2016

Introducing memory versatility to enhance memory system performance, energy efficiency and reliability

Yanan Cao
Iowa State University

Introducing memory versatility to enhance memory system performance, energy efficiency and reliability

by

Yanan Cao

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Zhao Zhang, Major Professor
Morris Chang
Tien N Nguyen
Akhilesh Tyagi
Joseph Zambreno

Iowa State University

Ames, Iowa

2016

DEDICATION

I would like to dedicate this thesis to my parents and my wife Meng Yan without whose support I would not have been able to complete this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who have helped me for conducting my research and writing this thesis.

First and foremost, I would like to thank my adviser Dr. Zhao Zhang with deepest gratitude for his constant encouragement and patient guidance through my five years Ph.D study. Not only is he an excellent researcher, but he also understands how to guide Ph.D students. He has a keen eye on finding interesting research topics but gives me great freedom in choosing research topics according to my own interest, then patiently leads me to identify key challenges and solve them. More importantly, he teaches me the skill needed to deliver key research ideas in both writing and presentations. Apart from being an exceptional academic adviser in academy, he is also an amazing mentor in my personal life. One of the greatest difficulties to pursuing Ph.D degree in Iowa State University is having to deal with several years of remote marriage. Fortunately, Dr. Zhang has been extremely supportive and flexible about time schedules that I could visit my wife in our difficult times, making our family life much easier and sweeter than it should have been. Without help and guidance from Dr. Zhang, I would have never made this far.

Then, I would like to send thanks to my wife Meng Yan for always being there for me with love, understanding and support. Especially that even she knows me studying in Iowa means being separated from each other for a long time, she still supported my decision fully. I know how hard the decision is for her to make. Yet she did it with no complaint. Without everything she has done for me, I would not even have a chance to start my Ph.D study.

I would also like to thank the rest of my thesis committee - Professors Akhilesh Tyagi, Joseph Zambreno, Morris Chang, Tien Nguyen for their suggestions to improve this dissertation.

I thank Long Chen for cooperating with me on so many projects. He is always the first person for me to turn to when I am stuck with technical difficulties. His knowledge in computer

architecture area is a treasure I can never exhaust. I also thank Sparsh Mittal for giving precious advice on my early research projects. Additionaly, I send my thanks to Pei Zhang who helped me with a lot of paperworks.

I would also like to thank Manoj Patel and Brian Poynor, who are my current supervisors at Google. Aside from being supportive at work, they are very flexible with work schedules, allowing me to travel back to our university and give seminar presentation and defend my thesis. Besides, I send my gratitude to Alex Joly and Marc Blackstein, who are my mentor and supervisor during my internship at NVidia. They guided me through my first industry work experience and shared a happy time with me.

I would additionally like to thank all my friends, Cheng Chi, Kai Zhu, Yue Zhao, Zhang Zhang, Rong Gao, Enlai Xu, Jie Sun, Yuan-yuan Zhao and so many more who shared happy memories with me. You made my life in Ames colorful.

ABSTRACT

Main memory system design is facing increasingly high pressure from the advances of computation power scaling. Nowadays memory systems are expected to have much higher capacity than before. However, DRAM devices have limited scalability. Higher capacity usually translates to proportional hardware cost and power consumption. Memory compression is a promising technology to contain those increases. Previous memory compression works are generally based on rigid data layout which limits their performance. We thus propose Flexible Memory which supports out-of-order memory block layout to lower compression-related overhead and improve performance.

Besides, the cost of memory reliability also increases with capacity growth. Conventional error protection schemes utilizes Hamming-based SECDED code that comes with 12.5% capacity and power overhead of entire memory system. However, it may not be necessary to protect a whole memory system because some data may not be critical or sensitive to memory errors. Memory capacity and power used in protecting those data are almost wasted. Therefore, Selective Error Protection (SEP) can be used to lower the cost and power of large scale memory protection. The method to select critical data and non-critical data has been proposed before, however a memory system design to support its partitioned memory is challenging and does not exist at that time. Therefore, we propose a memory system design that has the capability to maintain two or more partitions with different layout in main memory at the same time. This design makes SEP schemes a complete practical design.

Even with selective error protection, supporting memory reliability is still hurting the scale of memory capacity. Fortunately, memory data has been proved to be very compressible. Most common applications are expected to free up enough space that can be used to store their own ECC code. For these applications, memory reliability incurs very low space and power overhead. However, combining ECC and memory compression is not trivial. It is difficult

to achieve high percentage of coverage over entire memory when compressibility of different memory blocks varies a lot. We thus introduce Flexible ECC that is based on Flexible Memory to allow easier ECC code placement. When a block has more choices to store its ECC code, it is more likely to be covered by ECC. With Flexible ECC, a larger portion of memory can be covered by ECC codes whose storage overhead is lowered by memory compression.

CHAPTER 1. INTRODUCTION

Being straightforward, simple and effective, DRAM-based Traditional Memory Design has long been de facto standard for main memory system. During the time in whih the speed gap between processor and memory device was narrow enough, traditional memory was capable of handling demand of memory throughput. However, computation power of microprocessors roars, bringing much heavier load on memory traffic and raising the bar of required memory performance [61] such that DRAM-based memory struggles to meet. Memory systems are stressed in many cases. Traditional memory design is starting to hit its limits in terms of capacity, bandwidth, power efficiency and reliability. In fact, the performance of memory system has become limiting factor in many computing systems.

One of the most prominent demands toward a memory system is about providing **large capacity**. Today's applications store more data in memory than ever. Reasons behind this phenomenon include richer features and higher quality service of these applications.

We can take commonly seen email client as an example. Email client used to be very lightweight command-line based application, with only simple features like receiving and sending plain emails. However, today's local (not including web-based) email clients have more features than those. Many of them manages user's contacts, support rich format in email content, provide embedded email search engine and so on. The same trend can be seen in many other applications as well. The working data set of today's applications are increasingly larger. This time we can take video player as an example. Limited by many factors, including network speed, screen resolution, etc., the resolution of videos used to be low, like 800 x 600. However, when display technology improves and network speed dramatically rises, it is common for a video to be in 3840 x 2160, or so called 4K UHD resolution, which contains 17.28 times as many pixels as 800 x 600 does. Other than applications requiring more capacity because of

their nature, it is a common for a programmer to use more memory to improve algorithm performance. Such technique could speed up programs by a few order of magnitudes.

If capacity demand can not be met by main memory, widely employed Virtual Memory scheme [12] has to swap data in and out between main memory and main storage (usually hard disks) in order to create the illusion of large memory space and prevent these demanding application from failing due to insufficient memory. However, access speed of main storages are several orders of magnitude lower than main memory, making each swap a considerable performance burden.

Not only do computing systems keep more data in memory, they also require significant **higher memory bandwidth**. Memory bandwidth is defined as number of data bytes a memory system can deliver in certain amount of time. Without enough memory bandwidth, processors would starve for lack of data and stall until its requested data is served. As multi-core systems become common, more programs and threads are running simultaneously in a single computer. Even if each program manages to keep bandwidth requirement stable, the overall pressure on memory system would still be higher than before. In reality, many programs are increasing their bandwidth demand, like video players described above, which also increases bandwidth requirement by a considerable amount.

DRAM-based memory system improves its own bandwidth via increasing working frequency of its data bus and utilizing DDR (Double Data Rate) technology to transfer two trunks of data instead of one in a single cycle. Data bus clock frequency has been increased from 400MHz to 1066MHz during the past years. However, further dramatically improving working frequency in impractical because of physical limits.

If we call energy as the currency a computing system uses to "buy" more performance, energy is the price. A system must have **great energy efficiency** so that running same workload costs less when compared to a system with poorer energy efficiency. This price is explicitly reflected in both energy bill and heat emission. The performance of smaller personal devices, like smartphones and tablets, is greatly constrained by their battery life. Unfortunately, as the most popular technology that most main memory systems are based on, DRAM, as its name implies, relies on constantly refreshing its data cells to retain data with a short interval.

Simply keeping data in DRAM already costs a considerable amount of power. When memory capacity increases, the situation would only get worse. For larger-scale computers, energy efficiency is an even more serious problem than battery life. Back in 2006, data centers in US are reported to consume 1.5% of total US power [14]. In 2013, those data centers in U.S. alone are reported to consume an estimate of 91 billion kilowatt-hours of electricity. It is also projected that by the time of 2020, electricity consumption may go up to 140 billion kilowatt-hours, incurring about $13 billion per year monetary cost. Therefore, it is critical to reduce energy consumption of data centers. Memory system takes a considerable percentage in above numbers. It has been reported that on IBM eServer, main memory alone consumes as much as 40% [35] of the total system energy.

A DRAM model **with perfect reliability** would be able to keep its data intact over time without error until the data is overwritten. And many programs are built upon this ideal model assumption. However, this model is not necessarily accurate. Like any other logical circuit, DRAM-based main memory is also vulnerable to logic errors. Because of this, DRAM error can cause serious consequences. Memory errors can easily cause operating system or individual program crashing or Silent Data Corruption [15], which is even worse than crashing in many cases because it causes wrong data to commit and keep propagating without even being noticed. Previously, memory errors are considered as rare events caused by random events like high energy particle collision or electrical fluctuations. However, recent studies [52, 22] show otherwise. They reported a memory error rate of 25,000∼70,000 FIT (Failures In a billion-hour operating Time) per Mbit of capacity. This is equivalent to around 1.5 DRAM errors in a 8 GB DRAM module. With such high error rate, it is hard to trust its output without proper protection.

1.1 Challenges and Proposed Solutions

The most important challenge a memory system faces is increasing capacity, bandwidth and improve reliability without causing significant overhead in terms of storage and power efficiency.

First of all, we would like to meet high reliability requirement while keeping energy consumption caused by reliability improvement mechanisms. ECC (Error Correction Code) memory is

usually used for this purpose. A challenge in using and enhancing memory error protection, however, is to control the storage and energy overheads. The (72, 64) ECC scheme commonly used in ECC memory incurs a storage overhead of 12.5% and about the same ratio of memory energy overhead. Using stronger ECC may enhance error protection but may also increase the overheads. One approach to reducing the overhead of existing and future memory error protection is to use a SEP (Selective Error Protection) framework [38]. In that framework, the system provides a protected memory region and a non-protected memory region. Only the protected region uses ECC. The OS and compiler place selected memory data in the protected region by certain criteria, e.g. those of high access frequency as reported by a profiling tool. Application programmers may also provide input. Some data, e.g. the data used by audio, image and video processing, are insensitive to memory errors and thus can be put in the non-protected region so that overhead can be reduced.

An important issue in SEP is how to support two separate memory regions in memory system design. The previous work [38], which focuses on the SEP high-level framework, simply assumes to have the support. A straightforward but naïve design is to use two sets of memory channels and modules, one for ECC and one for non-ECC memory, and integrate them into a single physical memory address space. Such a design is not only costly but also inflexible: The size ratio of the protected and non-protected regions cannot be adjusted during runtime. Performance overhead and energy consumption may also increase because of the separation. It would defeat the purpose of SEP, i.e. to reduce the storage overhead and improve energy efficiency.

Thus, we propose an efficient memory system design to support a memory SEP system in Chapter 3. It is based on previously proposed Embedded ECC [67, 8]. Embedded ECC enables ECC protection on non-ECC memory. The ECC bits of an ECC word are embedded in the same DRAM rows (pages) with the data bits of the word. Such a storage layout minimizes the energy overhead from accessing the ECC bits, because no extra bank pre-charge or activation is needed. The complexity of embedded ECC is on memory address mapping, because now a DRAM row holds data and ECC bits for a non-power-of-two number of memory blocks (of cache block size). For example, if a DRAM page previously holds the data bits for 128 memory

blocks, it will hold data and ECC bits for only memory 112 blocks, assuming a conventional (72, 64) coding scheme with a 12.5% ECC storage overhead. Embedded ECC uses a Biased Chinese Reminder Mapping (BCRM) scheme, which uses efficient modulo operation to replace Euclidean division in the address mapping.

Second, increasing memory capacity is not as straightforward as in small-scale computers. DRAM, as the most commonly used main memory technology, is hitting its limits in scalability. It has become harder and harder to shrink DRAM cell size. Without downsizing its cells, adding more capacity would translate to linear increase in power consumption. And not only does it cost energy to read and write data from and to DRAM, simply keeping more data intact is already a high price to pay. This is because DRAM, as its name implies, relies on constantly refreshing its data cells to retain data with a short interval. Block-level hardware memory compression is promising to increase the effective memory capacity and bandwidth, as it reduces memory footprint and memory traffic [13, 48, 49]. It is unlike earlier hardware memory compression (e.g. MXT [57, 1, 58]) or OS-based compression, which increases memory traffic and therefore not suitable for memory-intensive multi-core applications. The approach only requires proper OS support and is transparent to application software.

The compressed memory approach also has its own challenges to overcome. First, address mapping from main memory address to memory device address is much more complicated. Conventional simple address mapping, which relies on uniform memory block size and simple, sequential layout, may not work in compressed memory. Second, there is a new scenario called fat write [33], which refers to the type of write that triggers storage expansion in compressed memory. When it happens, non-trivial operations are needed to make enough room for the new data. Such operation can be very costly as they may involve movements of multiple memory blocks. A mechanism has to be put in place to minimize number of fat writes and/or reduce overhead of each fat-write.

Therefore, we present Flexible Memory in Chapter 4, a new design of block-level hardware memory compression. Any memory compression scheme will complicate the layout of memory contents in the physical memory storage. While decompression latency can be minimized using recently proposed, fast decompression methods, locating the compressed block may potentially

incur high overhead to memory access latency. To help address this issue, our design uses a unique data structure called BMT (Block Mapping Table) and a set of supporting components in the memory controller. A BMT holds the location information of all memory blocks belonging to an OS page. The BMT structure is so designed that a single access to BMT is sufficient to locate any memory block in a page, and it is compact enough such that a BMT can be fetched into the memory controller easily. The memory controller embeds a small BMT cache to facilitate BMT access, which has high hit rate for the workloads we evaluated. Coupled with recently proposed, high-speed high-throughput Base-Delta-Immediate (BDI) compression [50] and Frequent-Pattern Compression (FPC) [4], the use of BMT and BMT cache eliminates the majority of time overhead in accessing a compressed memory block. Furthermore, the design optimizes the page-level organization to reduce page expansion from fat writes. Our evaluation shows that, on average, the design yields 1.5x improvement of memory capacity, 14% power saving, and 7.5% performance improvement in weighted IPC speedup.

We also find that memory compression can be a good match with memory protection schemes. Memory compression could provide free space to stored ECC, such that storage and energy cost to maintain ECC code in memory would be lower or even free. There have been some research works [54, 43, 31] that leverage the combination of compression and ECC to provide low-cost ECC protection. However, these works focus more on utilizing simple and effective memory layout to avoid compression overhead and optimizing either ECC coding or compression algorithm to improve protection coverage provided by compression. However, they miss the opportunity to utilize free space scattered in entire memory space because of having a rigid memory layout.

Therefore, Chapter 5 proposes Flexible ECC, a scheme that combines error protection with memory compression based on Flexible Memory. The key idea behind Flexible ECC is improving protection coverage by utilizing the flexibility provided by FM and provide flexible block-level layout and page-level layout designed to accommodate as many ECC codes as possible. This is a new direction on improving protection coverage and is fully compatible with compression algorithms designed towards high coverage and various types of ECC code. In our evaluation, Flexible ECC is able to greatly reduce number of compression exception compared

to state-of-the-art. We also found that combining ECC and compression might cause ECC weakening, if not handled well.

The rest of this dissertation is as follows: Chapter 2 gives an overview of memory system in modern computers, including their base technology DRAM, hierarchy and basics for accessing them. It also gives introduction to DRAM errors, related consequences and common ways to mitigate them. Chapter 3 presents Memory System Design for SEP to make original SEP complete and implementable. Chapter 4 gives design and implementation details of Flexible Memory compression scheme. Chapter 5 proposes Flexible ECC design to utilize innovative block and page-level structures to improve protection coverage in compressed and protected memory. Lastly, Chapter 6 concludes the works included in this dissertation and points out future research directions.

CHAPTER 2. BACKGROUND

2.1 Memory System Overview

Memory system is a vital part in modern computing systems. It usually consists of cache, main memory, main storage are shown as a pyramid in Figure 2.1. Higher components in the pyramid is usually more costly per bit but order of magnitude faster than the component at next lower level. Because of cost ratio, higher level components are a lot smaller compared to lower level components.

When processors request data, including both read and write request, component on higher level of the pyramid is accessed first. If data is not present there, next lower level is accessed until processor finds requested data at certain level. Therefore, higher level component holds more recent but smaller set of data and acts as cache to its next level.

Main memory stands between caches (L1, L2 or more levels) and main storage (hard-disk, flash, etc.). Its position in memory hierarchy makes it especially important from system performance's point of view. Typically, main memory holds both code and data that programs need to run. The source of these code and data can be lower level – main storage, like hard-drive disks. It is notable that main storage usually poses significant latency and bandwidth limit that it is order of magnitudes times slower than memory. For example, a DRAM access usually takes 10-100 nanoseconds, however, a hard-disk access could be several milliseconds. Having to access main storage for a program can mean great performance drop and ultimately cause visible delay or uncomfortableness to end users. Main memory, as a much faster cache for main storage takes the responsibility to prevent these cases from happening. Therefore, a good main memory design, including its hardware devices and architectural choices is critical to system performance. Another possible source of memory data is data generated or modified by the

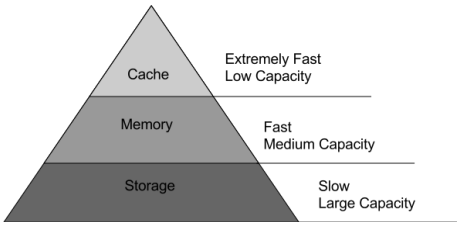


Figure 2.1: Memory hierarchy shown in a pyramid. Access speed is faster when going up the pyramid. However, capacity becomes smaller for components higher up in the pyramid too. (Some components like registers, are omitted as they are not involved in this study.

program during run time. This type of data sometimes is the only copy of valid data, which can be critical to guarantee correctness or a program. This poses high reliability requirement for main memory.

