memory, the update process to the TLB is the same as that of the allocation, as illustrated in Figure 7.5 (b). The memory manager uses the INQPG instruction to inform the TLB that the address starting at *0xAB143C2* of size *S* has now been set as free. The INQ flag bits in the augmented TLB are updated as in the case of allocation. Additionally the free operation invokes the INQCL instruction to mark the L2 cache lines as inconsequential. The cache lines typically have a valid flag which is reused in this case by the INQCL to mark the cache as inconsequential. This makes the cache line available for replacement and also avoids the

need for write backs even if the line is dirty. The INQCL instruction looks up the TLB to determine the physical address. If there is an entry, the corresponding physical addresses is used to lookup the L2 cache and mark that line as inconsequential by setting the invalid bit. If the instruction fails to find a TLB entry for the address being deallocated, that INQCL instruction is squashed. For example the *0xAB143C2* address range of size *S* is converted to one or more INQCL instructions. The maximum size of *size* in INQCL is one page so if *S>page_size* more than one INQCL is needed. INQCL *0xAB143C2* looks up the TLB and converts the virtual address to a physical address *0x1B14C*. Should INQCL not find the physical address translation of *0xAB143C2* in the TLB, then the INQCL instruction is squashed.

7.3.3 Store Operation

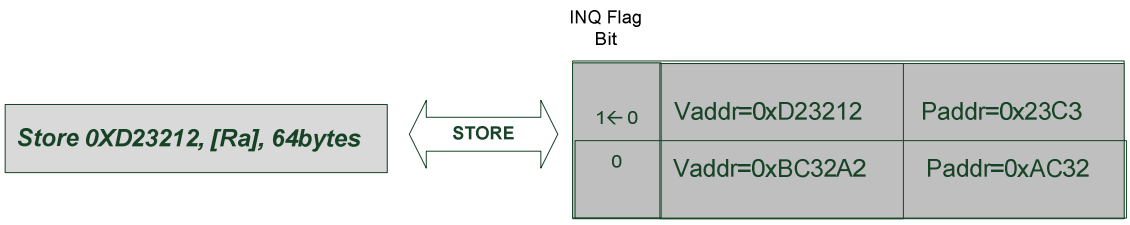


Figure 7.6 Events during store operation

When an address is accessed via a store operation (from any part of the code, not just memory manager code), the TLB is accessed to lookup the address. During the translation process if there is a TLB entry and the TLB INQ flag is set, the store is converted to a special store that requires no write miss servicing. The write miss arising from such a store is satisfied by populating the cache with a new line and filling the data with zero. After the first write miss, future write miss operations cannot safely assume the address range as inconsequential because the store operation has written data. So the TLB INQ flag is reset by the store.

7.3.4 Load Operation

During a load operation, the cache and TLB act normally. If the load causes a write back, then inconsequential write backs are avoided by discarding write backs for cache lines with the invalid bit set even if the dirty flag is set. Other actions in cases such as a TLB miss, page fault, etc are the same.

7.3.5 Caveats

One of the important system effects that come into play in schemes using inconsequential memory is the operation of the Direct Memory Access (DMA). A *DMA* access fetches data for a memory range in the background. This implies that it is possible for a memory to be considered allocated and unused but data might be in the process of being fetched into that memory range via the DMA operation. Since the DMA access is initiated by the operating system, it is detectable; and the TLB is updated using the reset mode of the INQPG instruction. Unlike the set operation of INQPG which ignores the instruction if it fails to be a TLB hit, the reset mode of the INQPG instruction needs to go through the operation of fetching the TLB entry from the page table on a TLB miss. This is necessary to ensure future correctness.

The tracking system proposed uses the TLB to store information. This can introduce coherence issues in a multiprocessor system. The TLB associated with each processor could have cached copies of the page mapping which could become out of sync if one of the threads on one of the processor cores updates the INQ flag in the TLB. This is a concern for scalability because the second core is not aware of the information created by the first core. Furthermore, if processor core 1 identifies a page as inconsequential without propagating it to the other cores, then an eviction of core 1's TLB will update the page table even if another core holds a copy of the translation. In such a case, the second core could start using the page after a memory allocation and data

write without updating the page table, hence resulting in an incorrect state. To solve this, a coherence protocol will need to be developed to keep the TLBs and page table consistent even as multiple threads or programs update the inconsequential status of the pages. Another approach would be to use a single unified last level TLB operating in an inclusive cache like fashion.

In a multiprocessor system, the cache coherence protocol also becomes a factor. When a recently modified cache block is shared between cores, cache line invalidation due to the INQCL instruction has to update the other core caches too to gain full benefit. Fortunately there is no correctness issue, but not updating the other cores can lead to lost opportunity.

The other caveat relates to cases in which the assumption of unnecessary write back does not hold. If the allocated regions of memory are zeroed out for security reasons, the IWB optimization would prevent those writes from propagating from the cache to the memory. When it is necessary to erase current data values in the main memory for security reasons it is not desirable that IWB block such write backs. This issue can be solved by facilitating a specialized case of the deallocation operation which does not mark the deallocated memory regions as inconsequential.

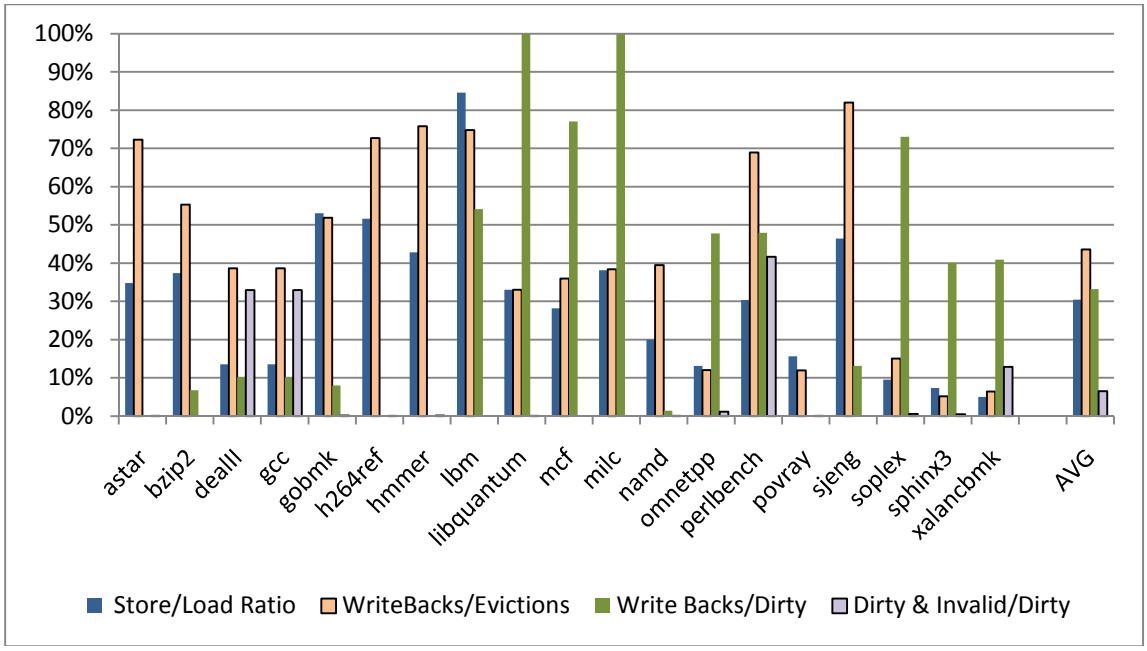


Figure 7.7 Write back related cache statistics

7.4 RESULTS

In this section the results of the analysis and measurements are presented and analyzed. My earlier publication [IJ09] uses the same ideas but ran the benchmarks for only the first billion instructions. The results here are based on at least a 10% run for the benchmark.

7.4.1 Inconsequential Write Back

The analysis of the results starts off with a look at last level cache behavior i.e. a 2 MB, 8-way L2 cache. In Figure 7.7 data related to the load store mix and write back related statistics are presented. The first bar (from the left) in the figure stands for the ratio of store to load instructions. This gives us an insight into the importance of stores in a program. This is of importance because IWM primarily benefits from store instructions that miss the cache. On an average, stores are about 30% of loads. Benchmarks such as *gobmk, h264ref* and *lbm* have half as many stores as loads. It is of interest to look at the

second data bar, the write back/evictions ratio. This ratio is the ratio of all write back operations to the number of evictions from the L2 cache. This represents the percentage of evictions that causes a write back. The higher this ratio the more chance IWB has in finding candidates and reducing energy consumed by the DRAM but what matters is the amount of activity caused in the DRAM because evictions that do not cause a write back normally does not contribute to DRAM cycles and activity. Here most of the benchmarks have a fairly high ratio. For example, *hmmer* has 70% of the evictions resulting in write backs. The next piece of data represents a ratio of the write backs witnessed by the L2 cache to the amount of cache line modifications experienced by the L2 cache. This ratio is on average about 30%. Although this data is a useful indicator, too much cannot be interpreted from this data. Store operations tend to update the same location over and over again hence contributing to the amount of cache line modification count while write backs happen only when the line has become old and is evicted based on LRU. The next bar presents the ratio of the number of times cache lines were marked as both dirty and invalid to the number of times caches were modified.

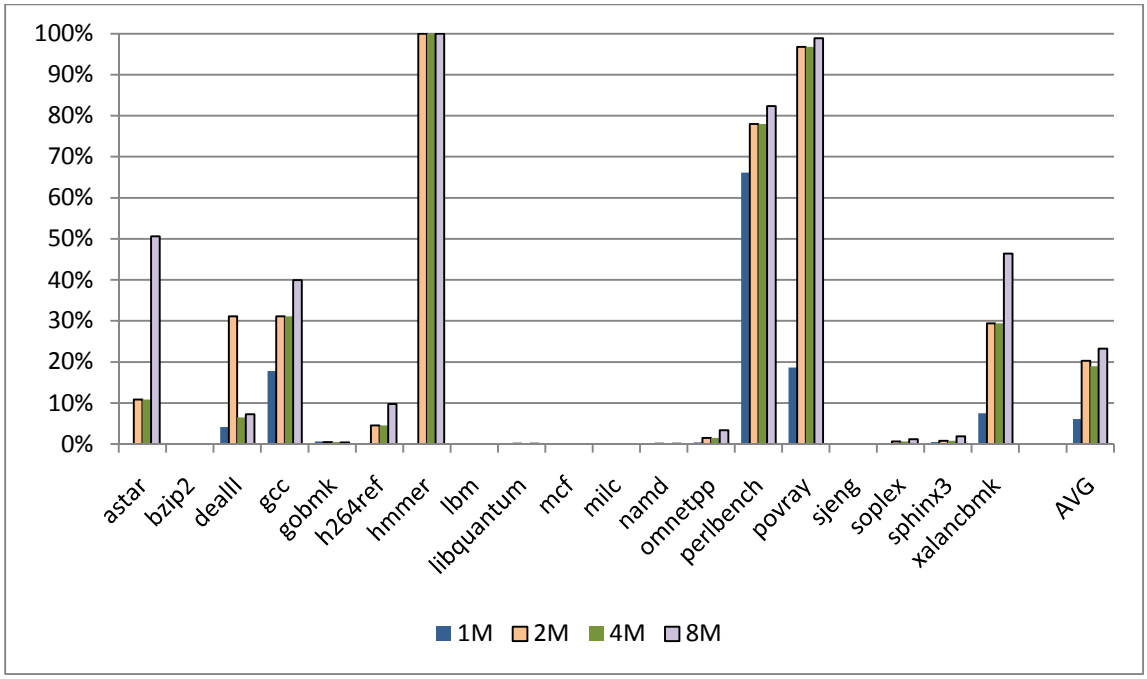


Figure 7.8 Inconsequential write back last level cache size sensitivity

Figure 7.8 is a true measure of the effect of inconsequential write back optimization. The figure presents the sensitivity of this optimization to the size of the L2 cache size (1 MB, 2 MB, 4 MB and 8 MB). The effect of cache size is complex and hard to gauge with two opposing factors at work. On one hand the larger the cache, the more dirty data can stay resident in the cache. This means a free operation will have a better chance of finding its memory range in the L2 cache thereby enabling the IWB optimization. On the other hand, larger caches also facilitate large ranges of memory to be resident and reused. Since memory allocators work on a LIFO principle, recently freed memory locations are first allocated. As time progresses, more and more freed data could become allocated. In self-managed languages with explicit allocation such as C and C++, this pattern is hard to predict. Among the benchmarks here *astar, perlbench, povray* and *xalncbmk* show a positive correlation to the L2 cache size. This suggests that larger cache

sizes capture more of IWB candidates for these benchmarks. The benchmark *hmmer* on the other hand stays unaffected by the cache size. But *dealII* has the odd behavior were it does better with a 2 MB cache than with 1 MB, 4 MB or 8 MB.