2.2 DRAM Technology

Dynamic RAM (DRAM) is the mainstream technology used to fabricate main memory devices. A DRAM device is constructed by connecting a matrix of DRAM cells.

Structure of a DRAM cell is shown in Figure 2.2. It is so-called 1T1C structure, namely one transistor and one capacitor. The transistor is used to switch on or off the access to bit capacitor when voltage on word line and bit line changes. The capacitor stores electrical charges, which is ultimately translated to a single bit binary value of either 0 or 1. Due to volatility nature of capacitors, electrical charges in capacitors fade over time. Due to the need to scale down DRAM cells, capacitor capacity is also reduced, making it easier to lose charge. Therefore, DRAM requires constant refresh operation to restore lost electrical charges over time. A refresh operation is essentially a coupled pair of read and write before information is lost completely. This naturally occurs energy cost simply to keep data alive.

When reading data stored in DRAM cell, word line and bit line must be set to appropriate voltage level to select a specific cell. Afterwards, charges in capacitor would flow to bit line, causing a slight fluctuation to be picked up by a sensor connected to bit line. Similarly, writing data is setting bit line voltage level to high or low in order to charge capacitor to desired voltage level.

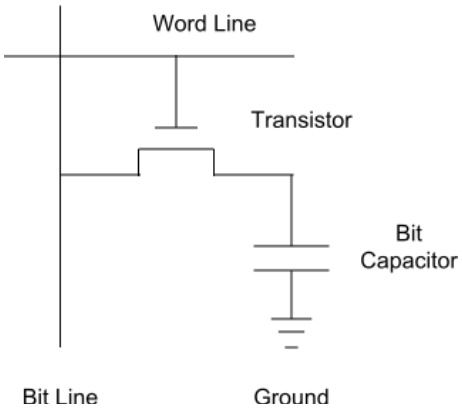


Figure 2.2: Structure of a DRAM cell. Each DRAM cell consists of one transistor and one capacitor.

2.3 DRAM Error

Ideally DRAM would retain data accurately until it is overwritten or DRAM itself loses power. In reality, DRAM can encounter errors at random, which compromises data integrity, even when data is not being accessed at the time of error happening.

DRAM error has been studied for decades [28, 40, 5, 6, 64, 55, 41]. It is now a growing concern as memory capacity and density scales. Recent studies on data-center computers have reported 25,000∼70,000 FIT (Failures In a billion-hour operating Time) per Mbit of DRAM systems [52], which means memory bit flips happen on daily basis. Another study [22] based on IBM Blue Gene (BG) super-computers also reports high error rate; for example, 167,066 FIT on BG/P computers at the Argonne National Laboratory.

To better understand these numbers, we can take 25,000∼70,000 FIT per Mbit as an example. On average, it can be translated to around 47,500 FIT per Mbit. Accordingly, if you own a personal device, like laptop, with 8 GiB memory capacity encounters approximately 39 DRAM errors in a 24-hour operation period.

2.3.1 Consequence of DRAM Error

DRAM errors can cause various consequences for a computing system. As stated before, DRAM holds run-time data for programs during execution. A DRAM error can cause instability to all programs, causing program crash or OS crash. Besides, memory errors can lead to silent data corruption [15], potentially causing wrong output from any program without noticing.

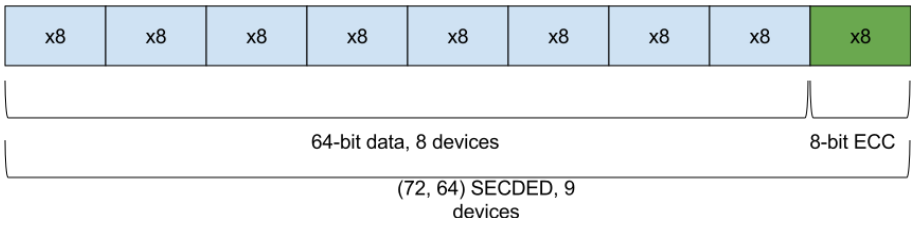


Figure 2.3: Nine-device SECDED Protection Memory Structure. During a device access, each device outputs eight bits. Out of nine devices, one delivers 8-bit ECC parity bits while other eight devices, together, deliver 64-bit data bits. Their output combines to be one (72, 64) ECC word.

What is even worse is that, memory errors can be taken advantage of by an adversary and exploited as a security vulnerability [17].

2.3.2 DRAM Error Protection

ECC memory conventionally uses Hamming [19] or Hsiao [21] based (72,64) SECDED (Single-bit Error Correcting Double-bit Error Detecting) code. A rank of ECC memory may use nine x8 [53] devices or eighteen x4 devices, with one x8 device or two x4 devices, respectively, dedicated to ECC bits storage. Figure 2.3 shows an example of nine-device SECDED protection memory. The notion of (72, 64) means symbol size is 72 bits, out of which, 64 bits are data. The rest 8 bits are parity bits. The storage overhead is 12.5% and the energy overhead is about the same. Chipkill Correct [11], used with x4 devices, further supports SDDC (Single Device Data Correction) that detects and corrects any number of errors from a single device. The storage overhead is 12.5% but the energy overhead is much higher. If in the future stronger error protection will be used, the storage overhead may become higher. Most desktop computers and mobile devices have not yet adopted any error protection because of the higher cost. In a word, protection schemes incur considerable overhead [10, 18, 27] in terms of both storage and energy consumption.

2.3.3 Variants of ECC Codes

SECDED is not only available in (72, 64) form. In fact, it can be designed according to different application scenarios. Given m-bit of parity bits available for SECDED code, its protection capability can be measured with maximum size of a symbol it can protect, namely

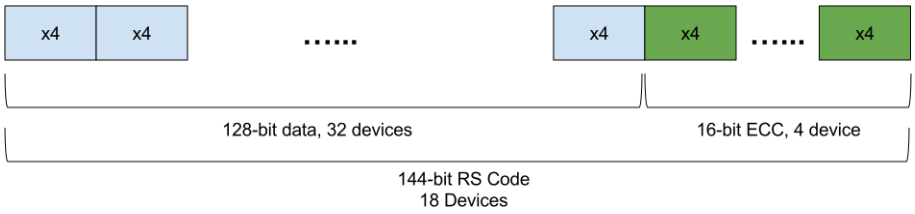


Figure 2.4: x4 Chip-kill correct memory Structure with (144, 128) Reed-Solomon code. For a device access, each device outputs 4 bit. Out of 36 devices, 32 of them deliver 128 data bits, while the rest 4 devices deliver 16 ECC parity bits. Together, they construct one (72, 64) ECC word.

n. Based on m, we can calculate n using formula shown below. According to this formula, we can get following table to show strength of SECDED code with different number of parity bits shown in Table 2.1.

$$n = 2^{(m-1)} - (m-1) \tag{2.1}$$

Table 2.1: Maximum number of data bits that can be protected with different number of parity bits.

Parity Bits	Maximum Data Bits
5	10
6	25
7	56
8	119
9	246

Besides variants of SECDED codes, there are stronger Error Correcting Codes than SECDED that provides the ability to correct and/or detect higher bits error, which of course comes with a higher cost in terms of parity bits. Another commonly seen code is Reed-Solomon (RS) code [51]. Chipkill is usually designed with RS code. x4 Chipkill puts x4 on each device, yet it still keeps ability to recover from single chip errors because of 4-bit symbol length RS code used.

2.4 DRAM Organization and Sub-ranked Memory

A DRAM rank is a set of memory devices that share same memory address/command bus and respond simultaneously to incoming commands [67, 60, 2, 3]. In DDR3 memory, a rank may consist of eight x8 devices, sixteen x4 devices, or four x16 devices to form a 64-bit data path. Sub-ranked memory design splits each memory rank into narrower sub-ranks,

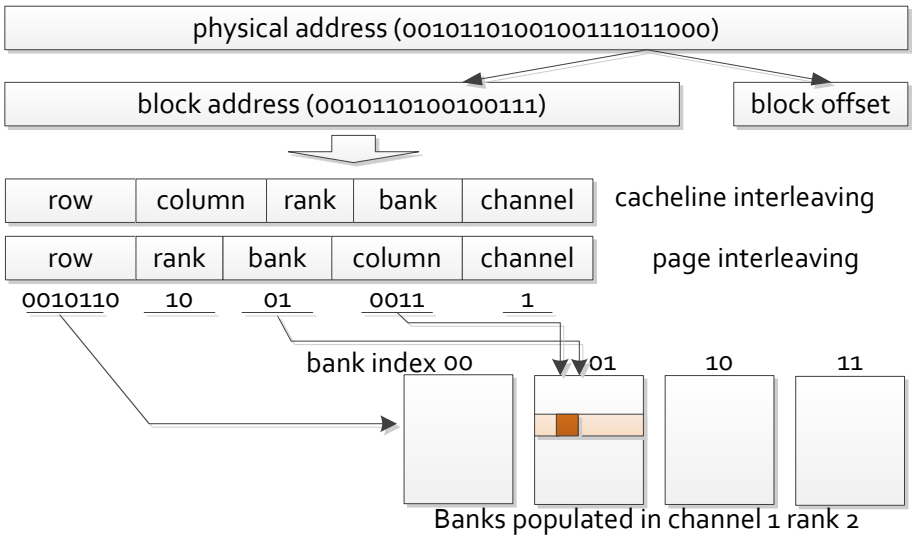


Figure 2.5: Representative memory address decomposition with cacheline- and page- interleaving schemes in DRAM memory systems. With page-interleaving address mapping, an artificial physical address "00101101001001110110000" is decomposed to row "0010110", rank "10", bank "01", column "0011" and channel "1" straightforwardly by splitting the sequence of binaries of the physical address. The last six bits "011000" of physical address are block offset assuming block size is 64-byte.

reducing the number of devices in a sub-rank. This change significantly reduces memory energy consumption, because fewer DRAM chips are involved in each memory access. For example, with a full rank of eight x8 devices, eight devices are involved in a memory access. Using 32-bit sub-ranking and 16-bit sub-ranking will reduce the number of involved devices to four and two, respectively. Sub-ranking may increase the data transfer time in DRAM access latency, but the increase is slight given the high bandwidth of today's DDR3 memory. Sub-ranking does not require any change of DRAM devices.

2.5 DRAM Address Mapping Basics

As shown in Figure 3.1, a DRAM device-level address mapping unit is required to map physical memory addresses to a designated DRAM region for data locating. It decomposes the physical address into multiple dimensions for locating the memory channel, rank, bank, row and column for the given access. Generally, the size of each dimension is power-of-two, which simplifies the address mapping between physical address and device address to simple bit decomposition. Conventionally, there are two address interleaving schemes, namely cacheline- and page- interleaving, and each has its variants. Figure 2.5 shows specific examples of those

two interleaving schemes[1] and the detailed address decomposition and layout for the page-interleaving example.

2.6 DRAM Accessing Basics

Commodity DDRx memory systems can have multiple channels, with multiple DIMMs (Dual Inline Memory Module) per channel, typically one or two ranks per DIMM, eight banks per rank, and a large number of rows and columns per bank. A memory request may be completed by three major commands, namely pre-charge, activation and column access (read/write operation). An activation command opens a DRAM row and pushes data from DRAM cells to row buffer. Each bank has a row buffer, which can be viewed as a single-line cache for DRAM data. The following column access command accesses the row buffer to fetch a block of data, usually 64 bytes for DDR3 memory. A row buffer is generally large for modern memory devices, and multiple column accesses can fetch data directly from same row buffer without opening the row again. This is called row buffer hit and it thus shortens DRAM access latency. Such a design is based on the presumption that there is available locality in the program and it can be captured in DRAM device-level through a proper address translation. As all rows in a bank share solely one row buffer, only one row can be opened in a bank at any given time. If a memory request hits another row of the bank, it incurs a bank conflict and the opened row needs to close first by pre-charge command before opening new row. Bank conflicts increase memory access latency and reduce memory throughput.

There are two commonly used page policies: close-page and open-page. Close-page policy attempts to pre-charge the opened row after column access so that incoming requests to other rows can be served immediately. It targets to take advantage of memory access parallelism to improve DRAM performance. Open-page policy, however, maintains a row open after column access. It favors applications with high row buffer locality so that the following requests hit the opened row and can be accessed without opening the row again. In general, open-page policy is less energy efficient than close-page policy as it consumes more power to maintain a row open. However, open-page policy can be more efficient for some applications as fewer row

[1] We use the mapping schemes discussed in a book [26] and our discussions can be extended to other variants.

activations are required. These two policies are equally important in practice, our discussion and evaluation will cover both policies.

A data block stored in multiple DRAM cells is addressed by decomposing physical address to DRAM device level address. Generally, one of two major address mapping schemes is optionally applied, namely cacheline-interleaving and page-interleaving. Cacheline-interleaving evenly distributes memory requests to DRAM channel, DIMM, rank and bank for high serving parallelism. While page-interleaving places nearby requests in the same DRAM row for locality. Cacheline- and page- interleaving schemes generally cooperates close- and open- page policy, respectively, to serve memory request efficiently. Close-page policy pre-charges an opened DRAM row after a memory request while open-page policy maintains the row opened attempting serving other requests hitting this row. The properties of two address mapping schemes are equally important in practice and they are opted dependent on real system requirements.

In commodity DDRx memory systems, cacheline-interleaving and page-interleaving address decompositions are designed to work with close-page and open-page policies, respectively. Cacheline-interleaving scheme distributes memory requests evenly among memory units for a high parallelism while page-interleaving scheme clusters nearby requests on same page for high row buffer locality. These two major properties are equally important in practice. We therefore devise different address mapping schemes in SEP to exhibit these properties to favor system requirements.

CHAPTER 3. MEMORY SYSTEM SUPPORT FOR SELECTIVE ERROR PROTECTION

Memory error protection is increasingly important as memory density and capacity continue to scale. This paper presents a memory SEP (Selective Memory Protection) design that enables SEP for commodity memory modules, with no change to the modules or memory devices. Memory error protection is provided through Embedded ECC, a recently proposed, energy-efficient ECC memory organization. The memory SEP design splits the physical memory address space into two memory regions of adjustable sizes, one with error protection and one without. With this support, the OS can adjust the size ratio of the protected region and non-protected region based on the needs of applications. In this scheme, the mapping from a physical memory address to memory device addresses is no longer power-of-two based. New and efficient address mapping schemes based on the Chinese Remainder Mapping are proposed to avoid the use of complex Euclidean division. The simulation results show that the memory SEP design may retain memory performance and cut memory power increase, while providing the ECC protection to commodity memory modules.

3.1 Introduction

Memory error protection is increasingly important as memory cell density and capacity scales. ECC (Error Correction Code) memory is usually used for this purpose. A few recent field studies on data center computers have reported that DRAM memory error rates are surprisingly higher than previously reported [52, 22]. Furthermore, memory errors are shown to have high correlation, which means simple memory error protection may not be as effective as previously thought. A challenge in using and enhancing memory error protection, however,

is to control the storage and energy overheads. The (72, 64) ECC scheme commonly used in ECC memory incurs a storage overhead of 12.5% and about the same ratio of memory energy overhead. Using stronger ECC may enhance error protection but may also increase the overheads.

One approach to reducing the overhead of existing and future memory error protection is to use a SEP (Selective Error Protection) framework [38]. In that framework, the system provides a protected memory region and a non-protected memory region. Only the protected region uses ECC. The OS and compiler place selected memory data in the protected region by certain criteria, e.g. those of high access frequency as reported by a profiling tool. Application programmers may also provide input. Some data, e.g. the data used by audio, image and video processing, are generally insensitive to memory errors and thus can be put in the non-protected region to reduce protection overhead.

An important issue in SEP is how to support two separate memory regions in memory system design. The previous work [38], which focuses on the SEP high-level framework, simply assumes to have the support. A straightforward but naïve design is to use two sets of memory channels and modules, one for ECC and one for non-ECC memory, and integrate them into a single physical memory address space. Such a design is not only costly but also inflexible: The size ratio of the protected and non-protected regions cannot be adjusted during runtime. Performance overhead and energy consumption may also increase because of the separation. It would defeat the purpose of SEP, i.e. to reduce the storage overhead and improve energy efficiency.

In this study, *we propose an efficient memory SEP mechanism to support a memory SEP system*. It is based on previously proposed Embedded ECC [67, 8]. Embedded ECC enables ECC protection on non-ECC memory. The ECC bits of an ECC word are embedded in the same DRAM rows (pages) with the data bits of the word. Such a storage layout minimizes the energy overhead from accessing the ECC bits, because no extra bank pre-charge or activation is needed. The complexity of embedded ECC is on memory address mapping, because now a DRAM row holds data and ECC bits for a non-power-of-two number of memory blocks (of cache block size). For example, if a DRAM page previously holds the data bits for 128 memory

blocks, it will hold data and ECC bits for only memory 112 blocks, assuming a conventional (72, 64) coding scheme with a 12.5% ECC storage overhead. Embedded ECC uses a Biased Chinese Reminder Mapping (BCRM) scheme, which uses efficient modulo operation to replace Euclidean division in the address mapping.

The proposed SEP mechanism partitions DRAM rows using a new, extended version of BCRM. The DRAM rows in the memory system are partitioned into two consecutive regions, one without ECC and one with embedded ECC. The boundary of the two regions is adjustable through the OS. The row-based partitioning makes SEP feasible for small and non-ECC memory systems of one or two memory ranks, e.g. those in laptop, desktop, and mobile computers. The SEP mechanism may also be revised for server memory systems, which use ECC memory or Chipkill Correct [11] memory, but this paper does not explore it. This mechanism makes two memory pools available to the OS, one with ECC protection and one without, and the sizes of the two pools are adjustable. Combined with the previously proposed SEP framework, it will enable a true SEP memory system.

Address mapping is a focus of this paper because the design has to use an extended version of the BCRM. In the original BCRM, the modulo operation uses a fixed divisor of 7, for which an efficient logic implementation is known. In the proposed SEP, the modulo operation of the BCRM has to use the divisor as a parameter that changes with the size of protected region. We call this new address mapping *parameterized BCRM*. We have carefully studied the logic design of parameterized BCRM and have found an efficient solution. We also present a new design called *Segmented BCRM* that further reduces the implementation complexity.

To summarize, we have made the following contributions in this paper:

- Based on Embedded ECC, we have designed a memory SEP mechanism that works on non-ECC DRAM modules, which are commodity DRAM devices and do not require higher-cost ECC DRAM modules.

- We use parameterized BCRM to implement the address mapping of the proposed SEP, and give an efficient logic design of the parameterized BCRM. We further propose a more efficient design called Segmented BCRM.

- We have fully evaluated the impact of the SEP design on memory system performance and power through simulation. The experimental results show that the SEP design retains system performance, reduces power consumption and meanwhile enhances system reliability.

The rest of this paper is organized as follows. Section 3.2 introduces the background and related work of the study. Section 3.3 examines the existing SEP framework and shows the design challenges. Section 3.4 presents the address mapping scheme to support SEP. Section 3.5 describes the experiment setup and Section 3.6 presents the simulation results. Finally, Section 3.7 concludes the paper.

3.2 Background and Related Work

3.2.1 CRM Address Mapping

Existing state-of-the-art CRM-based mapping is introduced in E^{3}CC based on Chinese Remainder Theorem [16] for coprimed memory system. It is a mapping from a physical address $d \in [0, RC-1]$ to a position in 2D array $L_{R\times C}$ represented by following formula[1]:

$$r = d \bmod R, c = d \bmod C$$

, in which r and c are row and column indexes of the array and $r \in [0, R-1], c \in [0, C-1]$. Given that R and C are coprime, CRM is a one-to-one mapping such that every physical address corresponds to one and only one position in the array.

CRM scheme is based on modulo operation only, which can be executed efficiently as shown in a previous work [56]. Table 3.1 shows an example layout using CRM with 8 rows and 7 columns.

3.2.2 Related Work

Memory error protection has been widely studied [64, 65, 41, 38, 8, 31]. Virtualized ECC [64, 65] proposes to maintain ECC in memory data space and relies on last level cache to

[1]The notations are different from those in paper [16].

Table 3.1: An example layout of CRM with $R = 8, C = 7$. An address d is mapped to exactly one pair of integer $\langle r, c \rangle$.

r/c	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	**0**	8	16	24	32	40	48
1	49	**1**	9	17	25	33	41
2	42	50	**2**	10	18	26	34
3	35	43	51	**3**	11	19	27
4	28	36	44	52	**4**	12	20
5	21	29	37	45	53	**5**	13
6	14	22	30	38	46	54	**6**
7	7	15	23	31	39	47	55

Table 3.2: An example layout of BCRM with $R = 6, C = 8$. $GCD = 2, T = 3$. Each cell maintains $n \rightarrow |2n : 2n{+}1|$. n means the super-address mapping to super-column; $2n : 2n{+}1$ are the corresponding normal addresses mapping to normal columns.

r/t → c	**0** → **0:1**	**1** → **2:3**	**2** → **4:5**
0	**0** → **0:1**	16 → 32:33	8 → 16:17
1	9 → 18:19	**1** → **2:3**	17 → 34:35
2	18 → 36:37	10 → 20:21	**2** → **4:5**
3	3 → 6 :7	*19* → *38:39*	11 → 22:23
4	12 → 24:25	4 → 8 :9	20 → 40:41
5	21 → 42:43	13 → 26:27	5 → 10:11
6	6 → 12:13	22 → 44:45	14 → 28:29
7	15 → 30:31	7 → 14:15	23 → 46:47

cache unused ECC to reduce memory traffic. The design is flexible and efficient compared to conventional ECC memory.

Frugal ECC [31] and COP [43] are two memory protection schemes based on compression. However, they are deliberately different. COP [43] exploits the possibility to distinguish a compressed ECC word from an uncompressed word without help of identifying flag. While, Frugal ECC [31] focuses on designing compression algorithm to improve protection coverage of compressed memory.

Archshield [41] proposes an architectural-level framework to protect fabrication faulty cells caused by extreme scaling of DRAM. Embedded ECC was first discussed as a generic idea in Mini-Rank [67] and later fully developed in E^{3}CC [8]. All those studies apply error protection to entire memory.

A recent study [38] proposes SEP for low-cost memory protection, which is closely related to our work. The details have been discussed in Section 3.1. Other works [7, 34] also discuss selective data protections. By comparison, this work focuses on device-level design issues,

mainly memory address mapping to support SEP with dynamic protection, which has not been fully studied before to the best of our knowledge.

This work is partially motivated by Embedded ECC, which stores ECC bits with data bits in the same DRAM row. It is designed for narrow-ranked low-power DRAM memories for both reliability and energy efficiency. It also works for full-rank non-ECC memory, which is the context of this paper. Because of ECC embedding, the effective memory capacity is no longer power-of-two. The authors therefore propose Biased Chinese Remainder Mapping (BCRM) for efficient device-level memory address mapping without using division. BCRM is used for the entire physical memory address space. This work faces a different problem, namely how to partition the address space into two regions, each with efficient device-level memory address mapping. The two regions coexist in the same memory system and the partitioning boundary may shift to vary the sizes of two regions, so we proposed parameterized BCRM to accommodate this change. BCRM can be viewed as a particular case of parameterized BCRM. We also propose another scheme called Segmented-BCRM that further reduces the cost with a non-continuous memory address space design.

3.3 Memory SEP Mechanism and Address Mapping

We first discuss the existing work on SEP (Selective Error Protection) framework and then present our memory SEP mechanism and address mapping. Mehrara et al. [38] propose Selective Error Protection (SEP) and show in detail that SEP is reasonable in balancing ECC power overhead and reliability requirement. Their study explores OS policies to utilize the SEP framework. Their work also presents a high-level structure to support SEP. In the structure, the memory space is partitioned into two regions, one protected by ECC and the other not. Memory traffic is routed to a region according to its sensitivity to memory errors. The memory controller maintains the address boundary of the two regions in a register. If the address of coming memory request is greater than the marked address, it is protected and its data transmits via ECC circuitry for integrity check between memory controller and DRAM devices. Otherwise, the ECC circuitry and logic is bypassed. Although their study outlines the SEP framework, it lacks a specific solution of providing a memory SEP mechanism that supports

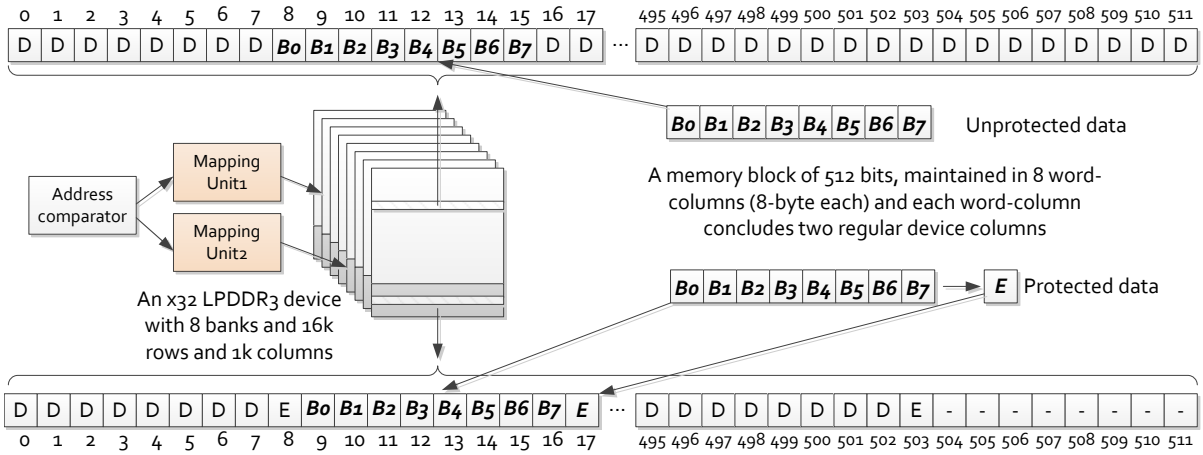


Figure 3.1: An example row-index based partitioning for selective protection and data/ECC layout inside an x32 mobile LPDDR3 DRAM device with 16k rows and 10k columns. The dark gray rows across all banks are protected by ECC. D represents a generic data word-column and E represents a generic ECC word-column. A word-column is eight bytes, or two regular DRAM device columns in a row. B_i represents a 64-bit subblock of a memory data block. There are eight data word-column followed by one ECC word-column in error protected region, assuming conventional (72,64) SECDED is applied.

those two regions. This is exactly what we are focusing on in this study, in order to make SEP framework a complete design.

3.3.1 Memory SEP Mechanism Based on Row Partitioning

A DDRx memory consists of a number of memory channels, DIMMs, ranks, and banks, with a large number of rows and columns in each bank. Although the partitioning can be done by any of those dimensions, row-based partitioning is the only practical solution for small-scale memory systems. In row-based partitioning, the memory system is viewed as an array of consecutive DRAM rows, and the partitioning is to split the array into two sub-arrays of consecutive rows. We use the first sub-array as non-protected region and the second as the protected region; the reverse is also a valid choice. Partitioning by channel, DIMM, rank, or bank cannot support fine granularity of partitioning. Furthermore, it will limit memory access parallelism and thus degrade memory performance and energy efficiency. The related performance evaluation is done in Section 3.6.1. We have conducted experiments of bank-based partitioning in Section 3.6.1 and they show an average of 6.6% and 4.7% multi-core weighted IPC loss when reducing bank-level parallelism by half for cacheline- and page-interleaving, respectively. Lastly, partitioning

by column reduces row buffer locality, which could hurt system performance and does not have obvious benefit over row-based partitioning. Therefore, row-based partitioning is used in this design.

A general memory system consists of multiple DRAM channels. Multiple ranks can also populate in each channel[2]. A Typical DRAM rank is formed of eight banks, which are two-dimensional arrays of storage cells. All these dimensions, including channel, rank, bank, column and row, are options for memory space partitioning. In SEP, we turn to *row dimension* for partitioning for the following reasons. First, channel-, rank- or bank-based partitioning lack flexibility as number of those units can be limited in a system. A mobile memory system may merely have one channel or two ranks. Although in large-scale workstations or servers, there may be enough channels or ranks to provide fine granularity, we make no assumptions about memory configuration in this work.

Third, column-based partitioning is also inappropriate as it reduces effective row buffer size, likely penalizing system performance.

Row-based partitioning avoids all these drawbacks and preserves available parallelism and access locality. In all banks, higher-indexed rows are protected with ECC while lower-indexed rows are exposed to errors without protection. As either region is evenly distributed to all channels, ranks and banks, memory access parallelism is maintained.

3.3.2 Memory ECC Storage

The protected memory region in SEP requires ECC for data integrity checking and it thus introduces the problem where and how to store its ECC bits. Conventional ECC design deploying extra DRAM device for ECC redundancy is obviously inappropriate for partial protection. We adopt Embedded ECC [67, 8] to implement ECC protection for the protected region (called ECC region thereafter). The layout of SEP is shown in Figure 3.1. A memory data block is mapped to ECC or non-ECC region through an address comparator and two mapping units. In the non-ECC region, each 512-bit data block takes eight word-columns (eight-byte each)

[2]DIMM index is count into rank index thereafter in our presentation as DIMM may not be applicable in devices like smartphones.

or equivalently sixteen device columns (four-byte each) in an x32 device. In the ECC region, one 8-byte ECC chunk is placed in a column following every eight data columns, assuming the conventional (72,64) SECDED is used. The last eight columns are leftover columns which are not utilized in Embedded ECC. Fortunately it is only a small percentage of memory storage. Its design is discussed in more detail in Section 3.4.

3.3.3 Address Mapping

As discussed in Section 2.5, conventional device-level address mapping is simple because the sizes of all DRAM dimensions are power-of-two. However, in SEP, the number of rows and columns of either region may be non-power-of-two due to partitioning and ECC embedding, which complicates device-level address mapping. Note that the address mapping units must be flexible for the OS to adjust the boundary of the two regions. Take the address decomposition in Figure 2.5 as an example and assume there are only 11 columns instead of 16 in one region, the physical address thus cannot be decomposed to column, bank, rank and row indexes by splitting the address bits. Without an effective solution of address mapping, the SEP may not be implementable. A straightforward solution is to use division operation in the mapping function, as the quotient and remainder of integer division operation can be used as the row and column indexes. However, division operation would introduce significant overhead in hardware cost and memory access latency, as well as limit memory throughput. The latency of integer division is significant in modern processors [8]; for example, the IDIV (Integer Division) instruction takes 38 to 123 cycles for 64 bit unsigned integer in Intel IA-32 and x86-64 architectures [25]. Additionally, division operation is hard to pipeline and thus will limit memory throughput unless multiple division units are used. The performance impact evaluation will be presented in Section 3.6.

The BCRM scheme proposed in Embedded ECC [8] uses modulo operation to replace division operation. The modulo operation in the BCRM scheme uses a fixed divisor of seven, for which a fast logic implementation is known [56]. In general, if the divisor is fixed and is a relative small number, then the modulo operation is known to have a fast logic implementation. The scheme cannot be directly used in SEP, however, because the divisor would have to vary

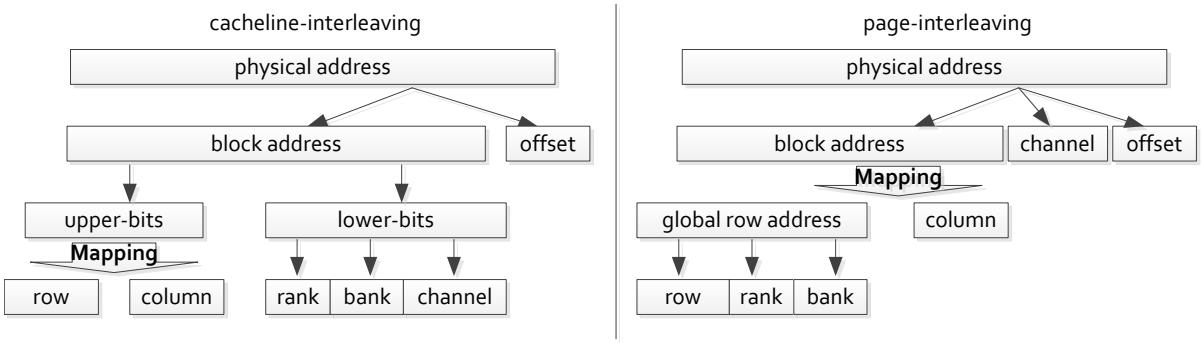


Figure 3.2: Address decomposition procedure. Mapping function used here is 2D, while other arrows are simple bit decomposition.

as the boundary of the two memory regions shifts. We propose a *Parameterized BCRM* scheme to solve the issue, and further propose a more efficient scheme called *Segmented BCRM*.

3.4 Parameterized BCRM

3.4.1 Mapping Reduction

The mapping problem in SEP is finding a fast and efficient method to map a physical address to DRAM device address components, which requires a multi-dimensional mapping function as shown in Formula 3.1.

$$(Chn, Rnk, Bnk, Row, Col) = map\_func(addr_{phy}) \tag{3.1}$$

To simplify the discussion, the rank index is a global identifier that contains the DIMM index. For example, a memory system of two DIMMs and two ranks per DIMM is considered to have four global ranks. As discussed, the row and column can be non-power-of-two because of the partitioning and ECC embedding. The number of channels, ranks and banks are still power-of-two. Therefore, we can reduce required mapping function from 5D mapping to 2D mapping plus simple binary decomposition. Figure 3.2 shows how 2D mapping can be used to complete DRAM address mapping. The left diagram in Figure 3.2 shows the address mapping procedure of cacheline-interleaving using 2D mapping. After removing the six-bit offset (for 64-byte data block size) from physical address, the remaining block address is first decomposed to lower bits and the remaining upper bits. The lower bits cover rank, bank and channel indexes.

The upper bits are then decomposed to row and column indexes using 2D mapping. The right diagram of Figure 3.2 shows the procedure of address mapping for page-interleaving scheme. Column indexes are lower bits after removing channel and block offset in a block address in page-interleaving scheme. We can thus first apply 2D mapping to obtain column and global row indexes. The global row indexes serve as a unique row identifier across all banks and ranks. Then the last few bits of the global row indexes are split to rank and bank indexes. In a word, we only need a 2D mapping function as in Formula 3.2 to achieve non-power-of-two memory mapping.

$$(Row, Col) = 2D\_map\_func(addr_{phy}) \tag{3.2}$$

3.4.2 Existing CRM-based Address Mapping

BCRM (Biased CRM) is introduced in Enhanced Embedded ECC [8], while CRM mapping is based on Chinese Remainder Theorem for prime memory system [16], more details can be found in 3.2.1. CRM uses modulo operations to map a physical address $d \in [0, RC-1]$ to a tuple $\langle r, c\rangle$ in a 2D array with R rows and C columns by the following formulas[3]:

$$r = d \bmod R, c = d \bmod C \tag{3.3}$$

When R and C are coprime, CRM is a one-to-one mapping such that every physical address corresponds to one and only one position in the array.

BCRM revises the formulas for embedded ECC with a modern DRAM system to resolve two issues in CRM. First, CRM requires the number of rows and columns to be coprime. Second, CRM breaks the row-level locality as continuous addresses are mapped to different rows, which reduces row buffer hit ratio and degrades system performance. BCRM first groups n columns together to form a *super-column*, where n equals to GCD (Greatest Common Divisor) of R, C, such that number of rows and super-columns can become coprime when popular $(72, 64)$ ECC is used. Then CRM Formula 3.3 can be applied to get a one-to-one mapping. An example of such mapping is shown in Table 3.2. CRM guarantees that the mapping from super-address to array $L_{R\times T}$ is one-to-one. Then BCRM extend the super-addresses and super-columns

[3]The notations are different from those in paper [16].