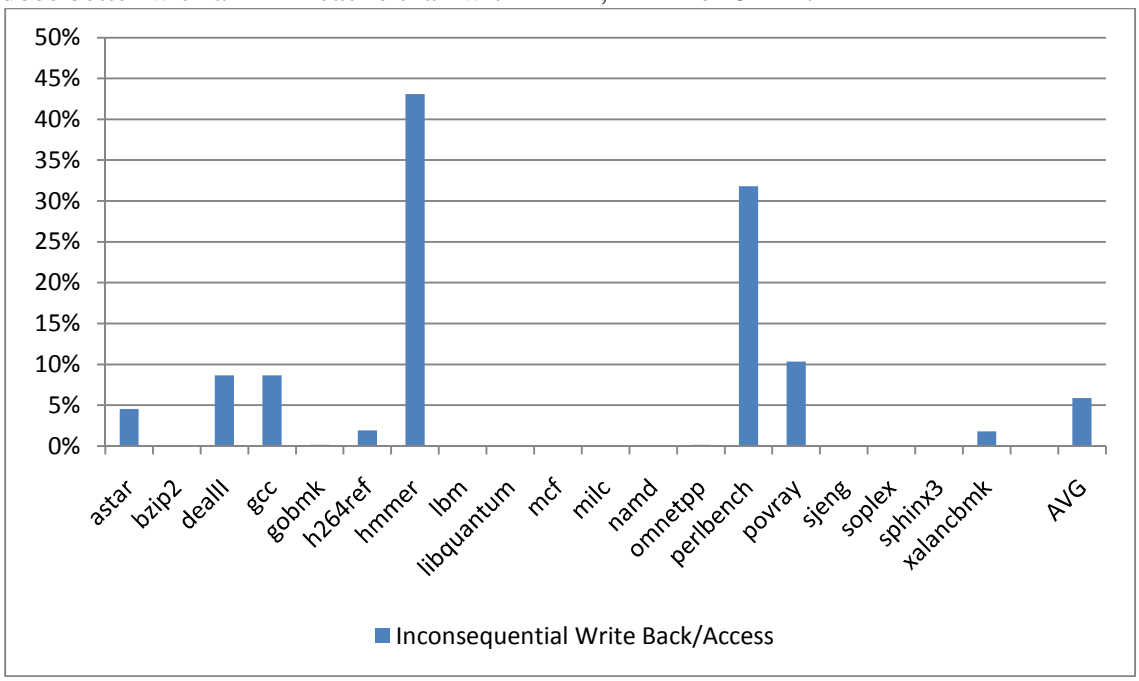


Figure 7.9 (a) Inconsequential write back memory access savings

In Figure 7.9 (a) the effect of IWB on the total memory access, i.e. traffic from L2 to the DRAM memory is presented. Here *astar, dealII, gcc, h264ref, hmmer, perlbench, povray* and *xalancbmk* show response to IWB in relation to the total memory access. On average about 6% of the memory accesses can be avoided using IWB optimization. Figures later on will demonstrate its impact on DRAM energy savings. To put this in perspective Figure 7.9 (b) shows the amount of write backs per thousand instructions.

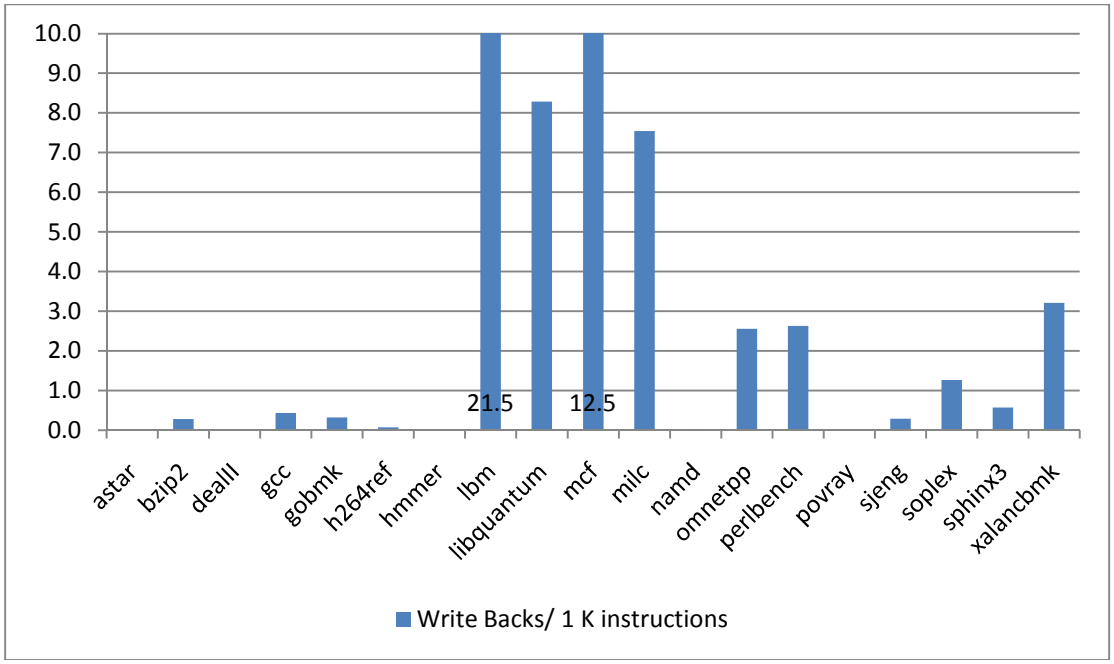


Figure 7.9 (b) Write back per thousand instructions from LLC

7.4.2 Refresh Power Savings

Refresh optimization based power saving technique was modeled using the simulator setup. In Figure 7.10 this dissertation presents the savings in total power arising from the semantic aware DRAM refresh. Note that, the refresh power of a DRAM is only a part of the power consumed by the DRAM. The benchmarks that gain from semantic aware DRAM refresh are the benchmarks that tend to have a larger amount of allocated or freed memory that is not necessarily hot and in the cache but is occupying the DRAM allowing it to be skipped over during refresh. Benchmarks such as *astar, dealII, gcc, h264ref, omnetpp, perlbench, povray, sphinx3* and *xalancbmk* are able reduce total power by a significant amount. The benchmark to demonstrate the best power savings is *omnetpp*, with a 10% reduction in total power consumed by the DRAM. The savings in power among the benchmarks with some impact ranges from 6% to 10%.

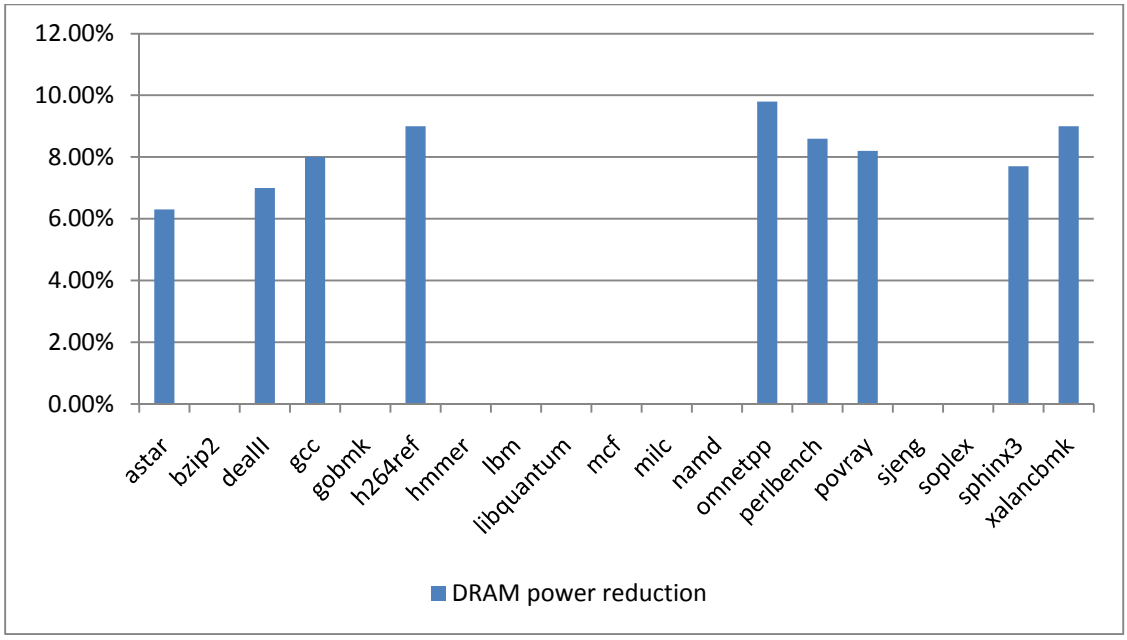


Figure 7.10 Inconsequential data occupancy based DRAM power savings

7.4.3 Inconsequential Write Miss

The next optimization that was modeled is the inconsequential write miss. Figure 7.11 shows the amount of write miss operations that can be avoided for each benchmark as well as the percentage of memory access that can be saved by enabling IWM. The inconsequential write miss optimization leverages the TLB to mark and store the inconsequential state due to fresh memory allocation. Thus, benchmarks that tend to allocate more memory are better candidates for inconsequential write miss. All the benchmarks that show good results from IWM tend to be the ones with good allocation rate particularly with a heavy churn of memory space i.e. memory is allocated, used, freed and then reallocated heavily. The data from Table 6.5 and Table 6.2 shows that the benchmarks *astar, gcc, h264ref, hmmer, milc, omnetpp, perlbench, povray, soplex sphinx3* and *xalancbmk* both allocate a large amount of data and also cycle through heap space. Cross referencing this data with the benchmarks results in Figure 7.11, uncovers

that there is a strong correlation. The only exception is *namd*, a benchmark which did not intuitively seem a good candidate and yet managed to save a significant percentage of its write miss traffic. On average the benchmarks reduce the write miss traffic by 20%. Benchmarks *astar* and *perlbench* does exceedingly well in write miss reduction by reducing as much as 70% of the write miss operations. Since write miss reduction is measured in relation to the amount of write miss traffic, the reduction in real memory access is a better indicator of potential DRAM energy savings. In Figure 7.11 (a) the percentage of memory access avoided is also plotted. Not all the benchmarks manage to convert the reduction in write miss to reduction in memory access. Only *astar, h264ref, hmmer, perlbench* and *povray* manage to convert the write miss to savings in memory access. This implies that for the other benchmarks, the write miss traffic is overshadowed by the normal memory access traffic due to capacity misses. On average, the memory access is reduced by 5% using IWM. In Figure 7.11 (b) the write miss per thousand instructions is shown. In comparing Figure 7.11 (a) and Figure 7.11 (b), we see that *perlbench* benefits the most from IWM. In Figure 7.12 the sensitivity of write miss to last level cache (LLC) size is examined. In this dissertation L2 is the LLC. Figure 7.13 looks at sensitivity of IWM based memory savings to LLC cache size. A larger cache could be capable of capturing more of the access misses due to the capacity limitations of the L2 and filtering that out. Hence the memory access left after that is composed more of misses such as write miss. Compared to Figure 7.11 (a), in Figure 7.13 *bzip2* and *gcc* showed an increased impact with a larger cache size. Benchmark *hmmer* on the other hand has the opposite effect and it suggest that the larger cache is working too well for *hmmer* in capturing most of its working set thereby reducing write misses mostly to first time references.