Table 3.3: An example layout of BCRM with $R = 8, C = 6$. Addresses 0∼5 are highlighted.

r/c	0	1	2	3	4	5
0	**0**	**1**	**2**	**3**	**4**	**5**
1	18	19	20	21	22	23
2	36	37	38	39	40	41
3	6	7	8	9	10	11
4	24	25	26	27	28	29
5	42	43	44	45	46	47
6	12	13	14	15	16	17
7	30	31	32	33	34	35

to normal physical addresses and columns by adding an offset inside the super-column. The mapping is thus obtained by

$$r = d_s \bmod R; c = (t \ll log_2 g) + (d \bmod g)$$

The left shifting ($\ll$) is to extend a super-column to regular columns; $(d \bmod g)$ adds the offset of the regular column inside a super-column. Table 3.2 shows an example layout with $R = 8, C = 6$. Then, BCRM adds a bias factor to adjust the skewed row indexes in CRM. The final mapping is obtained by formulas:

$$d_s = d \ll log_2 g; \quad C_s = C \ll log_2 g$$

$$r = (d_s - d_s \bmod C_s) \bmod R; c = (d_s + (d \bmod g)) \bmod C_s$$

while $-(d_s \bmod C_s)$ is the bias factor and g is the GCD. Table 3.3 shows a BCRM example layout with $R = 8, C = 6$.

With cacheline-interleaving scheme as shown in Figure 2.5, rank, bank and channel indexes are lower bits and they are still power-of-two. The left diagram in Figure 3.2 shows the address mapping procedure of cacheline-interleaving using BCRM. After removing the six-bit offset (for 64-byte data block size) from physical address, the remaining block address is first decomposed to lower bits and the remaining upper bits. The lower bits cover rank, bank and channel indexes. The upper bits are then decomposed to row and column indexes using BCRM. In our example, GCD of row and column is 8 and the mapping is done by the following formula:

$$\begin{aligned} r &= (d - (d \gg 3) \bmod 7) \bmod (3 \times 2^{13}) \\ c &= ((d \gg 3) \bmod 7) \ll 3 + (d \bmod 8) \end{aligned} \tag{3.4}$$

The right diagram of Figure 3.2 shows the procedure of address mapping for page-interleaving scheme. As shown in Figure 2.5, column indexes are lower bits after removing channel and block offset in a block address in page-interleaving scheme. We thus first apply BCRM to obtain column and global row indexes. The global row indexes act as a unique row identifier across all banks and ranks. It is thus 3×2^{17} in total. The mapping can be done by formula 3.4 after replacing 2^{13} with 2^{17}.

$$\begin{aligned} r &= (d - (d \gg 3) \bmod 7) \bmod (3 \times 2^{17}) \\ c &= ((d \gg 3) \bmod 7) \ll 3 + (d \bmod 8) \end{aligned} \tag{3.5}$$

Then the last four bits of the global row indexes are split to rank and rank indexes.

3.4.3 Parameterized BCRM and Hardware Cost

The original BCRM has a simple implementation in embedded ECC because the numbers of row and column are fixed, allowing a highly specialized and optimized modulo design. SEP, however, requires adjustable protection size. We have extended the original BCRM into *parameterized BCRM* that allows the divisor to be a configurable parameter. Parameterized BCRM has higher hardware complexity, which has to be dealt with carefully. Most hardware complexity of BCRM comes from the modulo operation. Consider a generic modulo operation $v \bmod m$. Study [56] presents an efficient logic design of modulo operation when divisor is a preset constant number, based on the following property:

$$v \bmod m = \sum_{i=0}^{n-1} (v_i \cdot (2^i \bmod m)) \bmod m \tag{3.6}$$

where v_i is the ith bit of v in binary form and m is a fixed small integer.

In a word, a modulo small number operation can be done by calculate the number of bits in a binary representation as the modulo is periodic as we can see in Table 3.4.

Therefore, the mod operation finally converts to count the number of bits in a binary operation, which can then be translated to a series of table lookup operations. With help of RAMs, these table lookup operations can be done efficiently.

Table 3.4: 2^n modulo small interger exhibits periodic behavior.

$2^n \bmod 3 = 1, 2, 1, 2, ...$ for $n = 0, 1, 2, 3, ...$
$2^n \bmod 5, = 1, 2, 4, 3, 1, 2, 4, 3....$
$2^n \bmod 6 = 1, 2, 4, 2, 4, 2, 4, ...$
$2^n \bmod 7 = 1, 2, 4, 1, 2, 4, ...$
$2nmod9 = 1, 2, 4, 8, 7, 5, 1, 2, 4, 8, 7, 5, ...$

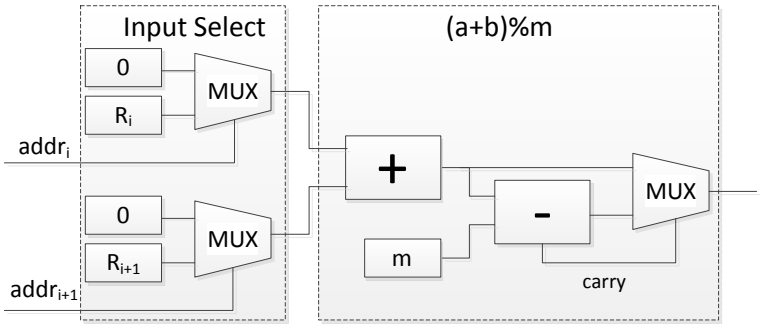


Figure 3.3: Part of modulo logic. $addr_i$ represents ith bit from input address. R_i is register holding pre-computed value corresponds to $2^i \bmod m$ in Formula 3.6. m is divisor. Input select takes address and uses each bit to select a pre-computed R_i. $(a+b)\%m$ is a modulo-sum block which is duplicated in a tree-like structure; duplicated logic is not shown in this figure.

Other previous work on modulo design focus on special values of m such as $m = (2^n \pm 1)$ [29, 46, 69]. Since SEP requires flexible modulo operation, none of these modulo algorithm fit in SEP's use case.

To our best knowledge, the design presented below is currently state-of-the-art divisor-parameterized generic modulo operation logic design. This design comprises of first-stage input select logic and a series of modulo-sum logic blocks in binary tree structure. In each of modulo-sum block, there are two $log_2 m$-bit registers R_i and R_{i+1} to hold pre-computed values of $2^i \bmod m$, one multiplexer and two $log_2 m$-bit adders. Both input select and modulo-sum are shown in Figure 3.3. In order to process a k-bit input, there should be $k-1$ modulo-sum blocks. In other words, cost to implement modulo-sum is proportional to $log_2 m$ and k.

In SEP, m can be controlled to be a small integer but its value may vary. We give our own design as follows. The modulo logic comprises of a first-stage input select logic and a series of modulo-sum logic blocks, shown in Figure 3.3. The modulo-sum logic blocks are duplicated, forming a binary tree structure (not shown in the figure). For each bit of input $addr_i$, the input select logic uses one k-bit register R_i holding pre-computed value of $2^i \bmod m$ and one

Table 3.5: Segmented BCRM achieves equivalent mapping performance to BCRM; width of CRM input is reduced by one bit

(a) BCRM; $R = 8, C = 7$

r/c	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	49	50	51	52	53	54	55
2	42	43	44	45	46	47	48
3	35	36	37	38	39	40	41
4	28	29	30	31	32	33	34
5	21	22	23	24	25	26	27
6	14	15	16	17	18	19	20
7	7	8	9	10	11	12	13

(b) Segmented BCRM $R = 8, C = 7, S_{seg} = 32$; two segments

r/c	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	21	22	23	24	25	26	27
2	14	15	16	17	18	19	20
3	7	8	9	10	11	12	13
4	32	33	34	35	36	37	38
5	53	54	55	56	57	58	59
6	46	47	48	49	50	51	52
7	39	40	41	42	43	44	45

multiplexer. Each modulo-sum block uses one k-bit register m holding divisor, one multiplexer and two k-bit adders, where $k = log_2 m$.

The complexity of the modulo logic is a function of the address width. Assume that a system uses 46-bit physical address, and 1/64 adjustment granularity is needed for the SEP. The modulo unit will use approximately 9,000 transistors. To our best knowledge, we are not aware of any existing design that is significantly better than ours for this set of design requirements.

3.4.4 Segmented BCRM

We further optimize the above design to reduce the hardware complexity. We have found a new design called *Segmented BCRM*, which uses a segmented physical address space instead

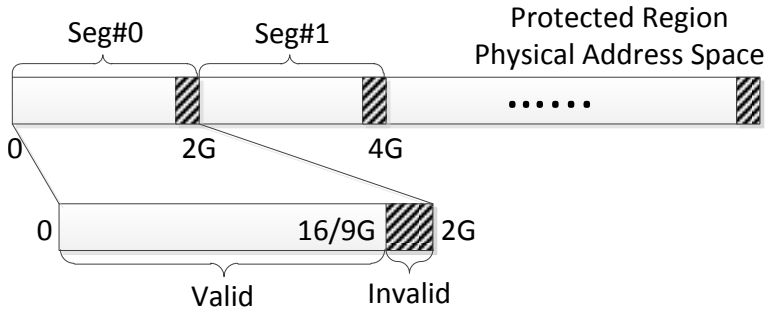


Figure 3.4: Address space layout under SBCRM. Assume segment size $S_{seg} = 2GB$, $(72, 64)$ ECC causes 2/9GB invalid addresses in each segment

of a flat space to reduce the complexity. With proper OS support, this change will not cause any performance loss. It works as follows. We extend the regional physical address space from $d \in [0, RC-1]$ to its next largest power-of-two address $d \in [0, 2^{log_2 RC} - 1]$. Then this extended address space is divided into segments with same segment size S_{seg}, which is a power-of-two design parameter. Table 3.5 shows an example. Table 3.5a shows BCRM mapping from $[0, 55]$ to 2D matrix with $R = 8, C = 7$, requiring 6-bit input to modulo logic. In Table 3.5b physical address range is extended to $[0, 63]$ and divided into two segments $[0, 31], [32, 63]$ assuming $S_{seg} = 32$ is used. BCRM now takes $log_2 S_{seg} = 5$-bit input to modulo logic instead of 6. In this example, segment index (id) is the MSB (Most Significant Bit) of the extended physical address, which can be obtained without complicated computation. The final mapped location $\langle r, c \rangle$ can be obtained by combining id and $\langle r', c' \rangle$ generated from BCRM within each segment.

Segmented BCRM can be expressed with following formulas:

$$R_s = S_{seg} \gg log_2 C'; id = d_s \gg log_2 S_{seg}$$

$$\langle r', c' \rangle = BCRM(d_s \& (S_{seg} - 1)))$$

$$r = (id \ll log_2 R_s) | r'; c = c'$$

R_s is a pre-computed power-of-two value indicating number of rows allocated to each segment. $S_{seg} - 1$ generates a binary mask to get lower bits from physical address d_s, which is the key part to limit input width into BCRM mapping. id is then shifted and combined with in-segment row r' to get global row r while column c is same as c'. All operations are simple bit operations.

Compared to BCRM, Segmented BCRM has same latency, row buffer locality but lower hardware complexity at the cost of not having a continuous physical address space. Note that it is addresses that are lost not actual memory capacity. Percentage of lost addresses is decided by choice of ECC. In previous example, $[28, 31]$ and $[60, 63]$ are invalid addresses. Figure 3.4 shows a realistic physical address layout within protected region under mapping of SBCRM. In the case of (72,64) SECDED code, one ninth of physical addresses are invalid, which is shaded part in the figure. Non continuous physical address space can be managed by the OS. Segment size S_{seg} is a key parameter. A smaller S_{seg} will lower the modulo complexity, but may limit the OS' ability to allocate very large and continuous memory space. After careful consideration, we find that using $S_{seg} \in [1GB, 4GB]$ is appropriate. These sizes can reduce modulo complexity to around 4,000 transistors, 55% lower than that of the BCRM. They also give OS enough freedom to allocate up to $32/9GB$ memory chunk that is continuous in physical address space, which is, in most cases, more than enough with the presence of virtual memory system.

This design limits maximum memory capacity a system can have. However, modern processors usually have the ability to support much more physical memory than needed. For example, intel I7 processor has 46-bit wide physical address [25], supporting 16TB memory, which is more than what most systems need.

3.4.5 Partition Choices and Protection Ratio

BCRM and Segmented BCRM are valid only when R, C are coprime (with or without super-column grouping). For ECC protection with $(72, 64)$ SECDED code and an adjustment granularity of 1/64 of total memory capacity (or total rows), 8 out of 65 choices of protection ratio, namely 7/64, 14/64, 21/64, 28/64, 35/64, 42/64, 49/64, 56/64, should not be used. The OS may use the other 57 choices of protection ratio. We believe this is sufficient for the purpose of SEP. The number of valid protection ratios may increase by using a finer granularity, if desired.

Table 3.6: Layouts of an S^2-CRM mapping before reconfiguration with $R = 6, C = 8$ and after reconfiguration adding one row to $R = 7, C = 8$.

(a) S^2-CRM; Before reconfiguration; $R = 6, C = 8$

row/col	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	24	25	26	27	28	29	30	31
2	8	9	10	11	12	13	14	15
3	32	33	34	35	36	37	38	39
4	16	17	18	19	20	21	22	23
5	40	41	42	43	44	45	46	47

(b) S^2-CRM; After reconfiguration. C =8 while R increases from 6 to 7

row/col	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47
6	48	49	50	51	52	53	54	55

3.4.6 Operating System Support

SEP demands some OS modification to support SEP memory system. OS needs to manage multiple separated memory spaces instead of one. We can safely assume OS, with moderate modifications, has the capability to handle this task. One of the straightforward approach is adding Protection Flags to memory management related data structures, so that OS can easily differentiate between protected memory and unprotected memory. More sophisticated and superior design may also be possible, but it is outside scope of this work, therefore we do not include the discussion here.

SEP (re)configuration support must be added to OS. This is needed at boot time and when SEP policy decides to adjust protection ratio. When reconfiguration happens, some registers in mapping unit need update to enable new mapping function. Moreover, data movement may be needed too, details of which is given in next section.

Overhead of data movement is determined mostly by memory utilization and frequency of reconfiguration. Fortunately, only the ordering of rows is changed while ordering of columns is

untouched. Therefore row-based data movement is most likely only bottlenecked by memory bandwidth. In an 8 GB DDR3-1600 memory system with 100% memory utilization, a pessimistic estimate of time is 1.9 seconds. when a reconfiguration is demanded, OS needs to read 8 GB data and write back same data, making a total of 16 GB. Assume OS suspends other activities when data is being moved, all available DRAM bandwidth can be used to transfer data. Bandwidth available for data movement is around 8.4 GB / Second, even if we take into account various DRAM timing restrictions that may limit effective bandwidth utilization to 70%. In this case, time overhead of data movement is only $16GB \div 8.4GB/S \approx 1.9Seconds$.

At time of protection ratio adjustment, mapping unit also needs to be reconfigured, especially its core part, modulo logic. Assume flexible design introduced in Section 3.4.3 is used, reconfiguration is nothing but updating registers like R_i and m shown in Figure 3.3 according to new divisor.

3.4.7 Various Error Protection Codes

Although previous case study use (72,64) hamming-based SECDED code, this framework can extend to many other error protection codes. From an architectural point of view, major differences of various codes is word size, which changes number of words a row can accommodate. Due to flexibility of CRM variants like S^2-CRM, we can find a perfect mapping for every combination of rows and columns.

Besides, the proposed address mapping schemes can be extended for memory systems with more than two regions for various error protections strategies. For example, a system can have three regions with one unprotected, one with SECDED protection and the remaining part with BCH DECTED (Double-bit Error Correcting Triple-bit Error Detecting), respectively. Such a partitioning further tunes the system in fine granularity for strong reliability while maintaining efficient energy consumption. The details of the extension is not presented and the address mapping is similar to that of two-region system.

3.5 Experimental Methodologies

We build a detailed memory system simulator for a x32 DDR3 memory system and integrate it into Marss-x86 [45], which is a cycle-accurate full system simulator for x86-64 architecture. In the DDR3 simulator, we integrate different address mapping schemes with both cacheline- and page-interleaving scheme. A set of real-world product technical specifications serves as configuration of DDR3 simulator, details can be found in MT41J256M8-32 Megx8x8Banks datasheet [39]. Table 3.7 and Table 3.8 show major parameters for the simulation platform and memory power calculator used in our experiments. We use sub-ranked memory scheme [67] because our framework uses ECC storage technique in E^3CC, which conserves more energy with sub-ranked memory.

We select benchmarks from SPEC CPU2006 [20] and run the simulation with both single-core and four-core configurations. For our proposed address mapping schemes, one extra memory cycle is added for each memory access. We also simulate division-based address mapping scheme, adding eight memory cycle latency [8]. Eight memory cycle is an optimistic division latency. It is equivalent to 32 processor cycles in our simulation setup, while an unsigned 64-bit integer division takes at least 38 cycles in Intel 64 and IA-32 architectures [25]. Division operation is difficult to pipeline and its throughput is usually limited. Experiments are also designed to measure performance penalty when there are limited number of dividers and division is not fully pipelined. We create checkpoints for Marss simulator after initialization and warm-up. After a warm-up period, we run the simulation for 300 million instructions in single-core experiments and 1 billion instructions in four-core experiments.

We do not simulate detailed SEP profiler or protection ratio decision logic since they are out of this study's scope. Therefore, reliability evaluation of SEP scheme is not included as it is mostly decided by high-level SEP profiler and policy, not by memory system support framework proposed in this work. In experiments where SEP logics are needed, we simply use arbitrary protection ratios, which should be able to represent typical usage scenarios.

Table 3.7: Major simulation parameters.