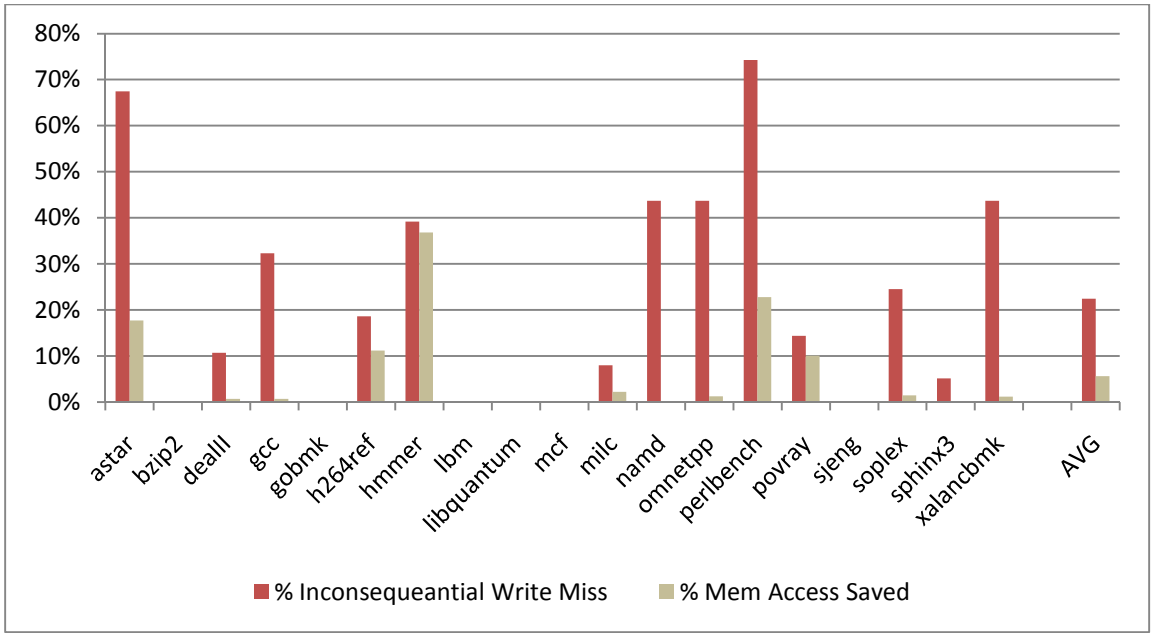


Figure 7.11 (a) Write miss reduction and memory access savings

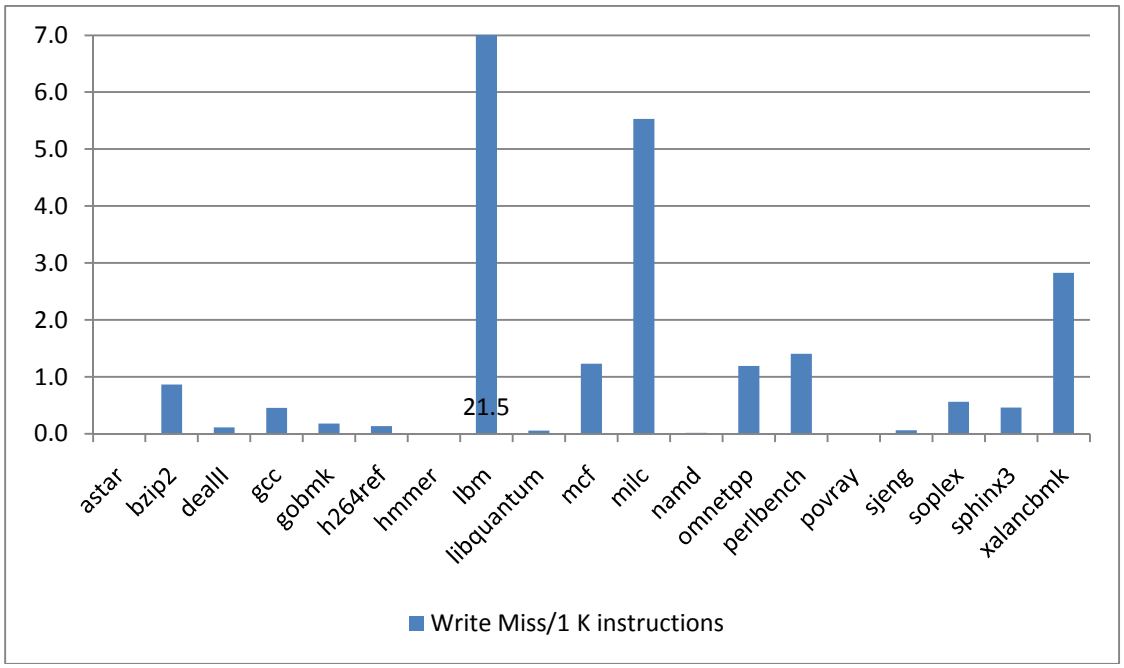


Figure 7.11 (b) Write miss per thousand instruction from LLC

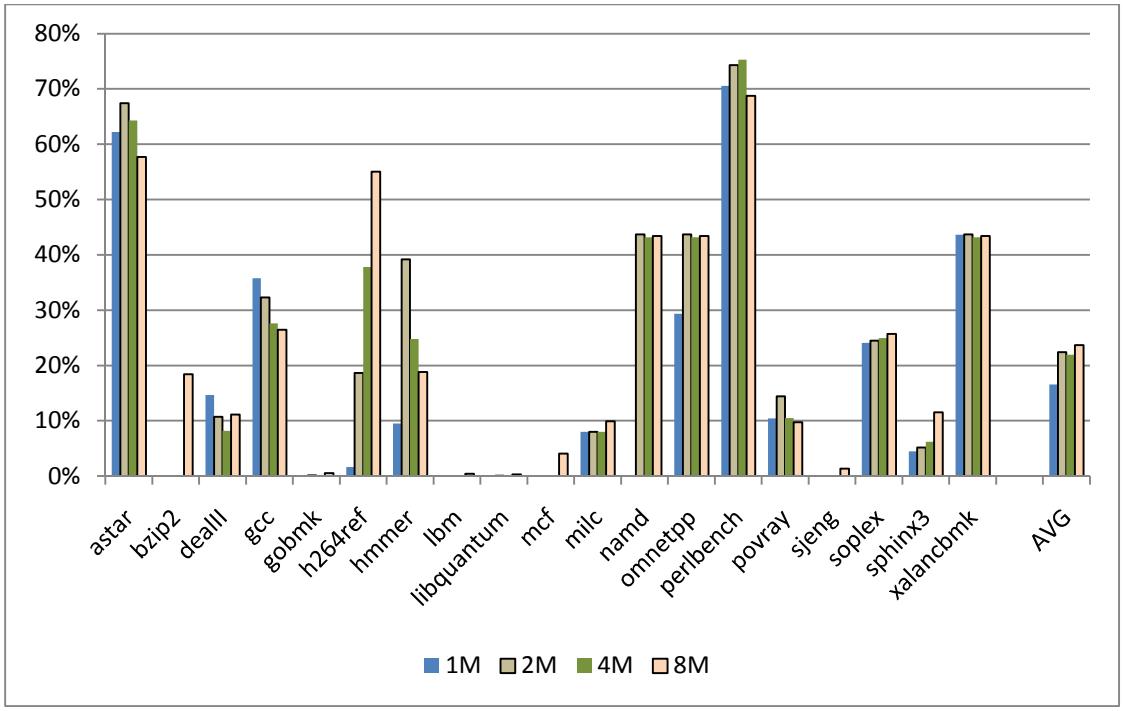


Figure 7.12 LLC size sensitivity - inconsequential data based write miss reduction

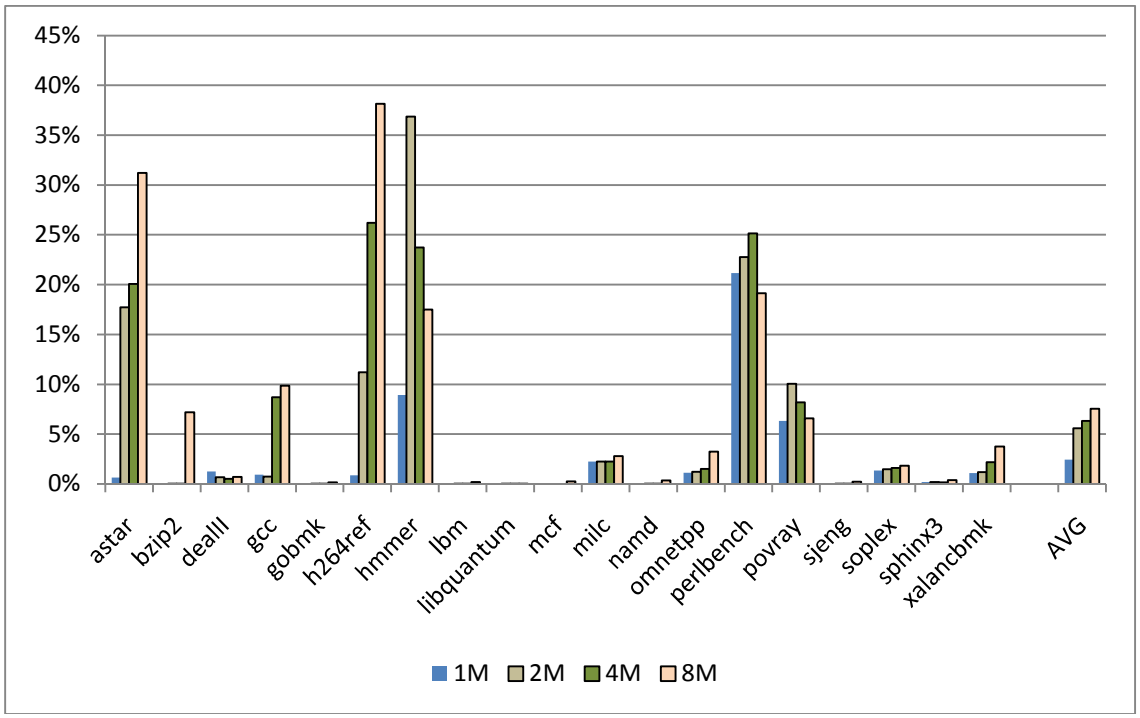


Figure 7.13 LLC size sensitivity - IWM based memory access savings

7.4.4 Energy Savings

The most important measure of all these optimizations is the final aim, which is energy reduction. In Figure 7.14 the reduction in energy due to IWM, IWB and the extended version of Lewis et al.'s [LBL02] work has been compiled. The key difference between Lewis++, the extension of Lewis et al.'s [LBL02] work (Lewis++) for DRAM energy savings, and IWM in this dissertation is in the ability to piggy back on the TLB thereby simplifying implementation, scalability as well as correctness. The energy savings presented here is as a percentage of the total DRAM system power. The memory access reduction due to write backs translates to energy savings for *astar, dealII, gcc, h264ref, hmmer, perlbench, povray* and *xalancbmk*.

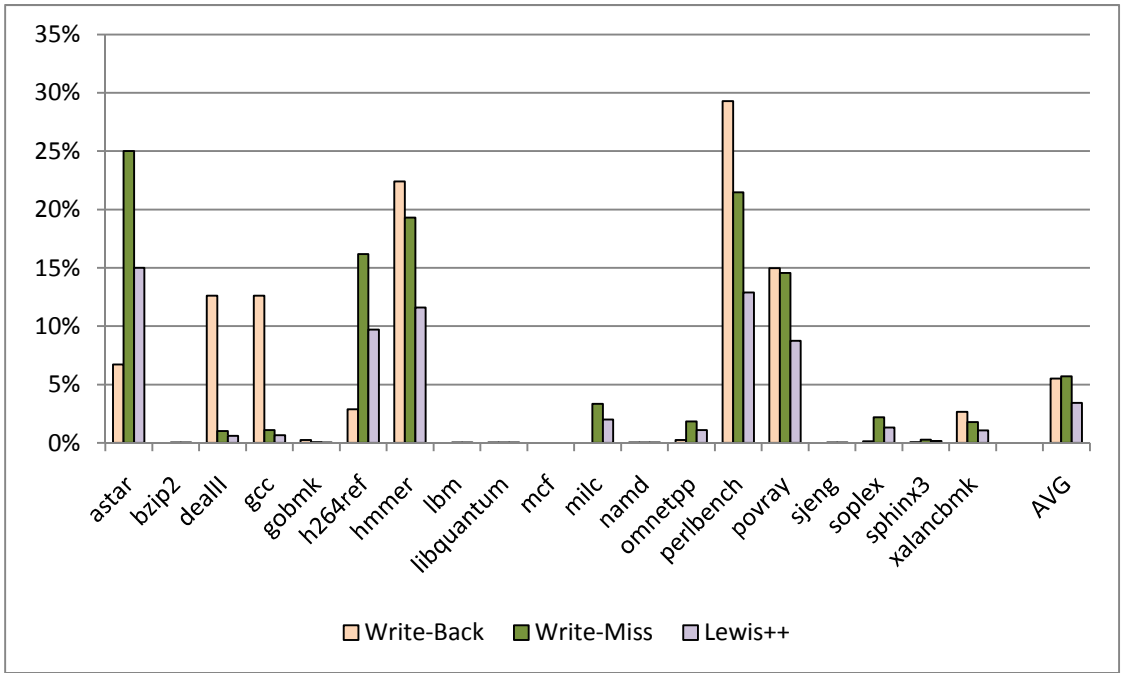


Figure 7.14 DRAM memory energy savings – IWB and IWM

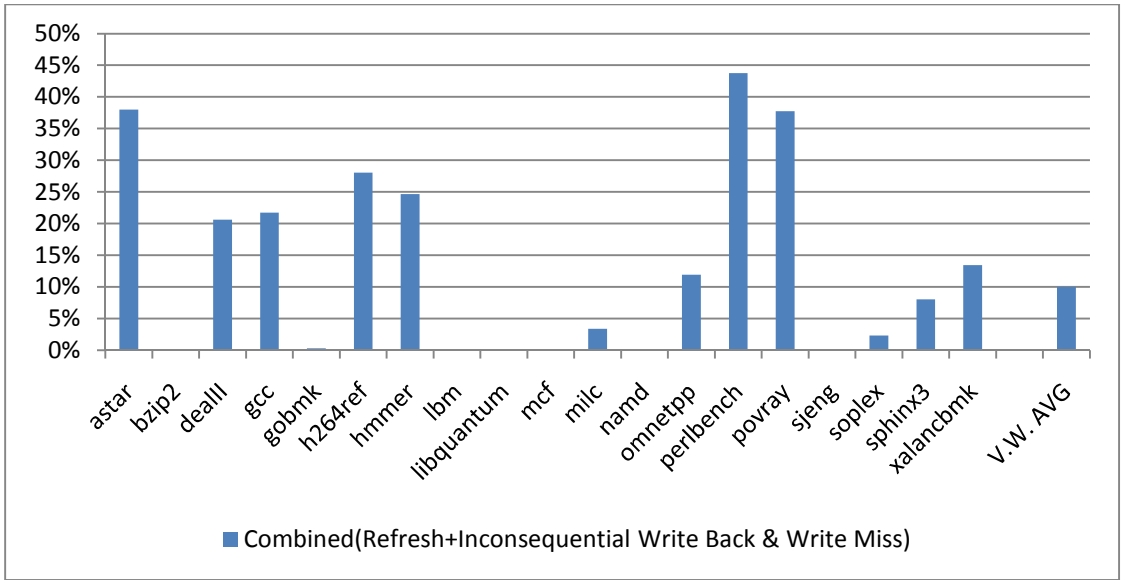


Figure 7.15 Total DRAM memory energy savings

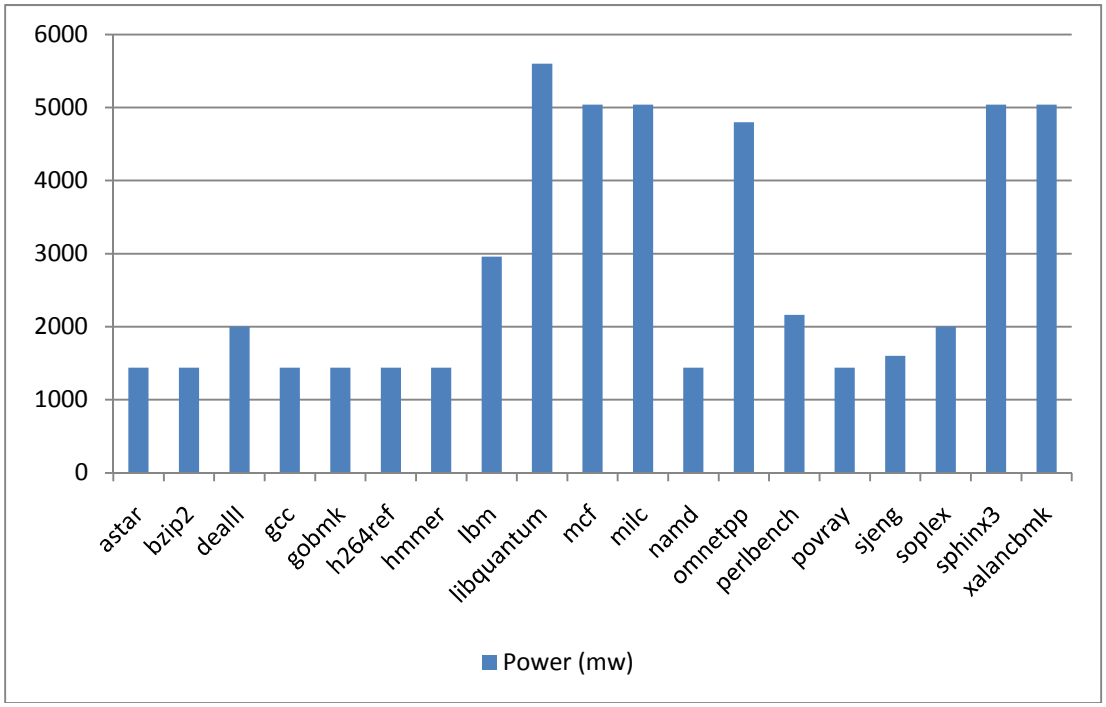


Figure 7.16 Power consumed by in the baseline DRAM

The cumulative result of putting together all the optimization is recorded in Figure 7.15. To put the results in perspective, the power consumed by the benchmarks is presented in Figure 7.16. As indicated in Figure 7.16, different benchmarks consume different amounts of power. Hence instead of simple average, a volume weighted average is more meaningful. The "V.W AVG" refers to ratio of total energy saved in all the benchmarks to the total energy consumed by all the benchmarks. The volume weighted average energy savings is about 10% which is about 7% higher than Lewis++. Four of the benchmarks (*astar, hmmer, perlbench* and *povray*) attain as much as 35% reduction in DRAM energy consumption. Three other benchmarks, (*dealII, gcc* and *h264ref*) show 20% or more reduction in energy consumption. These results speak of the usefulness of these optimizations based on the knowledge of inconsequential memory state such as IWB, IWM as well as semantic aware refresh. Among the high power consuming benchmarks shown in Figure 7.16, (*libquantum, mcf, milc, omnetpp, sphinx3* and *xalancbmk*) *omnetpp, sphinx3* and *xalancbmk* achieve reduction in energy consumption. It should be noted that in addition to reduction in power, the gains in energy also comes from reduction in active DRAM cycles.

In the contemporary world where power is a first class design constraint for system architects (both in the server and mobile space), the consumption of power by the memory subsystem is of particular importance. In this chapter, this dissertation demonstrates ESKIMO which uses a few mechanisms that reduce the amount of power consumed by refresh operations and the amount of write back and write miss reads and writes. These techniques reduce the pressure on the memory and the amount of charging and discharging of lines required, thereby reducing the energy consumed by the memory subsystem. Thus ESKIMO is shown to be a promising adaptation for DRAM based memory.

Chapter 8: mFilter – Increasing Effective Endurance of Emerging Memory Technology based Main Memory

The increasing complexity and size of modern applications puts tremendous pressure on the memory subsystem. Typically, the disk is about four orders of magnitude slower than the main memory making frequent misses in the system main memory very costly for overall performance. Hence, it is important that the memory system be able to feed these applications by supporting their growing working set; which means the memory capacity has to grow along with the working set size to keep up with the applications. DRAM technology has been the corner-stone of main memory in computer systems. Unfortunately, the main memory built from DRAM technology faces severe limitations in terms of power, cost [L03, BZE10] and scaling. In the DRAM technology, DRAM must not only place charge in a storage capacitor but it must also mitigate sub-threshold charge leakage through the access device. It requires the capacitors to be large enough to store charge for reliable sensing and transistors to be large enough to have effective control over the channel. Due to these challenges, viable solutions to manufacture DRAM with scaling beyond 36 nm does not exist and the knowledge on manufacturing techniques are projected only until 21 nm [ITRS09].

Several recent academic works as well as industry research have explored emerging memory technologies such as Phase-Change Memory (PCM), MRAM (Magnetic RAM), FeRAM (Ferroelectric RAM) and Flash [ZYZ09, LIMB09, Q09, QSR09]. For PCM, a 20 nm device prototype has already demonstrated the scaling mechanism and it is projected to scale to 8 nm [ITRS09, ITRS07, R08]. Table 8.1 summarizes the various emerging memory technologies. Most of the emerging memory technologies have the appeal of being a non-volatile storage mechanism (data retention >10 years) with better scaling potential and storage density than DRAM.

	DRAM	PCM	MRAM	FeRAM	Flash
Read Latency vs. DRAM	1x	4x	2x	3x	4x
Write Endurance	-	$<10^{8}$	$<10^{16}$	$<10^{14}$	10^{5}
Write Energy (J/bit)	$5x10^{-15}$	$6x10^{-12}$	$1.5x10^{-10}$	$3x10^{-14}$	$1x10^{-14}$
Retention time	64 ms	>10 years	>10 years	>10 years	>10 years
Process Scaling (Manufacturing solutions exist)	36 nm	8 nm	65 nm	65 nm	25 nm
Current Process Node	45 nm	45 nm	130 nm	180 nm	90 nm

Table 8.1 Summary of different main memory technologies [ITRS09]

However, these memory technologies have their drawbacks too. For example, in an MRAM, write operations require a very high current [ITRS09, BZE10, Z09], about 5 orders higher than DRAM writes, making write operations very expensive in terms of power. FeRAM, PCM and other memories also suffer from endurance limits as can be seen in Table 8.1. Additionally, these emerging memories suffer from a higher latency compared to DRAMs. The endurance limits, the write cost, as well as the higher latency relative to DRAM are challenges that need to be addressed before the positive attributes of emerging memory technology such as scaling and density can be exploited.

New applications, languages and design constraints such as process scaling, power, energy consumption etc makes it important to optimize the design across architectural boundaries. To this end, this dissertation extends the ideas of inconsequential memory optimizations such as inconsequential write backs (IWB) and write misses (IWM) along with zero valued sparse (ZVS) nature of datasets to reduce read/write access to Emerging Memory Technology based Main Memory (EMT).

This dissertation bases the memory organization, shown in Figure 8.1 (a), on the proposal of Qureshi et al. [QSR09] and uses that organization as the baseline. A filter mechanism capable of tracking the inconsequential and zero value states of memory is presented in this chapter. The zero value information is then used to improve emerging memory systems by skipping reads and writes identified as avoidable by the filter. The mFilter is an augmentation to the main memory as seen in Figure 8.1 (b). Its operation and other details are explained in the sections that follow.

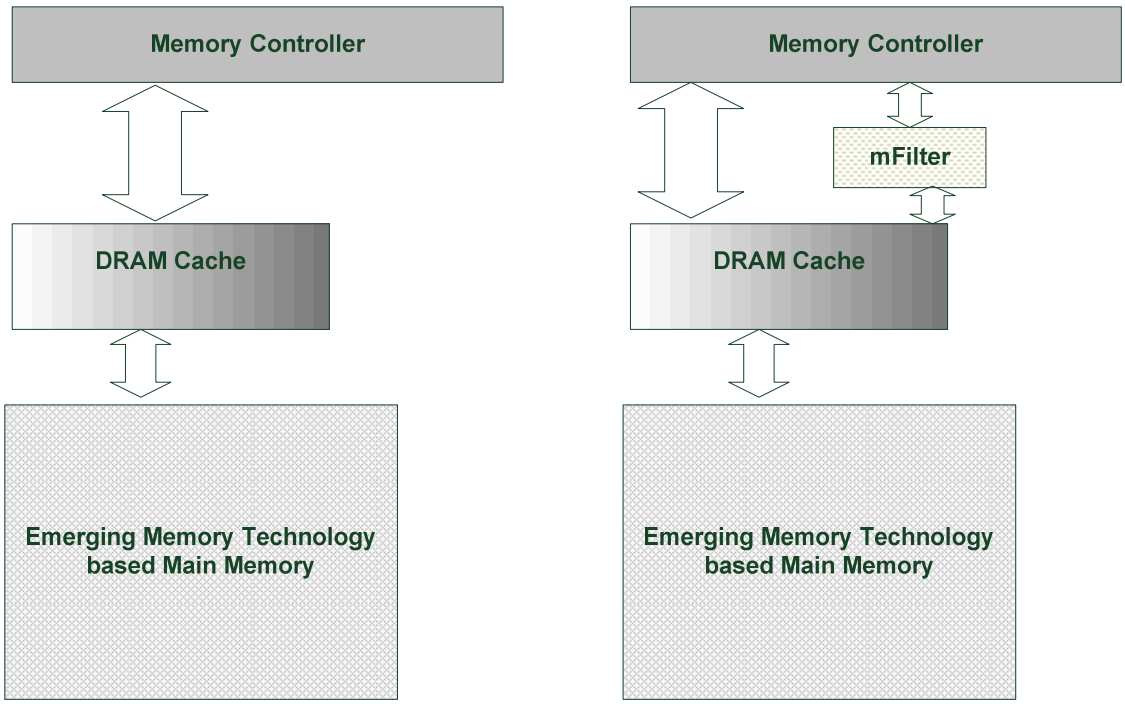


Figure 8.1 A DRAM-emerging memory technology hierarchy based main memory
(a) the baseline [QSR09] (b) the version with mFilter in this research

8.1 MEMORY EVENTS INFLUENCING THE MFILTER DESIGN

The mFilter is meant to track and store certain events which are covered next. The key factor in the design of the mFilter is that it needs to encompass information related to all these events uniformly.

8.1.1 Inconsequent Write Backs (IWB)

The insight into IWB detailed in Chapter 5 can also be used for EMT's. When a line is evicted, one can avoid writing the replaced data to the second level main memory which is the EMT based main memory, if it can be cross checked as to being in an inconsequent write i.e. IWB. Furthermore, marking a line as invalid makes it the next candidate for replacement which can improve the efficiency of the cache and thereby reduce load latency. Doing so would reduce the number of writes performed on the memory device. IWB optimization is used in this dissertation to reduce the write back traffic to the EMT memory which helps improve the endurance of EMT.

8.1.2 Inconsequential Write Miss Servicing (IWM)

Chapter 5 has dealt with the details of the basic concept of IWM. The concept of IWM is useful in optimizing EMT's too. In the case of EMT based memory IWM works as follows. When a write miss occurs the corresponding fetch searches the DRAM cache and the mFilter simultaneously. When a match is found in the mFilter, it implies that a previous *malloc* or operating system page allocation had allocated this memory region. Hence the load from EMT for a DRAM cache miss can be bypassed when possible and the request can be serviced quite easily with a set of zero bits. Reducing load misses due to IWM helps in reducing the access latency of EMT. During a read operation, the access occurs in parallel and the savings that arise from a faster response due to a hit in the mFilter is in latency and not reduction in access.

8.1.3 Zero-Value Stores (ZVS)

The concept of ZVS, discussed in detail in Chapter 5, is useful for EMT's. When stores write zeros to memory one can attempt to reduce the number of stores to the Emerging Memory Technology based Main Memory (EMT) memory by reducing the bit

writes to EMT. The stores can be reduced by using a bit map device (the mFilter) to store the data values. A small DRAM based bit array, shown in Figure 8.1, is used in conjunction with the EMT memory to flag blocks of cache line granularity as zeros. The granularity is assumed to be a cache line with one bit per cache line representing zero value. The bit array does not need to be as large as the memory; rather it can operate like a cache. If the granularity of zero value representation is larger, the storage size required would be correspondingly smaller. Thus, zero value stores can be stored in a bit array in a compressed form and avoid updates to the memory.

8.2 DESIGN OF THE MFILTER - ARCHITECTURAL IMPLEMENTATION DETAILS

8.2.1 mFilter –Segmented Bit Map Array

The mFilter is a bit array used to track both allocation/free state of memory as well as zero value stores to memory (Figure 8.2). The intuition applied here is that, data that is in an inconsequential state can be stored as any random set of bits. The choice of the bit values will not affect the correct execution to the program. For this very reason this dissertation represents data in inconsequential memory regions as zeros in the mFilter array. Thus allocated, freed, as well as zero value data can all be stored as zeros. The mFilter array tracks the data at a cache line granularity using one bit per cache line to store state. In essence, the mFilter is a bit array which reflects whether a certain cache line contains zero or not. We could maintain one bit per cache line but it is empirically observed that a smaller set of bits is sufficient to cover the temporal foot print, less than 1 MB. To provide better coverage, the bit map array is organized as a set of address tags and bit map pairs. This dissertation uses a 4 MB storage split into bit maps segments of size 128B i.e. 32K segments. Each bit map segment has a corresponding address tag which provides the start address of the cache lines represented by the bit map.

Additionally, a zero detector is used as a part of the mFilter. Zero detection circuits, like a tree of OR gates, are abundant in literature and hence it is not dwelt upon here. It must be noted that the zero detection logic is located inside the memory controller which in turn talks to the DRAM/ EMT as well as the mFilter, as shown in Figure 8.2. The data for the mFilter is modeled to be stored in a DRAM memory which facilitates a fast and cheap resource to be used to augment the EMT memory.

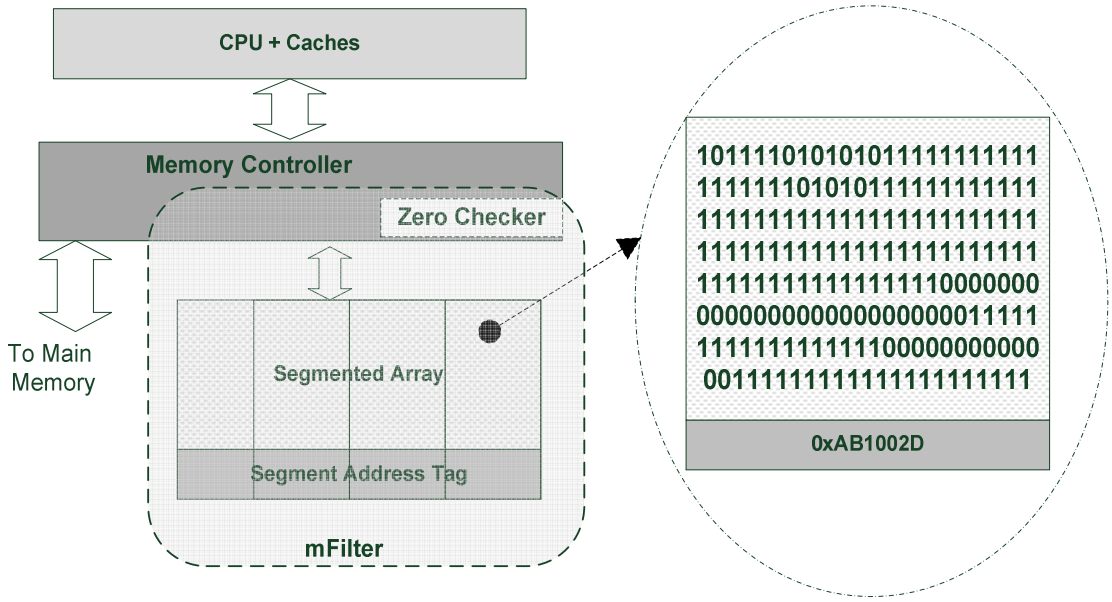


Figure 8.2 The segmented bit map array structure for mFilter

8.2.2 Adaptation in ISA

To communicate memory state to the mFilter, this dissertation proposes a single instruction similar in format to INQCL (from Chapter 7) and a few existing memory instructions. For example, the x86 ISA has the INVD (invalidate data cache) and WBINVD (write back and invalidate data cache) instructions while PowerPC ISA has DCBI (data cache block invalidate) and ICBI (instruction cache block invalidate) instructions. The INVD instruction invalidates the whole data cache rather than specific

lines while WBINVD instruction invalidates a cache line but causes a write back if the data contained is dirty. Both the PowerPC instructions unfortunately exist in privileged mode but provide part of the benefit required. The x86 instructions too exists only in privileged mode making all these instruction a good model but not suitable for the purpose desired. To be able to help in the optimizations presented in this dissertation the instruction needs to exist in user mode because they are expected to be triggered by the memory management library, which exists in user mode. If the instruction is privileged, the cost of transitioning from user to privileged mode during each memory state changes i.e. possibly during each allocation or free operation, which would be too costly. The instructions also need to aid in invalidating a cache line irrespective of its modified flag. Additionally they serve to communicate the state of memory ranges to the bookkeeping storage used to store the state of those memory ranges, the mFilter.