Parameter	Value
Processor	1 or 4 ooo cores 3.2GHz 14-stage pipeline 4-issue per core
Functional units	2 IntALU 4 LSU 2 FPALU
IQ, ROB and LSQ	IQ 64 ROB 128 LSQ 96
Physical registers	256 Int, 256 FP 48 BR, 24 ST
L1 caches (per core)	64KB Inst/64KB Data, 8-way, 64B line, hit latency: 3-cycle for Inst & Data
L2 cache (shared)	4MB, 8-way, 64B line, 13-cycle latency
DDR3 DRAM latency	DDR3-1600 11-11-11
DRAM Hierarchy	2 Channel, 1 DIMM per channel, 1 Rank per DIMM, 2 Sub-Ranks per Rank

Table 3.8: Major DRAM power parameters. They are taken from Micron datasheet [39].

Parameter	Value
Normal voltage (Vdd)	1.5V
Active precharge current (IDD0)	95 mA
Precharge power-down standby current (IDD2P)	12/35 mA
Precharge standby current (IDD2N)	42 mA
Active power-down standby current (IDD3P)	40 mA
Active standby current (IDD3N)	45 mA
Read burst current (IDD4R)	180 mA
Write burst current (IDD4W)	185 mA
Burst refresh current (IDD5)	215 mA

3.6 Experimental Results

3.6.1 Memory-Level Parallelism

We have excluded channel-, rank- or bank-based partitioning because they limit the choices of protection ratios, particularly for a relatively small memory system. They also limit memory-level parallelism and thus degrade performance, which is evaluated in this section. Assume that four of eight memory banks are configured with ECC protection, therefore a program using ECC memory has only four banks to access. In the row-based partitioning, the programs access all the eight banks. Figure 3.5 compares the performance of those workloads with 4-bank and 8-bank

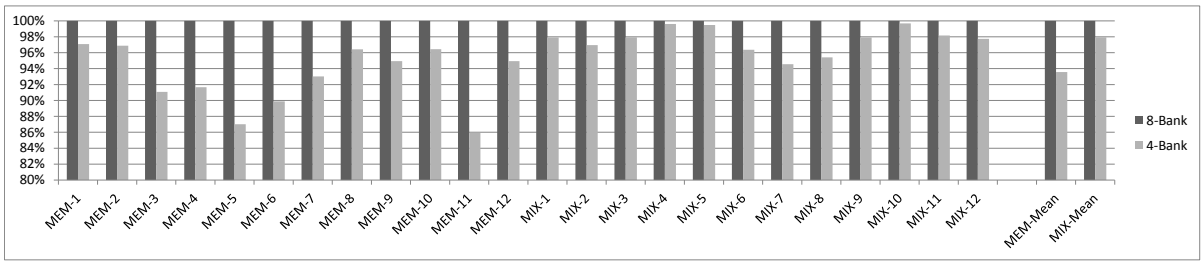
Table 3.9: Workload construction. Memory Intensive (MEM) workloads contains only those benchmarks whose MPKI (Misses Per Kilo Instructions) >= 10; Each MIX workload contains 2 memory intensive benchmarks and 2 less intensive (MPKI < 10) benchmarks.

Workload	Applications
MEM-1	mcf, soplex, libquantum, milc
MEM-2	mcf, soplex, libquantum, sphinx3
MEM-3	mcf, soplex, lbm, sphinx3
MEM-4	mcf, lbm, milc, sphinx3
MEM-5	soplex, libquantum, lbm, milc
MEM-6	libquantum, lbm, milc, sphinx3
MEM-7	bwaves, sphinx3, lbm, milc
MEM-8	bwaves, mcf, soplex, leslie3d
MEM-9	GemsFDTD, bwaves, lbm, xalancbmk
MEM-10	GemsFDTD, mcf, sphinx3, bwaves
MEM-11	soplex, libquantum, GemsFDTD, lbm
MEM-12	lbm, mcf, sphinx3, bwaves
MIX-1	lbm, sphinx3, tonto, calculix
MIX-2	lbm, milc, tonto, namd
MIX-3	libquantum, milc, gcc, namd
MIX-4	mcf, sphinx3, gobmk, calculix
MIX-5	mcf, soplex, gobmk, sjeng
MIX-6	soplex, libquantum, sjeng, gcc
MIX-7	lbm, sphinx3, perlbench, bzip2
MIX-8	lbm, leslie3d, gromacs, cactusADM
MIX-9	libquantum, GemsFDTD, povray, hmmer
MIX-10	mcf, sphinx3, h264ref, omnetpp
MIX-11	bwaves, soplex, perlbench, astar
MIX-12	bwaves, libquantum, cactusADM, povray

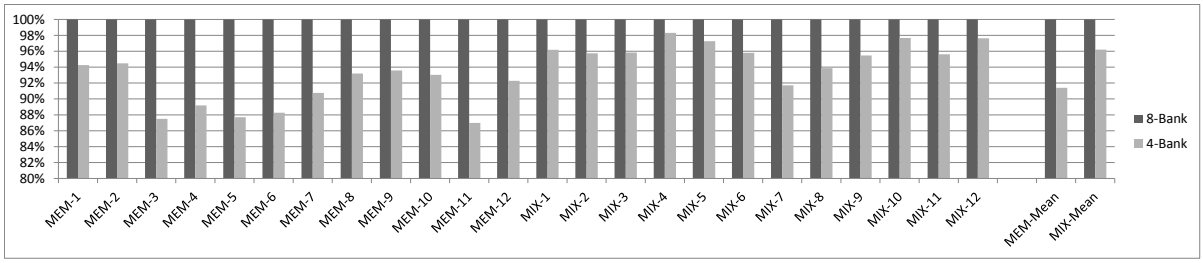
configurations. As high as 13% performance loss is observed. The average performance loss is 6.6% and 4.7% for page- and cacheline-interleaving schemes, respectively. The performance loss of channel- or rank-based partitioning will lead to much higher performance loss, because the degree of memory-level parallelism will be severely limited. Although these experiments only covers the case where parallelism is reduced at bank-level, similar result is expected when reducing channel- or rank-level parallelism by same percentage. Thus, partitioning by channel, rank or bank is not acceptable.

3.6.2 Performance Impact of Division

Division is the most straightforward way to implement non-power-of-two address mapping. However, using division operation as mapping function here could potentially penalize sys-



(a) Performance loss of cacheline-interleaving schemes

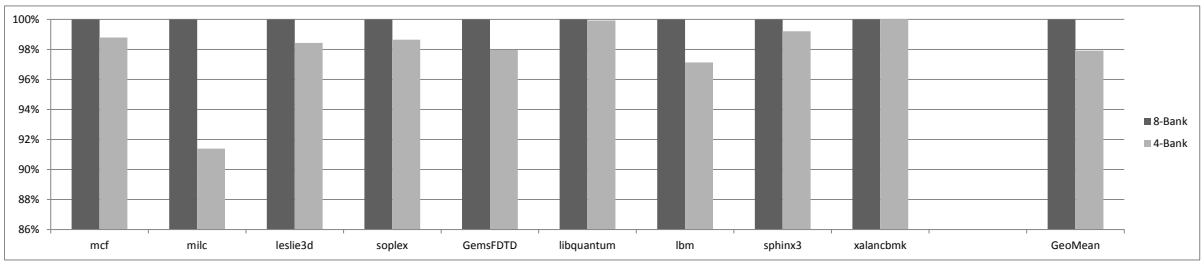


(b) Performance loss of page-interleaving schemes.

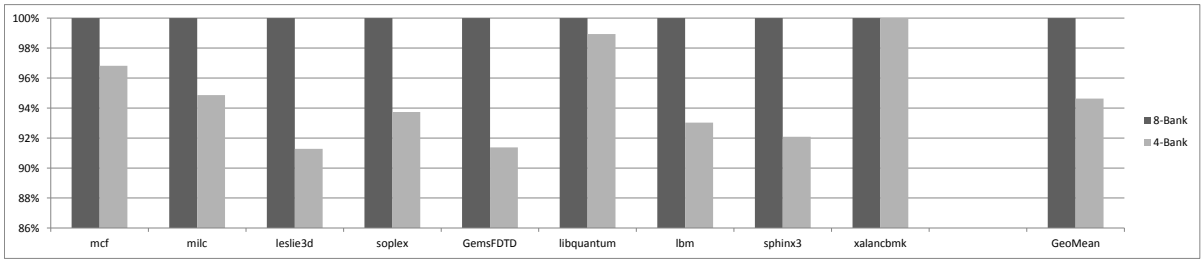
Figure 3.5: Normalized SMT speedup when bank parallelism reduces. All speedup are normalized to that of plain mapping.

tem performance. This is because division operation has relatively long latency and limited throughput.

In this section, we evaluate the performance impact of using division-based address mapping instead of BCRM and Segmented BCRM. We design experiments to measure the performance penalty when division is used as mapping function, including both an ideal scenario with no queuing delay and a realistic scenario with queuing delay. In our experiments, division operation is assumed to have an optimistic latency of eight memory cycles (equivalent to 32 processor cycles). In the ideal scenario, the number of dividers is assumed to be infinite so that memory requests have no queuing delay. We assume to have a FCFS queue for using the dividers, and evaluate the performance of one, two, four and unlimited number of non-pipelined dividers. Figure 3.7 and Figure 3.8 show that with unlimited number of dividers, performance loss is up to 6.1% for multi-core workloads and 13% for single-core workloads. The performance loss is caused by latency of the divider. With one divider, the performance loss ranges from 38% to 62% for those four-core MEM workloads. The performance loss shown here is mostly due to throughput limit of having only a few divider. In other words, a larger scale memory system



(a) Performance loss of cacheline-interleaving schemes



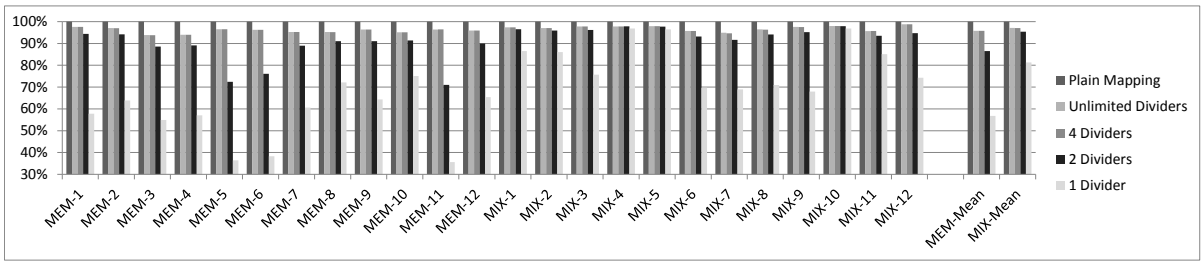
(b) Performance loss of page-interleaving schemes.

Figure 3.6: Single-core IPC when bank parallelism varies. All IPCs are normalized to that of plain mapping.

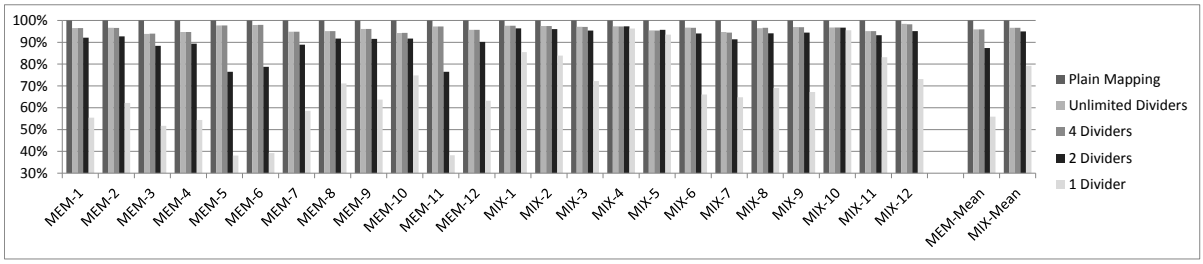
requires more dividers to minimize the performance penalty. In our experiments, four-divider system is enough to minimize the queuing delay for a two-channel memory system. However, it is impractical to include two or more dividers dedicated for memory address mapping purposes. Even in modern high-performance processors, the number of dividers is very limited due to its logic complexity. Therefore, division-based mapping is not feasible with reasonable hardware cost budget.

3.6.3 Performance Impact of BCRM and Segmented BCRM

BCRM and Segmented BCRM have identical performance. We run simulations with the plain address mapping and SBCRM and compare their performance. Figure 3.9 shows the normalized performance of multi-core workloads. Compared to the plain mapping, SBCRM has slightly longer latency for address mapping than plain mapping. We assume extra latency of one memory bus clock cycle. The performance impact is negligible. For MEM-6 workload, we even observe a slight 1% performance improvement which is probably caused by the change of memory address mapping pattern.



(a) Performance loss of cacheline-interleaving schemes



(b) Performance loss of page-interleaving schemes.

Figure 3.7: SMT speedup when various number of dividers are available. All speedup are normalized to plain mapping.

On average, SBCRM causes 0.4% performance loss for four-core workloads and 1.5% performance loss for single-core workloads.

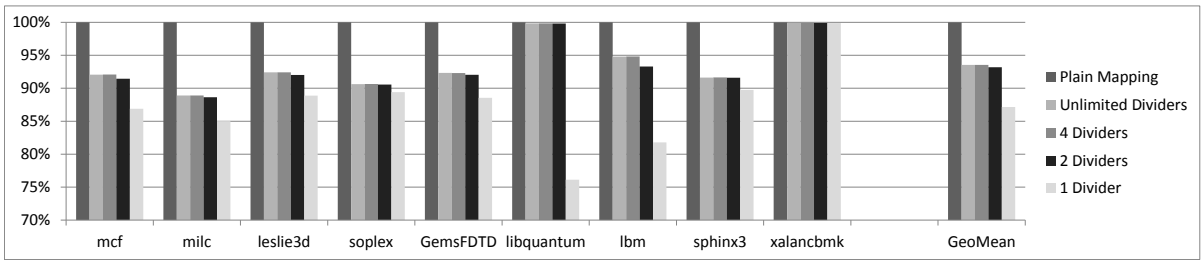
3.6.4 Memory Energy Consumption

Compared to non-ECC memory, ECC memory incurs energy overhead. The overhead can be split into two parts, the energy for operating the memory devices for ECC and the energy for transferring ECC on memory bus. SEP can effectively reduce energy of both storing and transferring ECC when compared to E^3CC and conventional 9-device ECC memory. To quantify the energy saving, we evaluate the energy consumption of programs running in the SEP framework.

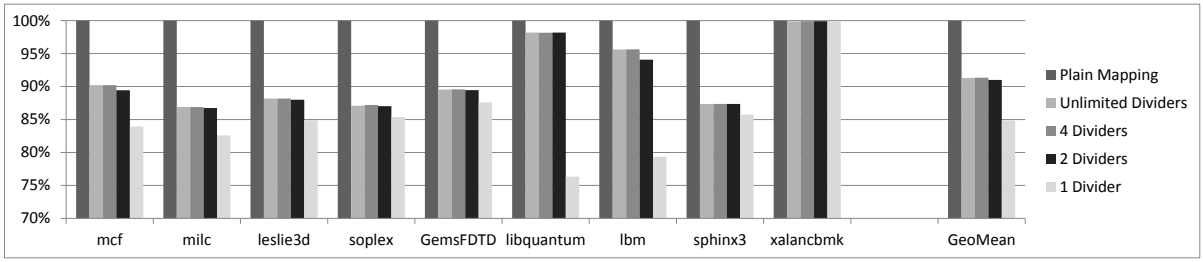
We model the energy consumption of SEP scheme using the following formula:

$$E_{SEP} = E_{noECC} + (E_{embeddedECC} - E_{noECC}) * Ratio \tag{3.7}$$

where $E_{embeddedECC}$ is the energy consumption of memory with embedded ECC, calculated in the same way as in [8]; and $Ratio$ is the protection ratio of a workload's memory. Note that this simple model does not consider the variation of access frequency to the two regions. Four



(a) Performance loss of cacheline-interleaving schemes



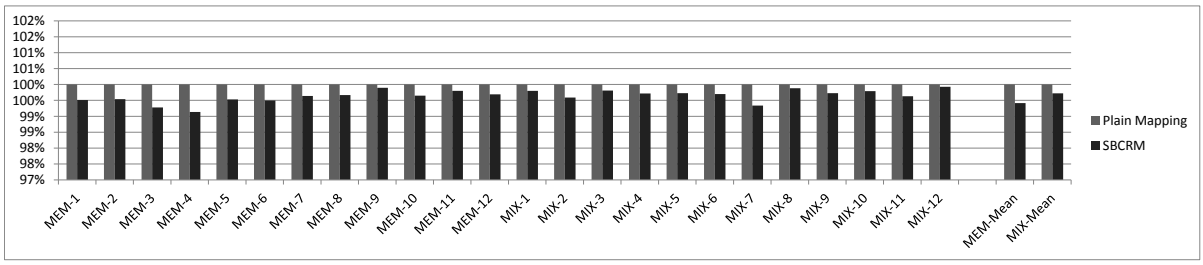
(b) Performance loss of page-interleaving schemes.

Figure 3.8: Single-core IPC when various number of dividers are available. All IPCs are normalized to that of plain mapping.

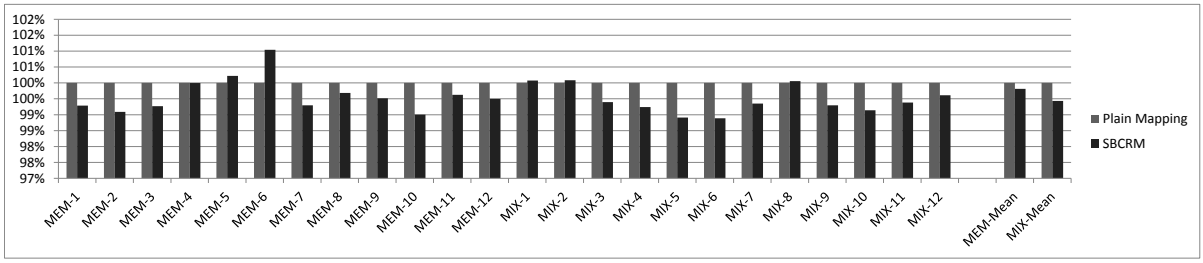
choices of the protection ratio are presented, namely 10%, 20%, 36%, and 50%, respectively. Out of all four choices, the key protection ratio is 36%, which provides 99% reliability according to previous study.