This dissertation proposes INQD addr, size, an instruction meant to be invoked by free or *malloc* of at least cache line size. The instruction communicates to the processor, which in turn communicates to the mFilter, that the address range starting at *addr* of size *size* has either been freed or freshly allocated by the memory manager or the operating system. The *size* is assumed to be a multiple of the cache line size. The system can now safely assume this address range to contain only zeros and this information can be stored in the mFilter by setting the corresponding bits. The insertion of INQD is assumed to be part of the modified memory management library and hence part of the code without explicit invocation by the programmer. The memory allocator or memory manager code helps in detections similar to Chapter 7. Each time a block of memory is allocated or freed, the memory allocator will invoke the *INQD addr, size* instruction similar to the detection mechanism in the DRAM optimization.

8.3 OPERATION OF THE MFILTER

In this section the dissertation presents a walk through the operation of each of the mFilter for various memory events.

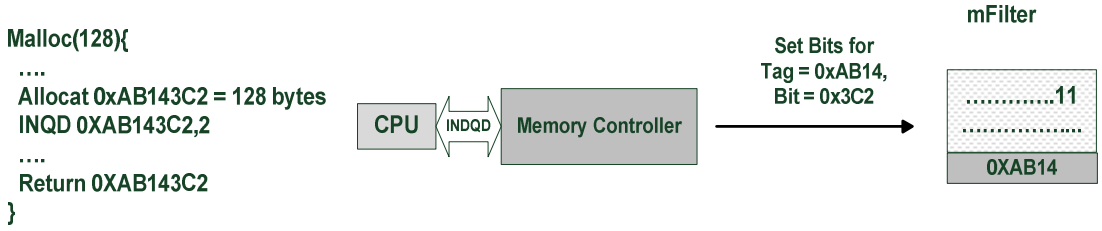


Figure 8.3 Working of mFilter during allocation of memory

8.3.1 Memory Allocation

As stated before, the memory manager library is modified to take advantage of the mFilter via the INQD instruction. Since the memory manager is a system library, it saves the normal programmer from having to do any changes to the application code. As illustrated in Figure 8.3, the library communicates the allocation of an address range using the INQD instruction. The INQD instruction tells the processor that the address range starting at address *A* of size *S* has been freshly allocated by the memory manager. The instruction informs the memory controller which looks up the mFilter for a tag (*0xAB14*) corresponding to the address (*0xAB143C2*). If a tag is present, then the bit corresponding to address is set to 1. If the address spans multiple cache lines, multiple bits are set to cover the necessary range. Note that the size S is represented at cache line granularity. If the mFilter does not have a tag corresponding to the current segment, the mFilter segment data (the tag and the bit array) is evicted based on LRU and the evicted data is stored in the disk. This action requires an interrupt service similar to a page fault and a penalty similar to a page fault is assumed for such an event in the model.

8.3.2 Memory Free

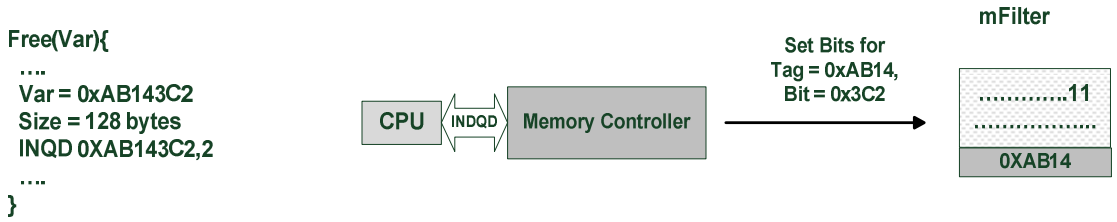


Figure 8.4 Working of mFilter during deallocation of memory

When the memory manger frees a region of memory the update process to the mFilter is the same as that of the Allocation as illustrated in Figure 8.4. The memory manager uses the INQD instruction to inform the memory controller via the processor that the address starting at *0xAB143C2* of size *2* has now been set as free. The bits in the mFilter are updated as in the case of allocation.

8.3.3 Store Operation

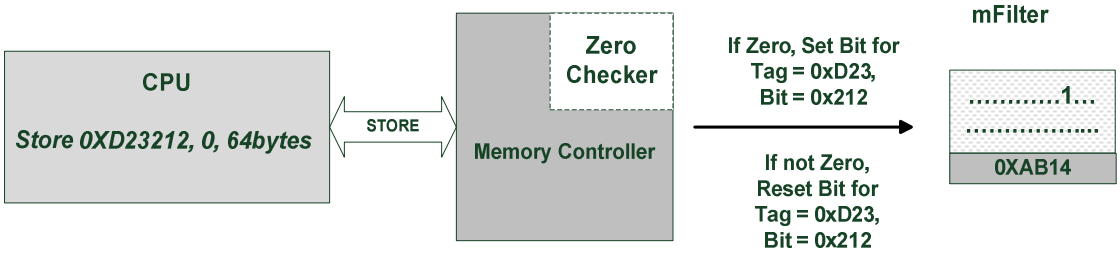


Figure 8.5 Working of mFilter during a store operation to memory

When an address is accessed using a store operation (from any part of the code, not just memory manager code) the memory controller signals the mFilter of this access, as shown in Figure 8.5. If the segmented address tag matches that of the mFilter entry then the bits corresponding to the address are reset to indicate that the address is not in any of the states the mFilter has information about (Allocation, Free or Zero Value). In parallel, the data being stored by the store instruction is examined by a zero checker

circuit. If the data turns out to be a zero value data then the array bits are set to 1. If the mFilter does not have the corresponding tag, the LRU algorithm operates to replace one of the segments.

8.3.4 Load Operation

During a load operation, the memory controller probes the mFilter to check for the status of the address being issued by the memory controller. If a tag exists then the bit corresponding to the tag is checked. If the bit is set, it implies that the memory controller can safely assume the data is a set of zeros and feed the processor zeros without need for further probing in the main memory.

8.3.5 Main Memory Hierarchy augmented with mFilter

The memory hierarchy of an Emerging Memory Technology (EMT) based main memory with a DRAM based cache and an mFilter is shown in Figure 8.1. The DRAM cache helps filter out access to the EMT as well as page fault related stores to the EMT. The access flows down from L1 to L2 and then simultaneously to both DRAM and mFilter. Since the mFilter is based on DRAM and is a fraction of the size of EMT and the DRAM cache, very fast access is possible. Loads that miss the DRAM cache and find a match in the mFilter can be resolved quickly without any access to EMT, thereby hiding the EMT access latency in such cases. For a store operation, the mFilter is set instead of the EMT memory if the data stored happens to be zero value. Thus, the mFilter helps in two fronts. Firstly, it helps reduce the latency of certain loads that find a match in the mFilter. These loads could be loads that load previously stored zero data value or loads to freshly allocated memory or page. Secondly, the mFilter avoids the need to store zero value data in the EMT. A single bit corresponding to the cache line is set to represent zero data value store as well as freshly allocated or freed memory.

Thus, the mFilter avoids a memory request from being serviced if the memory page/cache line being fetched has been allocated recently or if it contains zero value data. If the address being fetched belongs to a region that is inconsequential, that memory region and page is not fetched from the disk. Furthermore, since this data region is inconsequential, the data that is present in the cache line is not important. Thus, it is sufficient to fill up the data region in the cache with zeros instead of random bits. This factor is important because the optimization in this dissertation stores both inconsequential regions and zero data regions using a single bit flag. Although the mFilter does not know the distinction between the inconsequential/unimportant regions and zero data regions, it does not matter and it is able to provide the latency and write optimization benefits. Similarly, when the cache line data write to the memory occurs and is filled with zero value data, the data is stored in the mFilter using a bit flag to indicate that cache line area is zero or inconsequential. The key insight to reiterate here is that both inconsequential data and zero value data can be stored using a single flag and at the time of reproduction both cases can be recreated by populating the necessary space with zero value data.

8.3.6 Discussion of Impact on Other Components and Caveats

8.3.6.1 Memory Consistency

A system that diverges from the typical memory model has to consider its impact on memory correctness in a multiprocessor system particularly in the modern multi-chip/multi-core world. In a multiprocessor system the resolution of dependency and correctness related to a write into the main memory already incorporates correctness checks that are necessary and the implementation in this dissertation does not impose any additional requirement of protection than what is afforded by such consistency checks.

To clarify further, the zero bit is set in the mFilter only when the zero value data is forced to be written from the cache to the main memory.

8.3.6.2 TLB

Since the addresses used in the mapping are important for resolving a match in the mFilter, the presence or absence of an address map in the TLB is a factor of concern. Fortunately a TLB miss does not require one to replace, flush, or invalidate the mFilter since the mapping is still valid and maintained by the virtual memory page table. An eviction of an entry from the TLB does not affect the state of the page. On the other hand, when the address results in a page fault, parallel to resolving the address mapping a selective flush and book keeping (update of data pages) of the mFilter array is necessary. The flushing and book keeping is necessary if the previously mapped address has become remapped or unmapped. It is assumed this can be done in parallel to a page fault handling operation and hence there is no need for an additional penalty for this.

8.3.6.3 Page faults in baseline

The handling of the page fault is slightly different from convention in this memory hierarchy. Normally a page fault would cause the page data to be fetched from the disk and then be stored into the main memory. With the multi-level main memory, instead of storing the data in the EMT memory the data is stored in the DRAM cache first. This avoids unnecessary writes to the EMT memory caused by page fault servicing especially when those pages are not written to. Note that this is not an optimization this dissertation presents but rather one that is present in the baseline system, the work of Qureshi et al. [QSR09].

8.3.6.4 DMA

One of the important system effects that come into play is the operation of the Direct Memory Access (DMA). A *DMA* access fetches data for a memory range in the background. This implies that it is possible for a memory to be considered allocated and unused but data might be in the process of being fetched into that memory range via the DMA operation. Since the DMA access is initiated by the operating system, it is detectable. Using the reset mode of the INQD instruction, the mFilter is updated to indicate that it is not inconsequential any more.

8.3.6.5 Caveats

In a multiprocessor system the cache coherence protocol also becomes a factor. When a recently modified cache block is shared between cores, cache line invalidation due to the INQCL instruction has to also update the other core caches in order to gain full benefit. Fortunately there is no correctness issue, but not updating the other cores can lead to lost opportunity.

The other caveat relates to cases in which the assumption of unnecessary write back does not hold. If the allocated regions of memory are zeroed out for security reasons, the IWB optimization would prevent those writes from propagating from the cache to the memory. For security reasons it might be necessary to erase current data values in the main memory and hence it is not desirable that IWB block such write backs. This issue can be solved by facilitating a specialized case of the free operation which does not mark the deallocated memory regions as inconsequential.

8.4 RESULTS

8.4.1 DRAM Cache Occupancy

Before looking at the results of the optimizations presented for EMT, it is of interest to look into how the DRAM cache occupancy. Figure 8.6 presents the occupancy of cache lines for each category. For example, the "%of Dirty Lines" presents the percentage of the total cache entries that are marked modified at the end of the simulation. For almost all the benchmarks, at least 15% of the DRAM based cache lines are dirty; this amounts to an average of 50% which is almost the same as the average dirty line percentage for L2 cache shown in Figure 5.5. The patterns for the individual benchmarks vary. Almost all the benchmarks have a different amount of DRAM cache size dirty vs. the L2 cache This is to be expected since the L2 cache is in the 2 MB range while the DRAM based cache is 64 MB. Since IWB benefits from the amount of dirty data, a high percentage of dirty data is useful to the proposed technique. Another factor that is very important to the usefulness of IWB is the amount of cache lines that are both dirty and invalid thus giving rise to IWB. Programs such as *astar, gcc, hmmer, perlbench* and *xalancbmk* have a notable amount of cache lines and dirty cache lines that are invalid. *Xalancbmk* leads the pack with 50% of the cache lines being marked invalid and those turn out to be dirty and invalid as well. On average about 4-5% of all the DRAM lines are dirty and invalid.

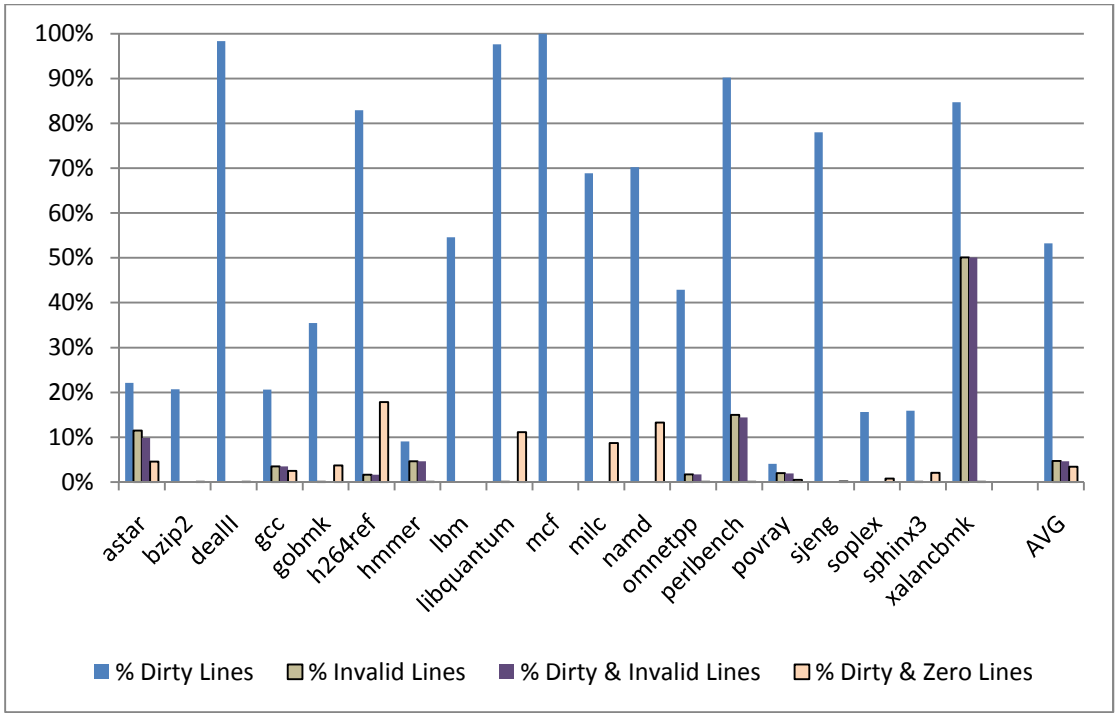


Figure 8.6 Pre-EMT DRAM cache occupancy

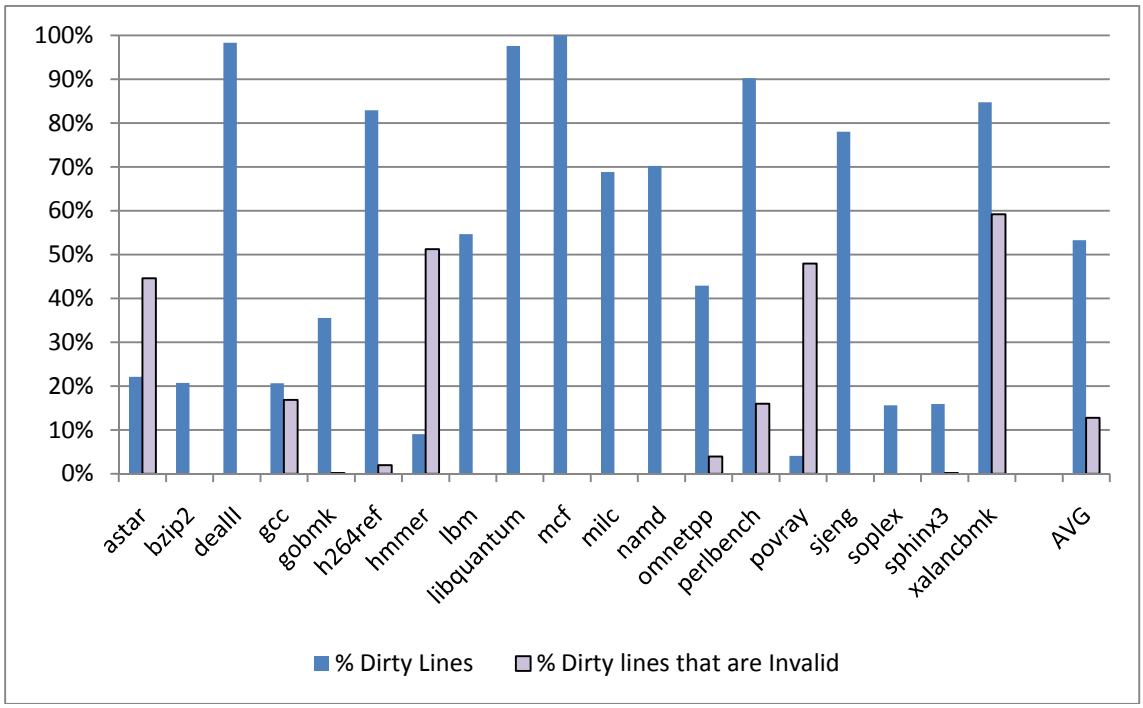


Figure 8.7 Pre-EMT DRAM cache occupancy for dirty lines

Figure 8.7 presents the cache occupancy data as a percentage of dirty lines. The first bar (from the left) is the percentage of DRAM cache entries that are dirty while the second bar represents the percentage of those cache lines that are marked invalid i.e. inconsequential. On average about 12% of the dirty lines are inconsequential and hence marked as invalid too.

8.4.2 Write Miss Reduction based on Inconsequential Write Miss (IWM)

Figure 8.8 and Figure 8.9 shows the savings arising from IWM. Figure 8.8 presents the write miss related data; the first bar (from the left) is the percentage of load misses in the DRAM cache due to write miss. The second bar represents the percentage of write misses that are inconsequential. On average 42% of the load misses from the DRAM cache are caused by write misses. This is a good starting point for IWM, meaning the savings it achieves will affect on average 42% of the load miss accesses. *Gobmk, mcf, omnetpp* and *soplex* have a very low percentage of load misses that arise from write misses which automatically makes them bad candidates for IWM. The other benchmarks have more than 15% of the load misses coming from write miss. Even with the larger amount of write misses, IWM has an impact only if a good percentage of those write misses are identified as inconsequential. On average 20% of write misses are inconsequential. Benchmarks such as *hmmer, lbm* and *sjeng* show good promise for IWB with the amount of write miss based load misses, but those fail to convert to inconsequential write miss candidates. Hence these benchmarks too make for bad candidates for IWM. Even though *libquantum* has a much smaller percentage of the write misses as inconsequential it has a very high amount of write miss based load miss. The rest of the benchmarks are expected to be reasonable candidates for IWM. Figure 8.9 has the actual amount of memory access saved using IWM. As expected *gobmk, mcf,*

omnetpp, soplex, hmmer, lbm and *sjeng* perform poorly. All the other benchmarks perform reasonably. On average (volume weighted) there is a 3% reduction in memory access due to IWM.

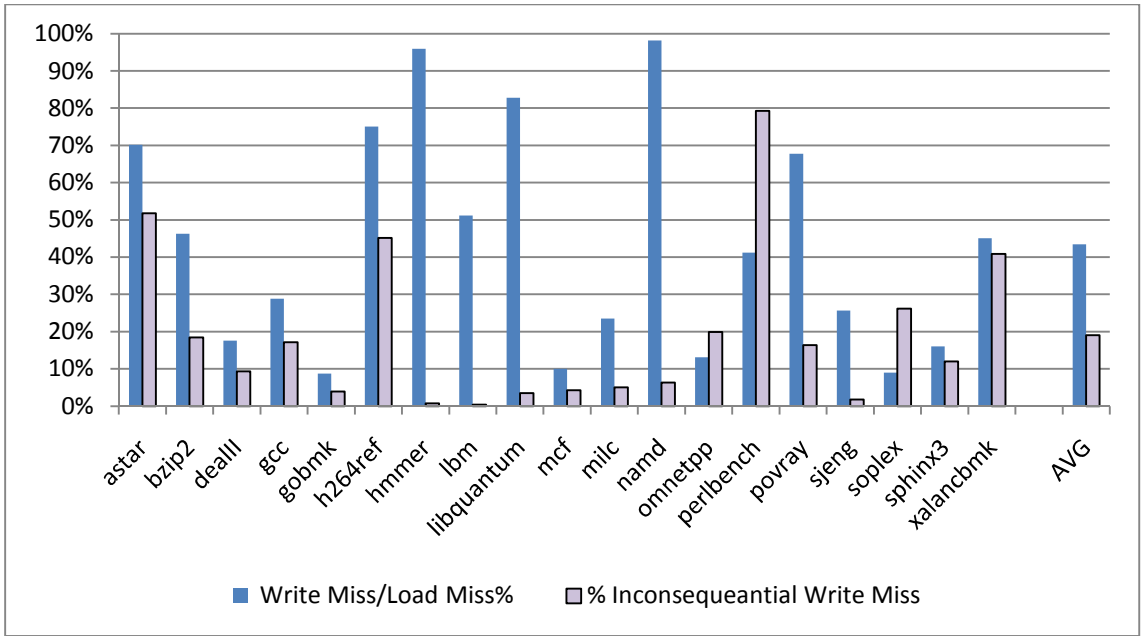


Figure 8.8 Pre-EMT DRAM write miss statistics

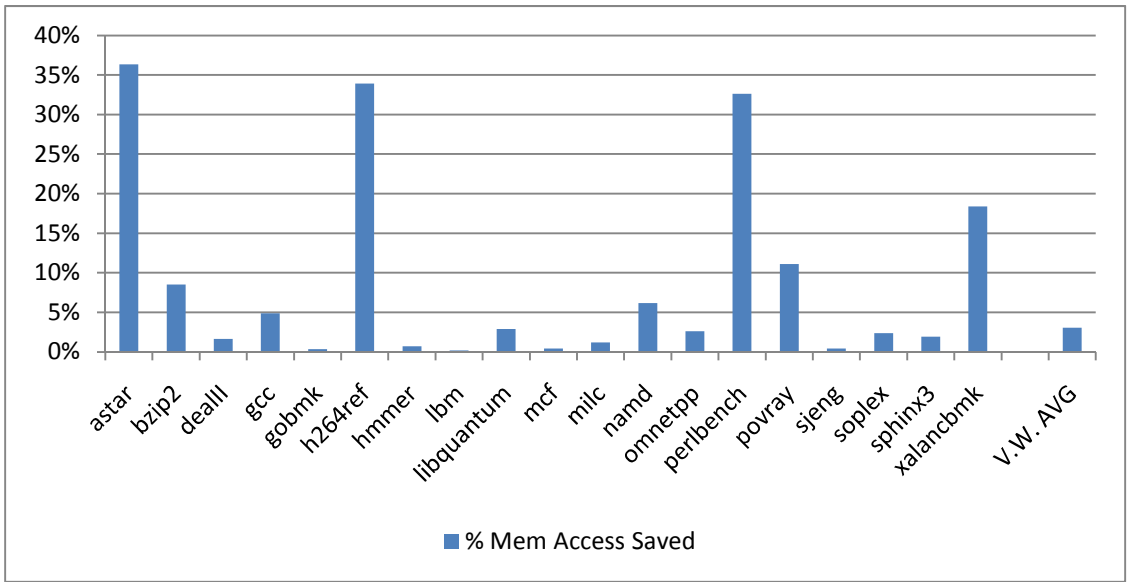


Figure 8.9 Memory access reduced via IWM

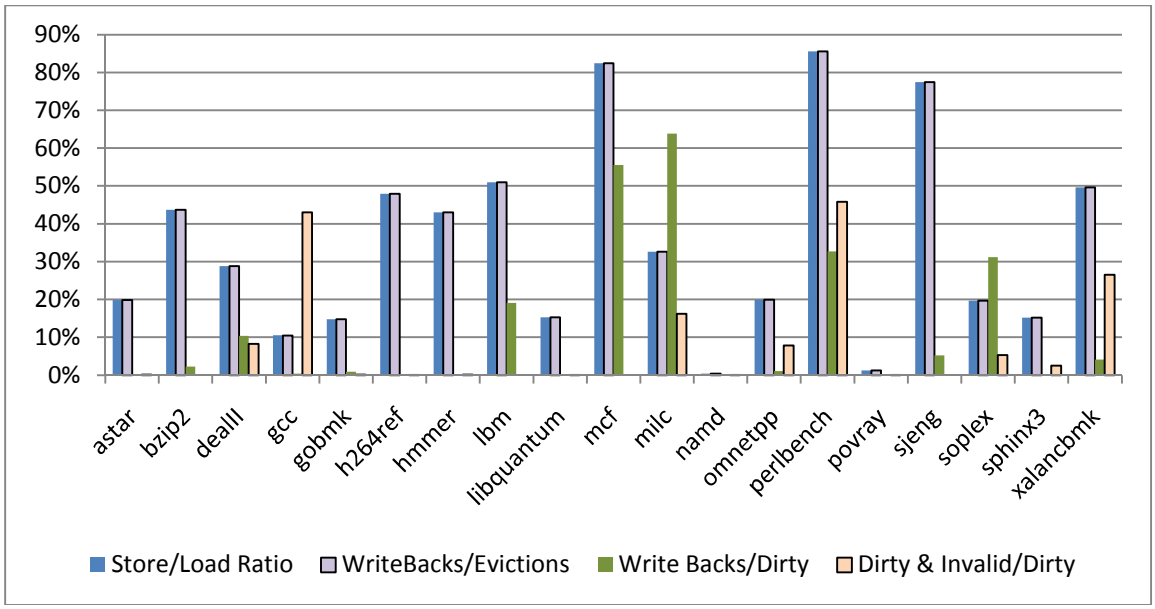


Figure 8.10 Pre-EMT cache state

8.4.3 Write Back Reduction due to IWB and ZVS

Before looking at the results of the IWB optimization presented for EMT, it is of interest to see how the DRAM cache occupancy is related to write backs. Figure 8.10 presents the occupancy of DRAM lines for each category. For example, "% Write Backs/Evictions" presents the percentage of the total DRAM cache evictions that cause a write back. These measurements are based on a snapshot of the DRAM cache at the end of the simulation. The first bar is the ratio of store operations going from the DRAM to the EMT memory to the load operations to the EMT from the DRAM. Based on this snapshot, on average EMT stores are about half that of EMT load operations. For most benchmarks, except *namd* and *povray*, at least 10% of the evictions result in a write back. The ratio of write backs operations to modification operations points to the amount of reuse the lines experience even for data store i.e. the smaller the ratio the larger the reuse because it means modification happen much more often than write backs. The last bar

representing the percentage of DRAM lines that are both dirty and invalid to the dirty lines represents the scope of IWB optimization; on average this is more than 26%.