In Formula 3.7, SEP energy is modeled as a linear function of protection ratio and E^{3}CC energy overhead against unprotected scheme. We believe this model is accurate enough because extra energy is only needed when protected region is targeted. Assuming uniform memory access, the number of memory requests in protected region should be linear to protection ratio.

Figure 3.11 shows the energy saving of SEP when the protection ratio varies. The energy consumptions for non-ECC and full-ECC are directly collected from simulation. Conventional nine-device full-ECC consumes approximately 12.5% extra energy than non-ECC, which (although not shown in figure) is higher than any other schemes presented. The SEP energy is estimated using Formula 3.7. As the figure shows, Embedded ECC with full ECC protection incurs extra energy consumption that ranges from 13% to 18% for memory-intensive workloads and 5% to 16% for mixed workloads. In the case of protecting 36% memory, energy overhead



(a) Performance loss of cacheline-interleaving schemes



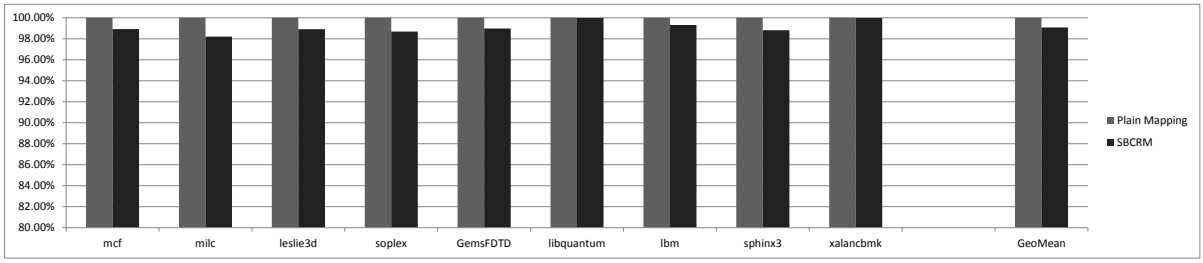
(b) Performance loss of page-interleaving schemes.

Figure 3.9: Four-core workloads SMT speedup with different mapping schemes. All speedup are normalized to that of plain mapping.

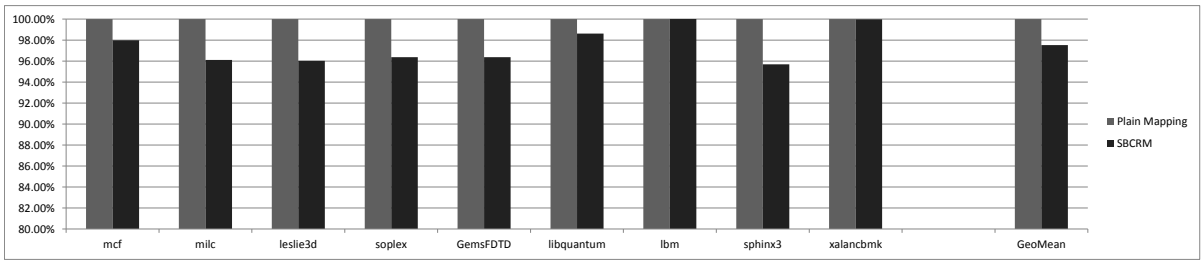
ranges from 2% to 6%, which is much lower than other full ECC protection at the cost of losing 1% reliability.

3.7 Summary

We have presented an efficient memory SEP mechanism to support a memory SEP system using commodity memory modules and devices. It partitions the whole set of DRAM rows into two regions, a non-protected region and an ECC protected region. A new address mapping scheme called parameterized BCRM is proposed to map physical memory address into DRAM device address components, as well as Segmented-BCRM design which reduces cost more by basing itself on a non-continuous memory address space. With this support, the OS may dynamically adjust the sizes of the ECC protected region and the non-protected region according to application demands. Our evaluation shows that the design incurs negligible performance overhead and improves memory energy efficiency.

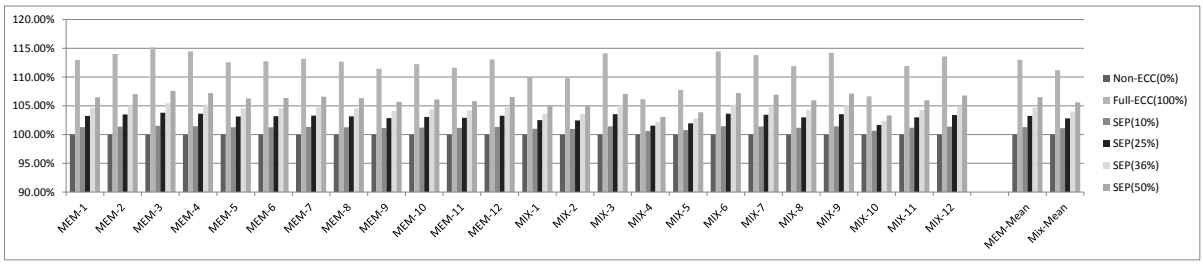


(a) Performance loss of cacheline-interleaving schemes

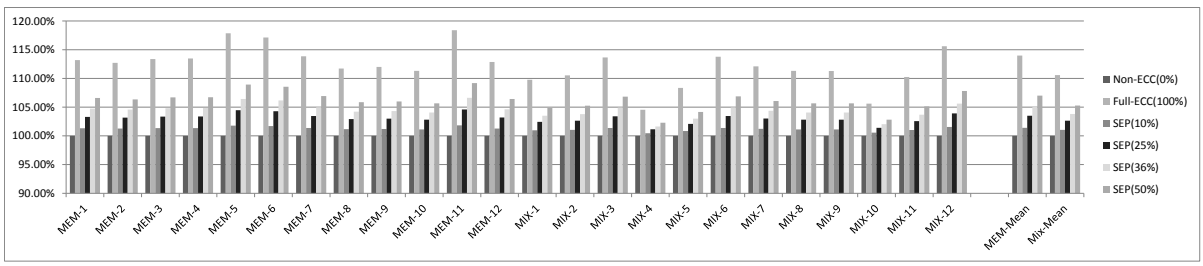


(b) Performance loss of page-interleaving schemes.

Figure 3.10: Single-core workloads IPC with different mapping schemes. All IPCs are normalized to plain mapping.



(a) Normalized energy consumption of various protection level with cacheline-interleaving scheme



(b) Normalized energy consumption of various protection level with page-interleaving scheme

Figure 3.11: Four-core energy consumption normalized to non-ECC memory system. Non-ECC is equivalent to 0% protection ratio with slight mapping latency difference. Full-ECC is equivalent to 100% protection ratio.

CHAPTER 4. FLEXIBLE MEMORY: A NOVEL MAIN MEMORY FRAMEWORK BASED ON MEMORY COMPRESSION

4.1 Introduction

Main memory has been identified as one of the most critical performance bottlenecks in modern computer systems. There are several reasons behind it. First, processors and on-chip caches have received more speed boost than DRAM-based main memory, making memory access more costly in terms of processor clock cycles. Furthermore, computer manufacturers are putting more cores in processor chips, enabling more processes or threads running concurrently. It is common for a GPU to have several hundreds of cores generating enormous amount of memory traffic. Applications are also becoming more data-intensive and therefore issue more memory access requests than before.

Today's data-intensive applications not only send more memory requests, they also keep more data in main memory, demanding higher capacity. If such demand can not be met by main memory, widely employed Virtual Memory scheme has to swap data between main memory and main storage (usually hard disks) to create the illusion of large memory space. However, access speed of main storages are several orders of magnitude lower than main memory, making each swapping a considerable performance burden. The situation becomes even worse for capacity-demanding applications because they tend to have more page faults. Simply scaling up memory capacity to meet the increasing demand is not a sufficient solution, because it also scales up memory cost and energy consumption, and DRAM fabrication does not scale as well as processor. In some server systems, for example, the main memory alone can consume more than half (57%) of whole system energy cost (not including energy of associated cooling components) [35].

Block-level hardware memory compression is promising to increase the effective memory capacity and bandwidth, as it reduces memory footprint and memory traffic [13, 48, 49]. It is unlike earlier hardware memory compression (e.g. MXT [57, 1, 58] or OS-based compression, which increase memory traffic and therefore not suitable for memory-intensive multi-core applications. The approach only requires proper OS support and is transparent to application software. The approach also has its own challenges to overcome. First, address mapping from main memory address to memory device address is more complicated. Conventional simple address mapping, which relies on uniform memory block size and simple, sequential layout, may not work in compressed memory. Second, there is a new scenario called fat write [33], which refers to the type of write operation that triggers storage expansion in compressed memory. When it happens, non-trivial operations are needed to make enough room for the new data. Such operation can be very costly as they may involve movements of multiple memory blocks. A mechanism that can minimize fat writes and/or reduce overhead of each fat write has to be put in place to make memory compression practical.

Several main memory compression schemes have been proposed in the past. MXT [57, 1, 58] uses an additional layer of address mapping to locate large chunks (1KB) of memory space and a large (32MB) cache to buffer decompressed data for quick access. This design not only adds considerable logic overhead but also increases memory traffic: Program execution at CPU can not directly access compressed data, and thus compressed data has to be read out, decompressed and then written to memory before the access, in a way similar to page fault handling. Recently proposed, block-level hardware compression has addressed this weakness by allowing direct program access to compressed data, usually in granularity of cache-line size. Robust memory compression [13] uses a page-level structure to organize the compressed blocks. However, its organization of compressed blocks incur high latency overhead when locating a compressed block. LCP [48, 49] simplifies the locating process by allowing only one common block size of compressed data in each page, keeping compressed cache-lines in sequential order. An over-sized block is stored in a reserved space with the indirection information stored in the original block location. However, such a design may not fully explore the compression opportunity of a program that has high variation in compressed block size. MemZip [54] uses

memory compression to reduce memory traffic. To avoid address mapping complexity, the extra space from memory compression are not utilized to improve memory capacity.

In this work, we investigate in Flexible Memory, a new design of block-level hardware memory compression. Any memory compression scheme will complicate the layout of memory contents in the physical memory storage. While decompression latency can be minimized using recently proposed, fast decompression methods, locating the compressed block may potentially incur high overhead to memory access latency. To help address this issue, our design uses a unique data structure called BMT (Block Mapping Table) and a set of supporting components in the memory controller. A BMT holds the location information of all memory blocks belonging to an OS page. The BMT structure is designed so that a single access to BMT is sufficient to locate any memory block in a page, and it is compact enough such that a BMT can be fetched into the memory controller easily. The memory controller embeds a small BMT cache to facilitate BMT access, which has high hit rate for the workloads we evaluated. Coupled with recently proposed, high-speed high-throughput Base-Delta-Immediate (BDI) compression [50] and Frequent-Pattern Compression (FPC) [4], the use of BMT and BMT cache eliminates the majority of time overhead in accessing a compressed memory block. Furthermore, the design optimizes the page-level organization to reduce page expansion from fat writes. Our evaluation shows that, on average, the design yields 1.5x improvement of memory capacity, 14% power saving, and 7.5% performance improvement in weighted IPC speedup.

The rest of this paper is organized as following: Section 4.2 discusses the previous work of main memory compression scheme. Section 4.4 illustrates the basic idea and detailed implementation of *Flexible Memory*. Section 4.5 presents the simulation methodology and environment. Evaluation results are shown in Section 4.6. Section 4.7 concludes the work.

4.2 Background and Prior Work

4.2.1 Compression Algorithms

To achieve good performance in a compressed memory scheme, a lossless compression algorithm with low latency, satisfactory compression ratio and simple implementation is essential.

In this section, we give introduction and discussion of algorithms used in previous works and Flexible Memory for compression of block granularity.

Zero-value compression [68] is a simple compression method based on the insight that many applications have dominating amount of zero-values in their memory spaces. Thus, simply marking all-zero pages/blocks with a flag bit in page table can be used as a compression method.

BDI (Base-Delta-Immediate) [50] is a cache-line granularity compression algorithm that looks for $B + \Delta$ patterns. B stands for a common data base shared by the data values stored in segments within a given cache-line, and a shorter Δ represents difference between each data value and the base. It requires less bits to encode Δ than the full data value. Immediate values are used in the case that most data values fit in the pattern but a few do not fit and are close to zero-value. After the compression, the size of the cache-line becomes $1 \times B + n \times \Delta_i + c \times I_i$. Both compression and decompression of BDI can be finished within one processor clock cycle. Since nothing but additions and subtractions are involved in computation of BDI, its implementation complexity is low enough to be duplicated to parallelize its operation. FVC (Frequent Value Compression) [66, 63, 62] is also a cache-line granularity compression algorithm. It exploits the pattern that some values appear in memory more often than others, like 00000000, 11111111, 01010101 etc. For these common values, a 3-bit encoding id is used to replace actual data. For other uncommon values, FVC leaves them uncompressed and mark them with a flag bit.

Latency of FVC algorithm depends on size of data, as compressed data segments have variable length, and the meaning of current value segment depends on the interpretation of previous one. Typically, for a 64B cache-line, decompression latency can be more than 5 cycles.

FPC (Frequent Pattern Compression) [4] is another cache-line granularity compression algorithm. FPC explores sparse data patterns. For instance, in a 32-bit word value resulted from sign extension of 16-bit word, only the lower 16 bits and the sign bit are significant. In this case, a 3-bit encoding id is used to identify this data pattern, followed by the 16 data bits. Data segments that can not fit in the pattern are left uncompressed with a flag attached. The compression/decompression latency of FPC is longer than that of BDI. It takes approximately

five processor clock cycles in a typical implementation compared to one processor cycle of BDI latency.

4.2.2 Prior Work

MXT (Memory Expansion Technology) [57, 1, 58] is a hardware memory compression scheme from industry. It divides main memory into relatively large (1 KB) blocks. Memory blocks are decompressed and filled into a large (32 MB) shared cache when requested. Program execution at the processor does not have direct access to compressed data. At the time of cache eviction, a data blocks is compressed again and written back to main memory. This approach may significantly increase the effective memory capacity, but may also increase memory access latency for uncached data and memory traffic.

Ekman and Stenstrom proposed RMCS (Robust Memory Compression Scheme) [13]. It is block-level compression with direct program access to the compressed data, therefore the increase of memory latency is moderate. In each page, the size information of all compressed blocks is stored in a header BST (Block Size Table), which is also used to locate a compressed block. A drawback of the BST is that the location of a block is calculated by summing up the sizes of all preceding blocks, adding extra latency on critical path. Additionally, this design does not allow out-of-order block placement or unutilized free space between blocks in most cases. Therefore, when block overflow or underflow happens, the only choice is often to shift following blocks to either make more space or to remove empty space.

Pekhimenko et al. proposed LCP (Linearly Compressed Pages) [49, 48]. It divides each page into three sections, namely compressed, uncompressed, and metadata sections. Blocks that can be compressed to an arbitrary size are stored in-order in the compressed section. The other blocks are stored out-of-order in the uncompressed section and indexed by pointers stored in the metadata section. This design enables LCP to quickly locate a block with little computation and partially allows out-of-order block placement to reduce the overflow overhead. However, compressibility is limited due to following reasons: 1) The size of smaller block has to be rounded up to linear size; and 2) an over-sized block costs storage space in both the uncompressed section and the compressed section.

Recently, Shafiee et al. proposed MemZip [54] that focuses solely on power, bandwidth and reliability benefits. It uses the same memory block layout as an uncompressed memory, but compresses block data to reduce memory traffic, and make the extra space in a given block available for reliability mechanism as extension. It does not increase the effective memory capacity, and therefore can avoid the complication from using a compressed memory layout. Blocks are compressed to smaller sizes but still allocated 64 Bytes and stored at same location as in uncompressed main memory scheme, leaving unused space available for reliability mechanism. Variable memory burst length combined with DBI (Data Bus Inversion) technique is employed to get considerable power and bandwidth gain. However, unlike the other designs, this design does not increase memory capacity.

4.3 Challenges of Compressed Memory

Traditional main memory design follows the following principles (although some variants of design may differ):

- Storing data in raw format, each byte of data occupies one physical address.
- Placing data in strict order according to its assigned address. Physical address is converted to device cell location by simple truncating.

Being straightforward, simple and effective, **Traditional Memory Design** has been long used as main memory system. During the time when speed gap between processor and memory device was narrow enough, sophistication level of traditional memory was more than enough to to handle demand of memory traffic. However, following Moore's law, computation ability roars, bringing much heavier load on memory traffic. Memory systems are stressed in many cases, especially in server clusters. It has been reported that on IBM eServer, main memory alone consumes as much as 40%[35] of total system energy. With huge power consumption and unsatisfactory performance, we can conclude that traditional memory design is starting to hit its limits in terms of latency, capacity, power and bandwidth limitations.

Power as the *currency* to buy more performance proposes greater and very unique challenge. Also it has drawn more and more attention in both academic and industrial field as the

popularity of mobile device advances and they generally have very limited energy budget and performance per watt metric has become one of top performance metrics. However, previous memory compression works doesn't reach DRAM operation details and thus not able to provide very accurate DRAM power changes caused by memory compression.

We can take bandwidth as an example to address the limitation of traditional memory design. Bandwidth is defined as maximum amount of data that memory system can process in a fixed amount of time. In multi-core processors environment, bandwidth demand is usually high and memory traffic has to be queued, leading to higher queuing delay. Straightforward and effective solution is upgrading to higher frequency bus or accommodating more memory channels, incurring monetary cost and more importantly higher power consumption.

In order to avoid a dilemma like this, we need to break the strong binding between **number of data bytes** and **amount of information**. In traditional memory design, 1 byte data $\equiv$ 1 byte information. While in essence, information is what processors need and amount of data is what burdens memory system.

So now we have a clear picture that the optimal way is to reduce number of *bytes* transferred on the bus while keeping amount of *information* conveyed unchanged. Main memory compression is an obvious solution to effectively save power and bandwidth.