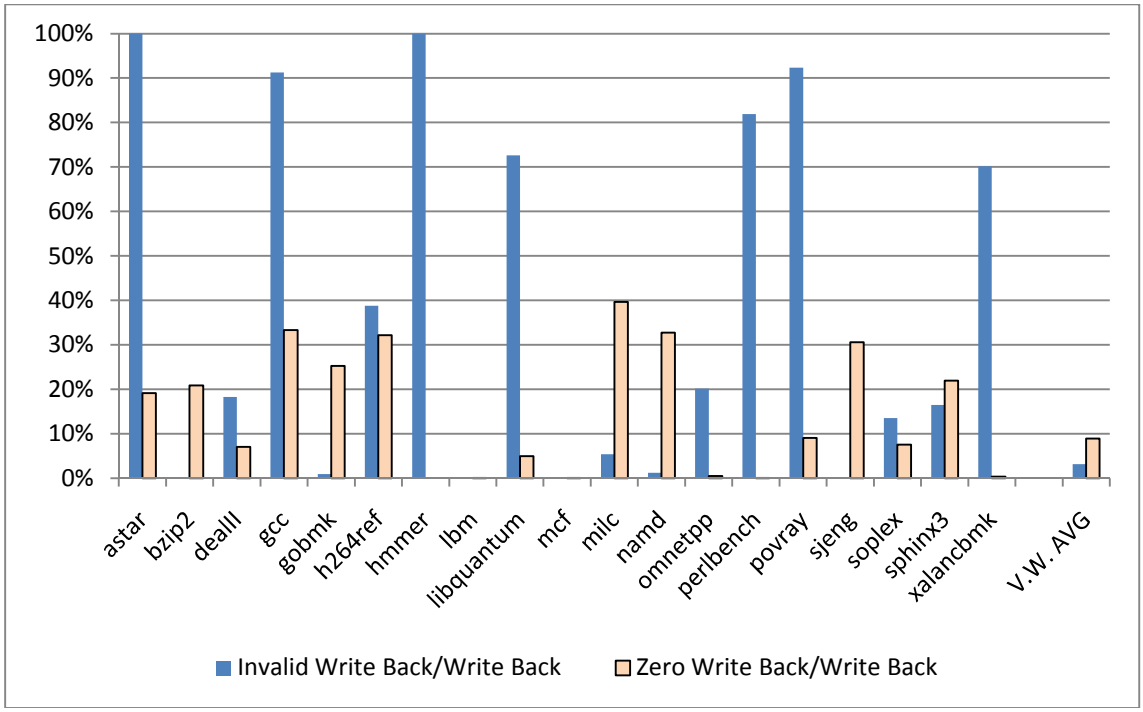


Figure 8.11 Write back reduction due to IWB and ZVS

In Figure 8.11 the actual savings in write backs i.e. the percentage of write backs that are either inconsequential or zero is presented. On average 38% of the write backs are avoided using Inconsequential Write Back (IWB), however when adjusted for volume, the volume weighted average is 3%. After adjusting for volume, on an average (volume weighted), 9% of the write backs are avoided by Zero Value Stores (ZVS). Volume adjustment is done by dividing the sum of total number of write backs saved for all the benchmarks by the sum of the total number of write backs in all the benchmarks. These numbers are converted into true store savings in Figure 8.12. On average (volume

weighted) about 11% of the store operations are avoided by a combination of IWB and ZVS. This is very significant in extending endurance of EMT based memories.

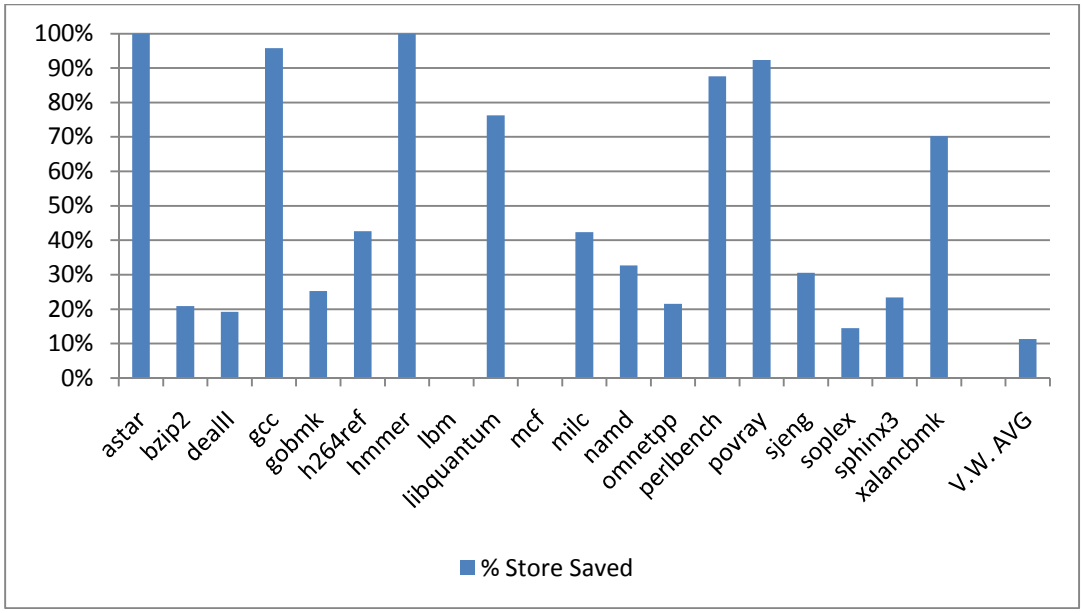


Figure 8.12 EMT stores saved by IWB and ZVS

8.4.4 Lifetime Estimation

Next, this dissertation presents the savings in lifetime of an EMT based memory. The EMT here is assumed to be PCM. In Table 8.2, one can see that the life of an EMT based memory, that has a DRAM cache, can be increased on average by 159% using these optimizations. The most significant improvement seen is for *milc*, an increase in expected lifetime of 3.28 years i.e. 74%. Certain benchmarks generate such a low traffic to the main memory that even without these optimizations, the expected life of an EMT based memory, as considered in the baseline, is very high. The high baseline lifetime of certain benchmarks is factored here. Benchmarks with more than 100 years of baseline lifetime are marked as N/A for the lifetime improvement and ignored from the calculation of useful average lifetime improvement.

The lifetime of the system is estimated using the following formulas:

$$Lifetime\ (in\ seconds) = \frac{EMT\ write\ endurance}{Effective\ Write\ Rate}$$

$$Effective\ Write\ Rate\ (per\ second) = \frac{Writes\ per\ second\ to\ EMT}{Bits\ in\ EMT(including\ mFilter)}$$

The lifetime of a benchmark on the EMT system is computed by finding the number of seconds it takes for any cell to reach the number of writes that breaches the EMT endurance limit. The endurance limit of the EMT technology is divided by the effective write rate (writes per second) for the benchmark to get the number of seconds it takes for the write endurance limit to be breached. Researchers have adapted ideas from FLASH memory to perform wear-leveling in EMT's by performing row shifting [ZYZ09, SLSB10], word shifting [ZL09], randomized address mapping [Q09,QSR09, SWL10] and data remapping [SLSB10, YMC11]. Schechter et al. [SLSB10] also proposed a pointer based error correction code to avoid the constant rewriting required by a normal ECC. To account for the effects of many of these wear leveling techniques proposed for EMT based systems, the effective write rate is determined by dividing the benchmarks write rate over the total number of bits in the system. Thus, the formula mimics the behavior of having the write operations spread out over all the available storage, i.e. avoiding write wear from focusing on a small portion, which is the ideal effect attainable by wear leveling techniques.

Benchmark	Baseline Lifetime (years)	Lifetime improvement(years)	% Improvement in Lifetime
astar	27318	N/A	N/A
bzip2	230	N/A	N/A
dealII	215	N/A	N/A
gcc	1264	N/A	N/A
gobmk	626	N/A	N/A
h264ref	10863	N/A	N/A
hmmer	56670	N/A	N/A
lbm	2.84	0.00	0%
libquantum	24710	N/A	N/A
mcf	2.39	0.00	0%
milc	4.46	3.28	74%
namd	1815751	N/A	N/A
omnetpp	359	N/A	N/A
perlbench	63	446.58	700%
povray	2917716	N/A	N/A
sjeng	162	N/A	N/A
soplex	39.79	6.73	17%
sphinx3	17128	N/A	N/A
xalancbmk	331	N/A	N/A
Useful AVG		91.32	159%

Table 8.2 Savings in lifetime – The improvement in lifetime gained in the optimized EMT based memory subsystem

Chapter 9: Conclusions and Future Work

The growing importance of memory in total costs, energy consumption and performance consideration of servers in datacenters has forced a rethinking of the design of memory subsystems. Increasing pressure on memory capacity due to memory requirements of applications and operating systems, as an artifact of trends such as automatic memory management, in-memory databases, virtual machine consolidation etc, requires increased scaling of main memory capacity, energy and performance to meet modern software demands.

9.1 SUMMARY

In this dissertation, a cross-boundary approach to solve some of these problems is explored. Dynamic memory management state and dynamic data value optimizations used is discussed and an in-depth analysis is done on the benchmarks to observe the scope of these events that are targeted. Of the 19 benchmarks analyzed about 13 benchmarks show sensitivity to such optimizations.

On the energy front, the idea of inconsequential memory is explored via an augmentation to the TLB. A detailed mechanism for detecting, tracking and using inconsequential memory state (inconsequential write backs as well as inconsequential write misses) was developed. Based on such a mechanism a detailed characterization of the benchmarks reveals that 12% of the memory system activity can be curtailed by using optimizations based on inconsequential memory state. This combined with a smart strategy to DRAM refresh leads up to 42% energy savings for some benchmarks and 10% volume weighted average energy savings in memory subsystem.

These techniques are combined with dynamic zero data value stores to improve the endurance of emerging memory technologies. A detailed mechanism for detecting,

tracking and using both inconsequential memory states (inconsequential write backs as well as inconsequential write misses) as well as dynamic zero data value stores was developed. Modeling the use of such a mechanism shows that about 3% of access to the slower emerging memory can be avoided and satisfied with much lower memory latency penalty. The model demonstrates that on an average 11% of the destructive store operations can be avoided thereby increasing the useful life of the emerging technology based memory. This translated to about 159% of increase in lifetime on average for benchmarks that had less than 100 years of lifetime in the baseline PCM based memory system. Among the SPEC CPU2006 benchmarks, only *milc* benefits significantly from the mFilter based optimization.

9.2 FUTURE WORK

All these analysis and results demonstrate that there is significant value in exploiting dynamic memory management state as well as dynamic data value. The knowledge that parts of the last level cache as well as the DRAM based cache are in inconsequential state opens the door for it to be used for aggressive perfecting when the occupancy of inconsequential data increase. A prefetcher that ramps up its activity based on the amount of inconsequential data identified in the last level cache, could be built, thereby reducing the pollution effect of aggressive prefetchers.

The spare space in the caches that are known to be data of inconsequential nature could also help solve reliability issues. It could be possible to exploit these spare cache lines to serve as redundant error correction bits. It is also possible to do this to help switch the cache into a more aggressive lower voltage state. At lower voltages the caches are more susceptible to soft errors. So in phases of the program where parts of the data

cache carries inconsequential data, those lines could double up as additional error correction code and thus allow the voltage to be dropped thereby saving power.

The very same techniques discussed in this dissertation could help alleviate the pressure on memory bandwidth in other applications, particularly OLTP (Online transaction processing) and databases. It is known that the relationship of bandwidth consumption to the main memory and performance is non-linear. Thus, even a small reduction in bandwidth consumption at the right time can have a large impact on the system performance. It is also possible to design simplified communication protocols that take advantage of the prevalence of zero value data that this dissertation has demonstrated.

These are only a few of the many applications of a coordinated approach between software and hardware to exploit the knowledge that the software has about memory and data. These approaches can help to design a less conservative hardware which can, in turn, help resolve some of the issues with hardware design including performance, power and reliability.

Bibliography

[A04] S. J. Ahn, Y.J. Song, C.W. Jeong, J.M. Shin, Y. Fai, Y.N. Hwang, S.H. Lee, K.C. Ryoo, S.Y. Lee, J.H. Park, H. Horii, Y.H. Ha, J.H. Yi, B.J. Kuh, G.H. Koh, G.T. Jeong, H.S. Jeong, Kinam Kim and B.I. Ryu, "Highly manufacturable high density phase change memory of 64Mb and beyond", *IEEE International Electron Devices Meeting,* pp. 907- 910, Dec. 2004.

[AGVO05] J. Abella, A. Gonz`alez, X. Vera, and M. F. P. O'Boyle, "IATAC: a smart predictor to turn-off L2 cache lines", *ACM Transactions on Architecture and Code Optimization* (TACO), pp.55–77, March 2005.

[B04] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, M. Tosi, R. Bez, R. Gastaldi and G. Casagrande, "An 8Mb demonstrator for high-density 1.8V Phase-Change Memories", *Symposium on VLSI Circuits,* pp. 442- 445, June 2004.

[B06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovi, Thomas VanDrunen, Daniel von Dincklage and Ben Wiedermann, "The DaCapo benchmarks: java benchmarking development and analysis", *SIGPLAN Not.* 41, pp.169-190, October 2006.

[B08] F. Bedeschi, R. Fackenthal, C. Resta, E.M. Donze, M. Jagasivamani, E. Buda, F. Pellizzer, D. Chow, A. Cabrini, G.M.A. Calvi, R. Faravelli, A. Fantini, G. Torelli, Duane Mills, R. Gastaldi and G. Casagrande, "A Multi-Level-Cell Bipolar-Selected Phase-Change Memory", *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pp.428-625, Feb. 2008.

[BAJR10] Peyman Blumstengel, Thomas Arenz, Simon Jordan, Alfred Richter, Susanne Brügelmann, Jürgen Häckel, Dieter Gottschling, Hansfried Block, Frank Koch, Michael Korp, Gerold Würthmann and Ulrich Norf, "White Paper - Leveraging Memory Technology to Cut Data Center Power Consumption", November '10. http://www.samsung.com/global/business/semiconductor/Greenmemory/images/download/enterprise/FTS/01/Whitepaper_30nmvs50nm_V21_November_2010.pdf

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson, "Hoard: a scalable memory allocator for multithreaded applications", *SIGPLAN Not.*, pp.117-128, November 2000.

[BZ02] M. Burtscher and B.G. Zorn, "Hybrid load-value predictors", *IEEE Transactions on Computers,* vol.51, no.7, pp.759-774, Jul 2002.

[BZE10] Salah M. Bedair, John M. Zavada and Nadia El-Masry, "Spintronic Memories to Revolutionize Data Storage", *In Proceedings of the IEEE Spectrum,* November 2010.

[C4] CCCC (C and C++ Code Counter). http://cccc.sourceforge.net/

[C96] H.G. Cragon, "Memory Systems and Pipelined Processors", Jones and Bartlett Publishers, Inc., Sudbury, ME, 1996.

[CK94] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Trans. Software Engineering.*, vol. 20, pp. 476-493, 1994.

[CR00] B. Calder and G. Reinman, "A Comparative Survey of Load Speculation Architectures", Journal of Instruction-Level Parallelism, pp.1-39, 2000.

[D89] G. Dunteman, Principal Components Analysis, Sage Publications, 1989.

[EGB03] Lieven Eeckhout, Andy Georges, and Koen De Bosschere, "How java programs interact with virtual machines at the microarchitectural level", In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (OOPSLA '03), ACM, New York, NY, USA, pp. 169-186, 2003.

[ES05] Magnus Ekman and Per Stenstrom, "A Robust Main-Memory Compression Scheme", In *Proceedings of the 32nd annual international symposium on Computer Architecture* (ISCA '05), IEEE Computer Society, Washington, DC, USA, pp. 74-85, 2002.

[EVB02] Lieven Eeckhout, Hans Vandierendonck, and Koenraad De Bosschere, "Workload Design: Selecting Representative Program-Input Pairs", In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques* (PACT '02), IEEE Computer Society, Washington, DC, USA, pp. 83-94.

[FF07] M. Ferdman and B. Falsafi, "Last-Touch Correlated Data Streaming", *IEEE International Symposium on Performance Analysis of Systems & Software,* pp.105-115, April 2007.

[FKMBM02] K. Flautner, Nam Sung Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power", In *Proceedings of the 29th Annual International Symposium on Computer Architecture,* pp.148-157, 2002.

[FW08] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology", *IBM Journal of Research and Development*, vol.52, no.4.5, pp.439-447, July 2008.

[GAV95] Antonio Gonz´alez, Carlos Aliagas, and Mateo Valero, "A data cache with multiple caching strategies tuned to different types of locality", In *Proceedings of*

the 9th International Conference on Supercomputing (ICS '95), ACM, New York, NY, USA, pp. 338-347.

[GL07] Mrinmoy Ghosh and Hsien-Hsin S. Lee, "Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs", In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 40), IEEE Computer Society, Washington, DC, USA, p134-145, 2007.

[HKM02] Zhigang Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: predicting and optimizing memory behavior", *Proceedings. 29th Annual International Symposium on Computer Architecture,* pp.209-220, 2002.

[IJ09] Ciji Isen and Lizy John, "ESKIMO: Energy savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM subsystem", In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 42), ACM, New York, NY, USA, pp. 337-346, 2009.

[IS09] M.M. Islam and P. Stenstrom. "Zero-Value Caches: Cancelling Loads that Return Zero", *18th International Conference on Parallel Architectures and Compilation Techniques*, pp.237-245, Sept. 2009

[ITRS07] Process integration, devices & structures. International Technology Roadmap for Semiconductors, 2007.

[ITRS09] Process integration, devices & structures. International Technology Roadmap for Semiconductors, 2009.

[J93] N.P. Jouppi, "Cache Write Policies And Performance", *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp.191-201, May 1993.

[JCMH99] T.L. Johnson, D.A. Connors, and M.C. Merten, W.-M.W. Hwu, "Run-time cache bypassing", *IEEE Transactions on Computers,* vol.48, no.12, pp.1338-1354, Dec 1999.

[JNW] Bruce Jacob, Spencer Ng, and David Wang, "Memory Systems: Cache, DRAM, Disk", ISBN-13: 978-0123797513

[JS03] J. Jalminger and P. Stenstrom, "A novel approach to cache block reuse predictions", In *Proceedings of International Conference on Parallel Processing,* pp.294-302, Oct. 2003.

[K06] S. Kang, W. Cho, B-H. Cho, K-J. Lee, C-S. Lee, H-R. Oh, B-G. Choi, Q. Wang, H-J. Kim, M-H. Park, Y-H. Ro, S. Kim, D-E. Kim, K-S. Cho, C-D. Ha, Y. Kim, K-S. Kim, C-R. Hwang, C-K. Kwak, H-G. Byun, and Y. Shin, "A 0.1um 1.8V 256Mb 66MHz synchronous burst PRAM", *In International Solid-State Circuits Conference*, 2006.

[KHM01] S. Kaxiras, Zhigang Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power", *Proceedings. 28th Annual International Symposium on Computer Architecture,* pp.240-251, 2001.

[KMS97] Johnson Kin, Munish Gupta, and William H. Mangione-Smith, "The filter cache: an energy efficient memory structure", In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (MICRO 30), IEEE Computer Society, Washington, DC, USA, pp. 184-193.

[KRM08] T. Kgil, D. Roberts, and T. Mudge, "Improving nand flash based disk caches", In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 327–338, 2008

[KS08] Mazen Kharbutli and Yan Solihin, "Counter-Based Cache Replacement and Bypassing Algorithms", *IEEE Trans. Compuert,* pp.433-447, April 2008.

[L03] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller, "Energy management for commercial servers", *Computer*, vol.36, no.12, pp. 39-48, Dec. 2003.

[L06] J. Laudon, "UltraSPARC T1: Architecture and Physical Design of a 32-threaded General Purpose CPU", *Proceedings of the ISSCC Multi-Core Architectures, Designs, and Implementation Challenges Forum*, 2006.

[L08] Kwang-Jin Lee, Beak-Hyung Cho, Woo-Yeong Cho, Sangbeom Kang, Byung-Gil Choi, Hyung-Rok Oh, Chang-Soo Lee, Hye-Jin Kim, Joon-Min Park, Qi Wang, Mu-Hui Park, Yu-Hwan Ro, Joon-Yong Choi, Ki-Sung Kim, Young-Ran Kim, In-Cheol Shin, Ki-Won Lim, Ho-Keun Cho, Chang-Han Choi, Won-Ryul Chung, Du-Eung Kim, Yong-Jin Yoon, Kwang-Suk Yu, Gi-Tae Jeong, Hong-Sik Jeong, Choong-Keun Kwak, Chang-Hyun Kim, and Kinam Kim, "A 90 nm 1.8 V 512 Mb Diode-Switch PRAM With 266 MB/s Read Throughput", *IEEE Journal of Solid-State Circuits*, vol.43, no.1, pp.150-162, Jan. 2008.

[Lai03] S. Lai, "Current status of the phase change memory and its future", *IEEE International Electron Devices Meeting*, pp. 10.1.1- 10.1.4, Dec. 2003.

[LAPKLLCK03] Jaegoo Lee, Yongseok Ahn, Yangkeun Park, Minsang Kim, Dongjun Lee, Kyuhyun Lee, Changhyun Cho, Taeyoung Chung, and Kinam Kim, "Robust memory cell capacitor using multi-stack storage node for high performance in 90 nm technology and beyond", *Symposium on VLSI Technology*, pp. 57- 58, June 2003.

[LBL01] K.M. Lepak, G.B. Bell, and M.H. Lipasti, "Silent stores and store value locality", *IEEE Transactions on Computers,* vol.50, no.11, pp.1174-1190, Nov 2001

[LBL02] J.A. Lewis, B. Black, and M.H. Lipasti, "Avoiding initialization misses to the heap", In *Proceedings of 29th Annual International Symposium on Computer Architecture*, pp.183-194, 2002.

[LF00] A.-C. Lai and B. Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction", In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 139–148, 2000.

[LFF01] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead block correlating prefetchers", In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 144–154, 2001.

[LIMB09] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger, "Architecting phase change memory as a scalable dram alternative", In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ACM, New York, NY, USA, pp.2-13, 2009.

[LW95] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors", *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, 1995.

[LWS96] M. H. Lipasti, C. B Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction", *In Proceedings of the seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147, 1996.

[MICRON] Various Methods of DRAM Refresh. TN-04-30, Micron.

[N08] T. Nirschl, J.B. Phipp, T.D. Happ, G.W. Burr, B. Rajendran, M.-H. Lee, A. Schrott, M. Yang, M. Breitwisch, C.-F. Chen, E. Joseph, M. Lamorey, R. Cheek, S.-H. Chen, S. Zaidi, S. Raoux, Y.C. Chen, Y. Zhu, R. Bergmann, and H.-L. Lung, C. Lam, "Write Strategies for 2 and 4-bit Multi-Level Phase-Change Memory", *IEEE International Electron Devices Meeting,* pp.461-464, Dec. 2007.

[O05] Hyung-rok Oh, Beak-hyung Cho, Woo Yeong Cho, Sangbeom Kang, Byung-gil Choi, Hye-jin Kim, Ki-sung Kim, Du-eung Kim, Choong-keun Kwak, Hyun-geun Byun, Gi-tae Jeong, Hong-sik Jeong, and Kinam Kim, "Enhanced write performance of a 64-mb phase-change random access memory", *IEEE Journal of Solid-State Circuits,* vol.41, no.1, pp. 122- 126, Jan. 2006.

[O68] Stanford R. Ovshinsky, "Reversible Electrical Switching Phenomena in Disordered Structures", *Phys. Rev. Lett.* vol 21, pages 1450-1453, Nov. 1968.

[OKM98] T.Ohsawa, K. Kai, and K. Murakami, "Optimizing the DRAM refresh count for merged DRAM/logic LSIs", In *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 82-87, Aug 1998.

[P03] A. Pirovano, A.L. Lacaita, A. Benvenuti, F. Pellizzer, S. Hudgens, and R. Bez, "Scaling analysis of phase-change memory technology", *IEEE International Electron Devices Meeting,* pp. 29.6.1- 29.6.4, Dec. 2003.

[PHBSC03] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder, "Using SimPoint for accurate and efficient

simulation", In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ACM, New York, NY, USA, pp. 318-319, 2003

[PIN] Pin Website: http://www.pintool.org/

[PIN05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation", *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 190-200, June 2005.

[PJJ07] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite", In *Proceedings of the 34th Annual International Symposium on Computer Architecture,* pp. 412–423, 2007.

[Q09] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling", In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, New York, NY, USA, pp. 14-23, 2009.

[QSR09] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers, "Scalable high performance main memory system using phase-change memory technology", In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pp. 24-33, 2009.

[R08] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-change random access memory: a scalable technology", *IBM J. Res. Dev.* 52, pp. 465-479, July 2008.

[RTDF98] Jude A. Rivers, Edward S. Tam, Gary S. Tyson, Edward S. Davidson, and Matt Farrens, "Utilizing reuse information in data cache management", In *Proceedings of the 12th International Conference on Supercomputing*, ACM, New York, NY, USA, pp. 449-456, 1998 .

[S05] Diomidis Spinellis, "Tool writing: A forgotten art?", *IEEE Software*, pp. 9-11, July/August 2005, http://www.spinellis.gr/sw/ckjm/.

[S10] Jennifer Bedke Sartor, "Exploiting Language Abstraction to Optimize Memory Efficiency", PhD Dissertation, University of Texas at Austin, 2010

[SL09] Sangyeun Cho and Hyunjin Lee, "Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance", In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture,* ACM, New York, NY, USA, pp. 347-357, 2009.

[SLSB10] Stuart Schechter, Gabriel H. Loh, Karin Straus, and Doug Burger,"Use ECP, not ECC, for hard failures in resistive memories", In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ACM, New York, NY, USA, pp. 141-152, 2010.

[SPEC06] SPEC. http://www.spec.org/

[SPS1] Seungyoon Peter Song, "Software control of DRAM refresh to reduce power consumption in a data processing system", US Patent 6542958.

[SPS2] Seungyoon Peter Song, "Method and system for selective DRAM refresh to reduce power consumption", US Patent 6094705.

[SVMW05] Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang, "Cooperative Caching with Keep-Me and Evict-Me", In *Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures*, IEEE Computer Society, Washington, DC, USA, pp. 46-57, 2005.

[SWHKAF04] Stephen Somogyi, Thomas F. Wenisch, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi, "Memory coherence activity prediction in commercial workloads", In *Proceedings of the 3rd workshop on Memory Performance Issues: in conjunction with the 31st International Symposium on Computer Architecture*, ACM, New York, NY, USA, pp. 37-45, 2004.

[SWL10] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee, "Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping", In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ACM, New York, NY, USA, pp. 383-394, 2010.

[TFMP95] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun, "A modified approach to data cache management", In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 93-103, 1995.

[VHR06] R. Venkatesan, S.Herr, and E. Rotenberg, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM", In *Proceedings of the Twelfth Annual Symposium on High Performance Computer Architecture*, pages 155-165, Nov. 2006.

[W05] David Tawei Wang, "Modern DRAM Memory Systems: Performance Analysis and a High Performance, Power-Contrained DRAM scheduling algorithm", PhD Dissertation, University of Maryland, 2005.

[W06] Michael Wong, "C++ benchmarks in SPEC CPU2006", *SIGARCH Computer. Architecture News* 35, pp 77-83, March 2007.

[WGTBJJ05] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob, "DRAMsim: a memory system simulator", *SIGARCH Comput. Archit. News 33*, pp. 100-107, November 2005.

[WM95] Wm. A. Wulf and Sally A. McKee, "Hitting the memory wall: implications of the obvious", *SIGARCH Comput. Archit. News* 23, pp. 20-24, March 1995.

[WMRW02] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems, "Using the Compiler to Improve Cache Replacement Decisions", In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques* (PACT '02). IEEE Computer Society, Washington, DC, USA, pp. 199-208, 2002.

[YLK07] Byung-Do Yang, Jae-Eun Lee, Jang-Su Kim, Junghyun Cho, Seung-Yun Lee, and Byoung-Gon Yu, "A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme", *IEEE International Symposium on Circuits and Systems,* pp.3014-3017, May 2007.

[YG02] Jun Yang and Rajiv Gupta, "Energy efficient frequent value data cache design", In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture* (MICRO 35), pp. 197-207, 2002.

[YMC11] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P. Jouppi, and Mattan Erez, "FREE-p: Protecting non-volatile memory against both hard and soft errors", *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA),* pp.466-477, Feb. 2011.

[YND10] Yongsoo Joo, Dimin Niu, Xiangyu Dong, Guangyu Sun, Naehyuck Chang, and Yuan Xie, "Energy- and endurance-aware design of phase change memory caches", In *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 136-141, 2010.

[Z09] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang, "Energy reduction for STT-RAM using early write termination", In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ACM, New York, NY, USA, pp. 264-268, 2009.

[ZL09] Wangyuan Zhang and Tao Li, "Characterizing and mitigating the impact of process variations on phase change based memory systems", In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture,* ACM, New York, NY, USA, pp. 2-13, 2009.

[ZYZ09] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang, "A durable and energy efficient main memory using phase change memory technology", In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ACM, New York, NY, USA, pp. 14-23, 2009.