However, main memory compression obviously breaks base rocks that traditional memory system is built on. Thus, it proposes several challenges and if not handled well, it incurs great complications and performance downgrade. So we propose a new DRAM-based memory scheme Flexible Memory.

First challenge is addressing, blocks no longer have fixed size and it would be wasting memory resource and negating the whole purpose of main memory compression to allocate memory blocks same space and location as in traditional uncompressed memory. Therefore, memory blocks should be able to be placed in arbitrary order. However, traditional memory system rely on simple implicit address operation to locate a block, which can not be easily adapted towards arbitrarily placed blocks.

For addressing challenge, we propose a unique data structure BMT (Block Mapping Table) and a whole set of supporting components. Its core data contains offset/size pair of every block

in each compressed page. BMT resides in main memory together with memory pages, And we propose a fully-associative BMT Cache to ensure timely access to BMT structure of each page without requiring 2x memory access. Main memory controller can then read out page offset of each memory block according to its BMT. Also, we add another layer of address mapping after traditional virtual-physical address translation that supports various page sizes. We call it virtual physical address. Combining virtual physical address and BMT information, it is easy to locate a block in DRAM.

Second challenge is page reorganization handling. In compressed main memory, fat write [33] is the most common and major reason for overhead. It comes from the fact that each block is able to have various size and a memory write request could mess up block layout by trying to fit a larger block into its original smaller slot. In order to avoid correctness issue, block movements are necessary to adjust block layout to make room for new block. Without proper data structures, policy and logic support, this could incur high overhead.

To tackle this challenge, we rely on great flexibility provided by BMT. As any block can be placed at any location that has enough free space for its compressed size, instead of following sequential order of any kind, we are able to move away any block that is in the way of any layout adjust attempt. With least restrictions, main memory controller can pick wisest block movement method to make room for new blocks.

Third challenge is how to get most bandwidth benefits out of main memory compression. Previously proposed main memory compression works rely on DRAM Cache to hold additionally fetched data. These extra data may be useful for future memory accesses. However, this relies on memory access locality to work. For applications with poor memory access pattern, DRAM Cache traffic reduction may not perform as well as expected.

4.4 Flexible Memory

4.4.1 Page Structure

The design of flexible memory aims to have fast access to compressed memory blocks, high utilization of memory capacity, and flexible structure for future extension (e.g. for reliability).

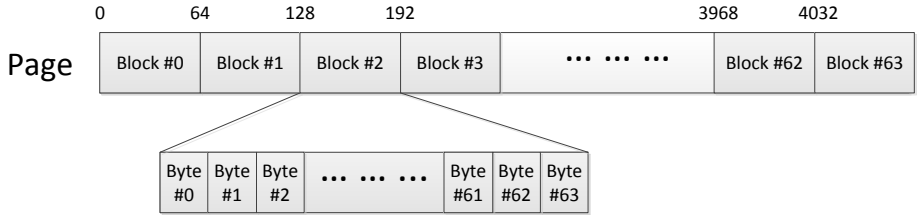


Figure 4.1: Traditional OS page. All blocks are placed back to back sequentially.

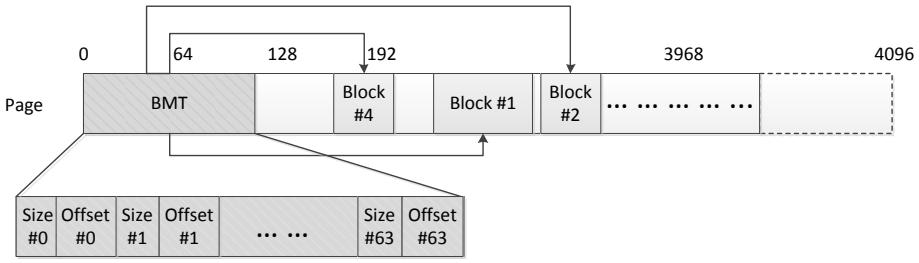


Figure 4.2: A Flexible Memory OS page. The beginning of page is Block Mapping Table, containing pointers to each memory blocks represented by grey boxes.

The design uses a page structure that allows arbitrary size (among a list of choices) for each memory block while keeping block locating as efficient as possible. In each page, the header stores the offset and size of all memory blocks in the page. The page header is carefully design such that it is the same size of a memory block (which is of cache-line size), so the page header can be fetched using a single memory access and can be cached in processor. This requires both offset and size of each cache-line stored in page header. Although doing so adds more space overhead, it is only a minor offset to memory capacity improvement by compression. With support of such page header, cache-line layout can be very flexible.

Figure 4.2 shows an example of the page structure. A compressed page in Flexible Memory is composed of two parts, a BMT section and a data section. The BMT is always located at very beginning (offset 0) of each compressed page, followed by the data section. The data section includes 64 compressed blocks followed by unused space. Within the data section, the block layout is highly flexible. The block layout can be out-of-order; for example, in Figure 4.2, block #4 is located before block #1. The layout also allows slacks between memory blocks, and the slacks can be utilized for expanding or relocating a memory block. Each compressed block can have an arbitrary size (in a small granularity). It is not required that all compressed blocks in a given page must be of the same size.

The process of locating a compressed block is highly efficient, even with the flexible layout. It only requires a single BMT lookup, which is cached in a BMT cache in the memory controller. The performance of the BMT cache is similar to that of TLB (translation look-aside buffer), which is usually very good for real-world programs. If a page located at address 0x3456000 has same layout as example shown in Figure 4.2, a memory request accessing block #4 can be completed with following simple steps. Firstly, BMT entry for block #4 located at $0x3456000 + pair\_size * 4$. After this lookup, FM gets its relative offset of 192 (0x12). Then a simple addition 0x3456000 + 0x12 = 0x3456012 would give physical address of requested block.

The flexibility in this design helps reduce memory access overhead. By comparison, RMCS does not allow slack space, and all blocks have to be placed back-to-back. When there is size change of a compressed block in RMCS, following blocks in the same page have to be shifted to avoid leaving slack space, causing high overhead. Flexible Memory allows slack space to exist in data section, eliminating this overhead. This flexible design also helps improve compression ratio. When a page with n blocks is loaded into main memory, each block is compressed to as small size as possible. Also, it is beneficial to add a small piece of slack space of size s_r on top of it as buffer area for future fat writes.

The slacks in a compressed page are managed as follows. At time of initial compression, there is no slack between blocks. Compressed page size is calculated by adding up the sizes of BMT and all blocks. To reduce management overhead, we pre-define 16 sizes of compressed page, and the page size is typically rounded up to the closest pre-defined compressed page size level. This leaves a relatively larger free slack space at end of page. During program execution, a fat write may cause a block to be re-located, leaving the previous location unused to become a chunk of slack. These slacks are useful in fat write handling. Suppose there is a block at offset o with size s, and an incoming write request causes the block to increase by Δ bytes. If there happens to be a piece of slack space adjacent to the compressed block, and the slack size is larger than Δ, then this fat write can be done with a single memory write, just like non fat write requests. The only overhead incurred is updating the block info in BMT, which is very convenient as it is cached in the BMT cache (see Section 4.4.5 for more details). Even if no adjacent slack space is available, a non-adjacent slack with size $s + \Delta$ is also able to hold

incoming data. A simple write request redirection to slack space would suffice to handle fat write. Similar to previous case, no extra memory operations are needed.

4.4.2 Memory Operation Handling

In order to comply with variable page size, physical address space of Flexible Memory is not uniform. Therefore, cache mapping needs changing in order to avoid performance overhead. Thus we use block index for cache mapping instead of page offset.

When OS sends a memory access request, unlike traditional memory where page offset is sent together with page address, page offset is separated into block index and block offset. Block index, together with page base address is good enough for all cache operations.

When LLC (Last Level Cache) issues a request to a compressed page in main memory, BMT is essential to completing the request. Firstly, a query to BMT cache is sent to see if corresponding BMT entry is cached. If not, a BMT cache miss event is executed to fill BMT Cache.

After retrieving BMT, memory controller uses block index to get block offset from BMT within the page. Combining BMT offset and physical page base address gives memory controller location.

To distinguish physical address layer with variable page size from traditional physical address, we name it VPA (Virtual Physical Address).

Memory address locating process is shown in Algorithm 1.

After locating the block, memory read access is usually nothing but determining whether accessed block is compressed or not. If uncompressed, retrieved data is directly returned. Otherwise, another step of decompression is needed. This memory read process is shown in Algorithm 2.

Memory read access usually goes as what is described above, while memory write is more complicated because it may change block size. In this case, main memory controller needs to figure out if current page has a free slot for it. If so, then likely a block movement is required, then a BMT update event is triggered and new block offset and size are filled to corresponding entry of BMT Cache, ensuring correctness of future memory access.

Algorithm 1 Memory Block Locating

$PA \Leftarrow Physical\ Address$
$PPN \Leftarrow PA\ \&Page\ Number\ Mask$
$VPN \Leftarrow Page\ Table(PPN)$
if Page Compressed **then**
  if PPN NOT cached in BMT Cache **then**
    Retrieve BMT from (PPN, BMT Offset)
    Evict victim BMT from BMT Cache
  end if
  $BMT \Leftarrow BMT\ Cache$
  $BlockID \Leftarrow PA\ \&Block\ ID\ Mask$
  $Block\ Offset \Leftarrow BMT(BlockID)$
  $Block\ Size \Leftarrow BMT(BlockID)$
else
  $Block\ Offset \Leftarrow BMT(BlockID)$
  $Block\ Size \Leftarrow BMT(BlockID)$
end if
$VPA \Leftarrow VPN|Block\ Offset$
return $(VPA, BlockSize)$

Algorithm 2 Memory Read

$PA \Leftarrow Physical\ Address$
$(VPA, BlockSize) \Leftarrow locateBlock(PA)$
if $BlockSize \neq RawBlockSize(64Byte)$ **then**
  $Compressed\ Data \Leftarrow retrieve(VPA, Block\ Size)$
  $Uncompressed\ Data \Leftarrow uncompress(Compressed\ Data)$
else
  $Uncompressed\ Data \Leftarrow retrieve(VPA, Block\ Size)$
end if
return $Uncompressed\ Data$

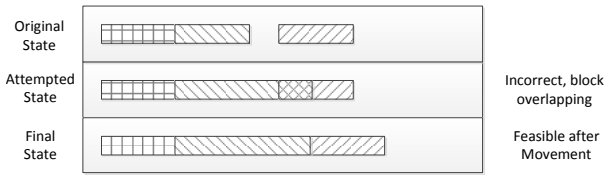


Figure 4.3: Reorganization Example

However, it is possible a new block is expanding to a new size and there isn't a continuous free slot in its owner page to hold it. Then some block movement are required so that separate slots can be combined together, creating enough continuous space.

Figure 4.3 demonstrates a simplest example of page reorganization. In original state, three blocks are represented by three different kind of shades and they don't overlap with each other, and there is 8 byte free space between block #2 and block #3 . However, when memory controller tries to write new data to block#2 that cause a block size expansion with size delta larger than 8 Byte, if no adjustment is made, "attempted state" would happen and it is obviously over-writing data of block #3, causing correctness issue. Thus, assuming block #3 has enough following free space, we only need to shift block #3, giving enough room for enlarged block #2. Though many cases are more complicated than this and more blocks are involved, solutions used are similar. And we call this method reorganization.

However, when a page doesn't even have free space to accommodate size delta at all, we have to extend size of page to create more space. And we name this process as page expansion. Moreover, if a page is already of largest compressed size possible, next expansion would cause size to be over 4KB, expansion is not possible any more. Then this means this page is not suitable to be compressed , thus it should be decompressed to its original state. Detailed process can be found in Algorithm 3.

4.4.3 Page Table and Sub-page Management

In Flexible Memory, the memory is divided into 4KB physical pages (which is assumed to be OS page size), and each page is divided into sub-pages of 256 bytes each. A compressed page must occupy a set of sub-pages within a single memory page container. Thus, the size of a compressed page is multiple of 256 Byte, resulting in 16 possible sizes (1 to 16 sub-pages).

Algorithm 3 Memory Write

```
PA ⇐ Physical Address
(VPA, OriginalBlockSize) ⇐ locateBlock(PA)
VPN ⇐ VPA&Page Number Mask
NewBlockSize ⇐ compress(WrittenData)
if NewBlockSize ≤ OriginalBlockSize then
    writeToMemory(VPA, NewBlockSize)
    if NewBlockSize ≠ OriginalBlockSize then
        updateBMT(VPA, NewBlockSize)
    end if
else
    NewOffset ⇐ findSlot(BMT, NewBlockSize)
    if slotFound then
        NewVPA = VPN||NewOffset
        writeToMemory(NewVPA, NewBlockSize)
        updateBMT(NewVPA, NewBlockSize)
    else
        LockPage
        Page Layout Reorganize
        UnlockPage
        NewOffset ⇐ findSlot(BMT, NewBlockSize)
        if slotFound then
            NewVPA = VPN||NewOffset
            writeToMemory(NewVPA, NewBlockSize)
            updateBMT(NewVPA, NewBlockSize)
        else
            Decompress Page()
            (UncompressedVPA, BlockSize) ⇐ locateBlock(PA)
            writeToMemory(UncompressedVPA, BlockSize)
        end if
    end if
end if
```

It requires some changes to the page table as follows. In the PTE (Page Table Entry), extra fields are added to keep track of the starting address and number of sub-pages used by a page. Suppose sub-page size is s_{sub}, and size of a page container is s (4KB), then the number of bits needed to store its starting address is: $log_2 s/s_{sub}$, same number of bits are also needed to store number of sub-pages occupied. In our design, this adds up to 8 bits. In current processors, the PTEs usually have reserved bits or unused bits [9] that can be utilized for this purpose.

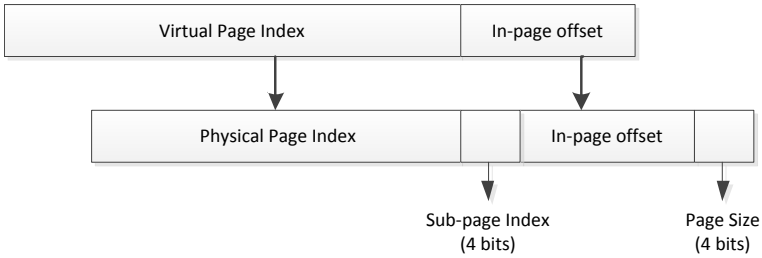


Figure 4.4: Virtual to physical address mapping. Adding 4 bit sub-page index and 4 bit page size to each PTE

Figure 4.4 shows the translation from virtual memory address to physical page index, sub-page index, page offset, and actual page size, using the revised page table and PTEs. The physical page index is the index of the physical page that contains the virtual memory page. The sub-page index gives the starting point of a virtual memory page in the physical page, and the page size field gives the number of sub-pages that it occupies.

Figure 4.5 shows an example of a 4KB physical page as a container of multiple compressed pages, which we call FM (Flexible Memory) pages. Of all 16 sub-pages, FM page #a takes 3 sub-pages starting from offset $0x100$, #b uses 4 sub-pages starting from $0x400$, and page #c occupies 5 sub-pages from $0xb00$ to the end of page. The other 4 sub-pages, marked as *Free*, are unused. An FM page can not cross boundaries of page containers. In this example, three FM pages could fit in one physical page container, saving 8KB in memory space. However, they still need three PTEs in the page table. Their sub-page index fields are 0x1, 0x4 and 0xb, respectively; and their size fields are 0x3, 0x4 and 0x5, respectively.

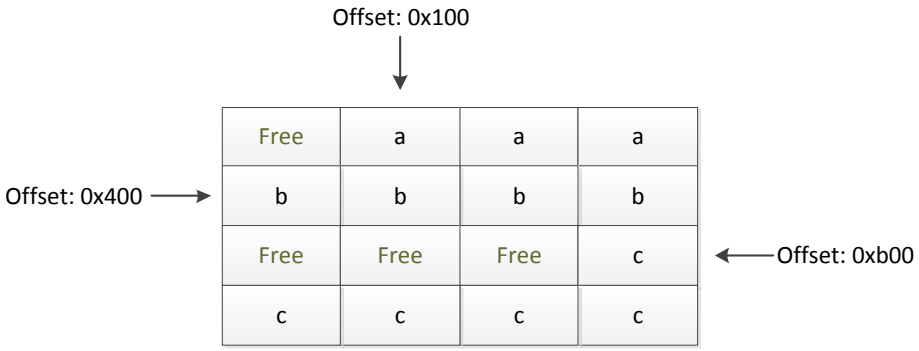


Figure 4.5: Physical pages and sub-pages structure; this page container holds 3 FM pages: a, b and c

4.4.4 Space Usage Efficiency Analysis

For a given set of data, in most cases FM (Flexible Memory) yields a smaller compressed page size compared to LCP by using allocated space efficiently. For example, assume that among the 64 blocks of a page, 16 blocks can be compressed to 8 bytes each, and 32 blocks can be compressed to 32 bytes each, and the last 16 blocks stays uncompressed (64 Byte). LCP would need at least $(16 + 32) * 32Byte + 16 * 64Byte + 64Byte = 3136Byte$. By comparison, FM only needs $16 * 8Byte + 32 * 32Byte + 16 * 64Byte + 96Byte = 2272Byte$. Even if we add additional 512 Byte slack space for future possible fat write, FM can still compress this page to a smaller size. An LCP page can be seen as a special case of a FM page where compressible blocks have same size. The drawback of FM is its larger page header size, 96 bytes compared to 64 bytes of LCP, which is a small difference that can be offset or usually completely hidden by the gain from better compressed blocks in FM than in LCP.

A management granularity of eight bytes is used to roundup compressed blocks, which also makes eight byte minimum operation granularity, in order to reduce management overhead. In other words, the sizes of compressed blocks are rounded up to $8 * n$ Bytes, and their offsets are rounded up as well. Note that DDRx memory devices support write data mask for partial update of memory block: In our design writing a compressed block, whose granularity is eight Bytes, does not require a read before the write.

In both FM and LCP, there are spaces that can not be efficiently used. They affect the efficiency of how memory compression schemes make use of available space. In an LCP page,

there are a fixed number of exception slots set aside for cache-lines that do not compress well. When a memory block is stored in exception slot, its corresponding slot in compressed section is still reserved to keep the layout aligned. Considering the fact that a memory block can be only in the exception section or the compressed section, there are always some space that is reserved but not used. The size of wasted space or zombie space in each LCP page is $num\_total\_exception\_slot * linear\_size$. Also, since LCP rounds up all smaller blocks to same linear size, there is roundup fragmentation.

In FM, the small slack space in a page can be hard to use; we call it fragmentation. Besides, roundup fragmentation exist in FM because of eight-byte minimum management granularity. A block is always allocated a space with size of multiples of eight bytes, causing certain roundup fragmentation too. The full evaluation can be found in Section 4.6.2.

4.4.5 BMT

The BMT (Block Mapping Table) is a critical part in locating compressed memory blocks. Some BMT fields are essential for block locating, and we call the BMT structure that only has these fields the *basic BMT*. The implementation of a basic BMT is simply encoding offset and size pair of every block and tightly packing into BMT structure, as shown in Figure 4.2.

Design overhead of BMT is basically space overhead that only exists in compressed pages. For uncompressed pages, BMT overhead is 0 since all memory blocks follow simple ordering.

As discussed earlier, *Block Size Granularity* is introduced to help mitigating BMT overhead. We will compute overhead for both basic BMT fields and advanced optional BMT fields. As a comparison, if we design Block Size Granularity as 16 Byte, BMT size would be 80 Bytes.

Basic BMT has following fields for each block:

- Block Size: number of possible block size levels is

$$\#BlockSizeLevles = \frac{64Byte}{BlockSizeGranularity} \tag{4.1}$$

  When Block Size Granularity is 8Byte, we have 8 block size levels, requiring $log_2 8 = 3$ bits.

- Block Offset: Similarly, depending on block size granularity: Number of possible block start offsets is:

$$\#PossibleblockOffsets = \frac{4096Byte}{BlockSizeGranularity} \tag{4.2}$$

For 8-byte granularity, it requires $log_2 512 = 9$ bits. needs 1 bit.

To reduce the overhead, we specify a block size granularity and require all block offsets and sizes must be a multiple of it. For example, with eight-byte granularity, a block starting from byte 256 with a size of 16 is represented as $offset = 256/8 = 32$ and $size = 16/8 = 2$. Obviously, the granularity is a critical design parameter that needs careful tuning. With a coarse granularity, we can reduce the BMT size. However, it also means that there is more rounding overhead. When we choose granularity of eight byte, size of BMT is only 96 Bytes, giving overhead of $BMTSize/PageSize = 96/4096 \approx 2.3\%$. If larger granularity such as 16 byte is chosen, BMT of a page can fit in a 64 Byte block.

The memory controller of FM embeds a small BMT cache that caches recently used BMTs. It has high hit rate because it resembles behavior of translation look-aside buffer, which is known for its high hit rate, and thus the majority of memory requests only require a single memory access.

4.4.5.1 Advanced BMT

BMT can be more than pointers to memory blocks. Here are some advanced features that can be added to each entry of BMT:

- Free Slot Pointer: In some cases, we need to find spare space to fit a larger block than previous one. With a basic BMT, it is necessary to traverse the whole BMT and find out which slot is big enough. If we keep track of some slot pointers like free list in memory management, it is much easier to find proper slots.

- Free Space Record: In order to avoid higher space overhead of Free Slot Pointer. Flexible Memory can keep track of 1) Largest free slot available 2) Offset of largest free slot in page 3) Total free space available in page to help in page reorganizations. We call these three fields Free Space Record.

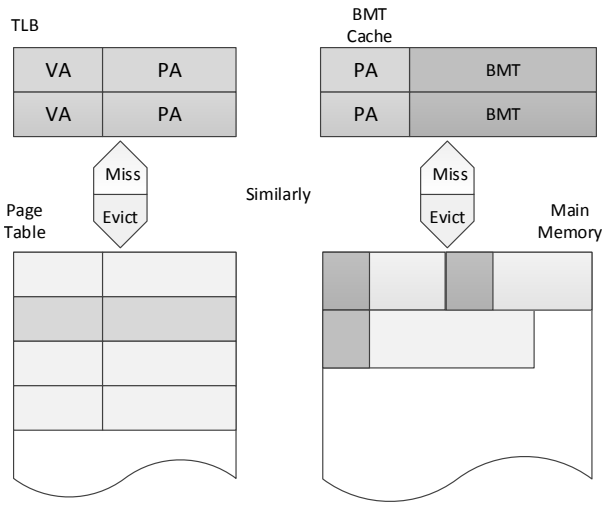


Figure 4.6: BMT Cache

4.4.5.2 BMT Cache

From OS' point of view, BMT is attached closely to each TLB entry in flexible memory scheme, although is is actually stored in a separate structure called BMT Cache which is very similar with TLB. TLB is usually built with CAM (Content Addressable Cache)[26], which is fully associative cache with all blocks in a single set. One restraint of CAM is that it gets slower when its size gets larger[42]. Therefore, we are not directly extending TLB CAM size to accommodate BMT of each page, which is 104 Byte per page according to previous calculation. Instead, another piece of RAM is added which follows entry updating and evicting behavior of TLB to guarantee same hit rate as TLB and low access latency of TLB.

Typical TLB design [47] can achieve a high hit rate of 99% with only 12 TLB entries. If chip size allows more entries, TLB can increase hit rate to 99.99%. Suppose a processor with 128-entry TLB, total BMT Cache size is $\#TLBEntry * BMTSize$ = 128 * 96Byte = 12KB.

4.4.6 Memory Access Handling

The page structure of Flexible Memory provides two properties, namely quick block locating and flexible block relocating. The first property accelerates access to compressed blocks, which is only a few cycles slower than uncompressed memory when the related BMT access hits in the BMT cache. The second property makes Flexible Memory capable of using any slack available

to deal with fat write with low overhead. For convenience of discussion, we will assume that the BMT is buffered in the BMT cache at time of access. In case of BMT cache miss, a memory access can fetch BMT to BMT cache. We assume that there is a way to distinguish compressed page from uncompressed page (not all memory pages have to be compressed), like keeping a compression flag in PTE.

After BMT lookup, a memory read request is completed by fetching data according to offset provided by BMT and then decompressing the fetched data. The extra latency is BMT lookup delay and the decompression delay. Therefore, memory read latency is only slightly increased. Considering memory reads are on critical path of program execution, this property guarantees system performance impact is low. Considering that memory reads are on critical path of program execution, reducing those latency overheads is critical to program performance.

Handling memory writes can be more complicated than reads, especially in the case of fat writes; however, write requests are usually not on the critical path of processor execution. A write is called a fat write if the compressed block expands and the original location does not have enough free space for the expansion. We further categorize fat writes into three types:

- Type 1: There is another large enough free slot in the same page to hold the expanded block.
- Type 2: There is no large enough single free slot, but combined free space in the page is enough.
- Type 3: The combined free space in page is not enough.

Type 1 fat write can be handled with virtually no extra overhead in our design. Since each block has its offset and size stored in the BMT, such fat write is handled efficiently by *write redirection*. The memory controller redirects write operation to the free slot and update the related BMT entry in the BMT cache. The BMT cache is write-back and the BMT update in main memory is only needed when a BMT cache entry is evicted.

Type 2 fat write is more complicated than Type 1 and triggers a *page reorganization*, which results in multiple memory accesses to move around memory blocks. For example, assume there are eight free slots in a 1 KB compressed page, each of 16 Byte. A fat write that expands a

compressed block to 32 Bytes can be served by merging those part or all of those free slots. The design has to be very careful as a fat write of this type may incur high performance penalty and memory traffic. A naïve solution is *complete page reorganization*. When it happens in the worst case scenario, the memory controller may issue up to 64 pairs of memory reads and writes.

Considering that in most cases, the required size increase of the expanded block can be satisfied by combining only a few free slots instead of all, we include a low-overhead mechanism called *smart reorganization*. It tries to minimize the number of blocks moved by creating just enough space for the expanded block, instead of combining all free slots.

If a type 3 fat write happens, a *page expansion* must be executed to handle it. This is managed by the OS. In our simulation, we assume the overhead is on average four microseconds [37] of OS interrupt handling plus the time required to move the memory blocks. Such expansion can happen in-place when there is trailing free space. Otherwise it requires a page movement to free space. A special case is that with the expansion, the compressed page will become 4KB or larger. In this case, the whole page is uncompressed and the page table entry is updated to mark the page as uncompressed.

4.4.7 Compression Algorithm Design

We settle down with a hybrid BDI-FPC compression algorithm which incorporates two pattern-based compression algorithm BDI and FPC to reach higher compression ratio by combining their pattern coverage while keeping compression/decompression latency low considering that BDI and FPC both have few-cycle compression/decompression latency.

4.4.8 Compression and Decompression Engines

The compression and decompression latencies are critical to the performance of compressed memory. As discussed earlier, the compression method is based on a combination of BDI [50] and FPC [4] compression.

The structures of compression and decompression engines are shown in Figures 4.7 and 4.8. Each compressor box represents compression logic of a certain pattern. In BDI, the patterns

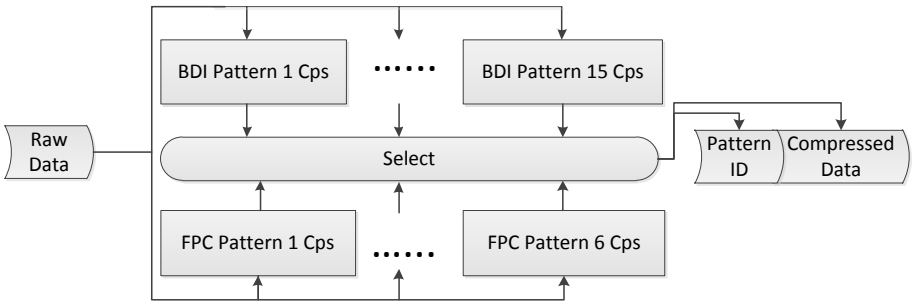


Figure 4.7: The compression engine structure. All pattern compression units work in parallel. Uncompressed data is sent to every pattern compressor of BDI and FPC. After compressing, all compressed data are sent to a select logic, which picks the pattern most suitable for this batch of data. Pattern ID and compressed data are concatenated as the final output.

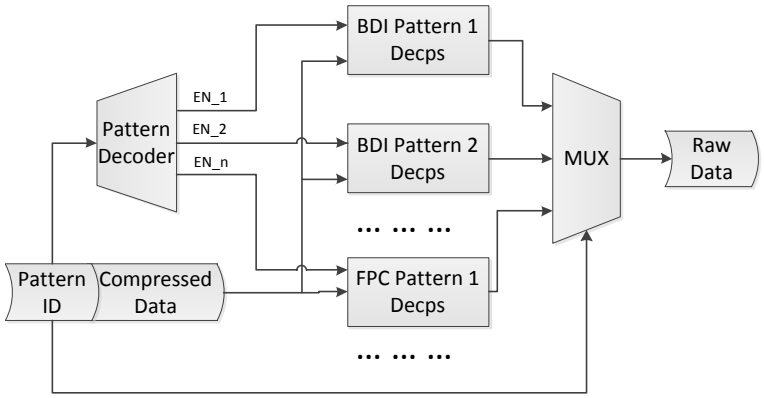


Figure 4.8: The decompression engine. All pattern decompression units work in parallel. First, a pattern ID is extracted from green compressed data and decoded into decompression engine enable signals $EN\_1, EN\_2 \cdots EN\_n$. These signals enables one of all decompression engines , which translates compressed data to raw data. Also the enable signal is routed to a multiplexer, redirecting effective raw data to final output.

are B8D1, B8D1I, B4D1, B4D1I etc. For FPC, the patterns are zero-run, repeated value, half-word sign extended etc. Same representation is used in decompression engine.

To reduce the latency of compression/decompression, every pattern of BDI and FPC has an independent logic unit, all of which running in parallel, and whichever pattern yielding the best compression ratio is selected. An encoding type id for the selected compression pattern is concatenated with the data output to form the compressed data, and is padded to be a multiple of eight bytes. At the time of decompression, the data portion is sent to the input of all decompression units, and the encoding type id is used to select one decompression unit.

Due to the simplicity of BDI and FPC compression algorithms, the BDI compression/decompression latency is as short as 1 clock cycle [50], and that of FPC is up to 5 clock cycles [4]. The in-

ternals of the compression/decompression engines only comprise of addition, comparison and MUX logics, whose size (number of gates) is linear to the input/output width, which is up to 64 bytes in our design. Compared to the complexity of modern processors, the area overhead is very small. As for power overhead, a previous study reports that the dynamic power consumptions for BDI compression and decompression are 29.7 mW and 23.6 mW, respectively; and the static power consumption is 52 μW [36]. By comparison, in our experiments the DRAM power consumption is from 4 W to 16 W, for a memory system of four DIMMs and the selected workloads, so the power overhead of the BDI compression/decompression is negligible. FPC has lower implementation complexity than BDI and thus its power overhead is also negligible.

4.4.9 Traffic Reduction by Over-Fetch Cache and Merging Write Queue

A potential problem for compressed main memory is that DDR3 memory has a fixed burst length of eight, as each data I/O pin has to transfer eight bits per column access (for a read or write command), and the memory rank returns 64 data bytes in total. There is a DDR3 Burst Chop 4 mode that terminates the transfer after the first four bursts; however, for the rank being accessed, the time for the following four bursts may not utilized. There are other timing constraints and complications in using this mode, so we do not consider it in our study. With memory compression, the size of the demanded block can be less than 64 bytes, but 64 bytes of data have to be transferred, which we call *over-transferring*. Without addressing this problem, for full-rank memory, memory compression may not reduce memory traffic at all.

For read requests, we use an over-fetch cache to address the problem. It is a small cache to store over-fetched data. A memory read request is first checked with the cache. If it is a hit, the data from the over-fetch cache is used; otherwise, the request continues as normal. We find that this small cache is highly effective, which is not a surprise given the spatial locality existing in many workloads.

For write requests, the write queue of the memory controller is revised so that each entry holds a write mask additional to write data. For write request to a compressed block of less than 64 bytes, the write mask ensures that other part of data in the same 64-byte block will not be overwritten. We have also revised the write queue, now called merging write queue, so that

it can merge compressed writes to a same 64-byte block: If a latter write hits on a write queue entry, the new data is merged into the entry and the write data mask is extended to cover the new data. Note that the merging write queue does not reduce write traffic as effectively as the over-fetch cache, because there is much less spatial locality in the write traffic from a normal write-back cache.

To illustrate the difference in two methods, let's use following example, in a full-rank memory system, block container is 64-byte wide. Two consecutive blocks (block Id 0 and 1) are compressed to size 16 byte and 48 byte respectively. They can fit into same 64-byte block container. When block 0 is being accessed main memory, it leaves an entry in processing requests queue of main memory controller. Since its data has not been returned, DRAM Cache doesn't have its corresponding entry yet. When second request accessing block 1 arrives, memory controller checks dependency of it and found that there is a previous request accessing same block container by comparing high bits of address, memory controller can mark second request as linked to first one. Then when first one finishes, second request also finishes. This process all happens without DRAM Cache. However, when second request comes much later due to low locality or processor stall, first request should have long been completed and block 1 data is already fetched and stored in DRAM Cache. This also saves second request.

Finally, memory sub-ranking will reduce the amount of over-transferring. Memory compression and sub-ranking is a good combination for DDR3 memory. To sustain high bandwidth, DDR3 devices has a minimum burst length of eight bits. Each DDR3 memory access returns a 64-byte memory block[1]. Block-level memory compression may compress the size of a 64-byte cache block to less than 64 bytes, but reading/writing a reduced-sized block is not efficient to DDR3 memory. Extra data will have to be fetched to the processor. Sub-ranking will reduce the minimum size of memory block and improve the utilization of memory bandwidth. Sub-ranking reduces the number of devices involved in a memory access and therefore reduces the number of bytes transferred per column access. The number is 32 and 16, respectively, for 32-bit and 16-bit sub-rank. Sub-ranking reduces the amount of data transfer for every column

[1]There is a burst chop mode that may cut this minimum length to 32 bytes, but using it will reduce the throughput of the devices and the utilization of the memory bus.

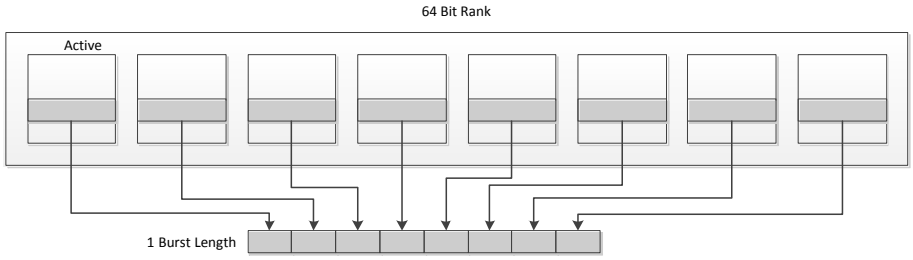


Figure 4.9: Full-rank traditional memory system

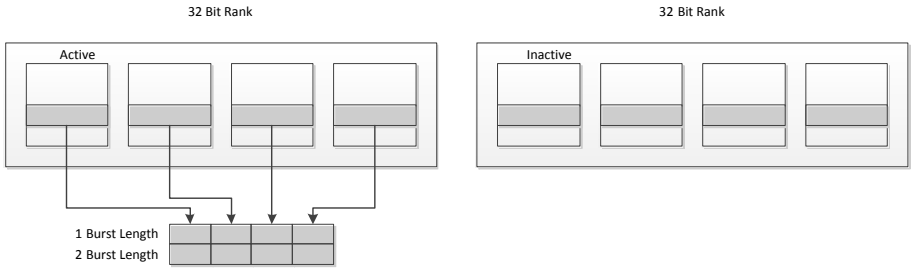


Figure 4.10: Sub-rank traditional memory system

access, while the over-fetch cache reduces column accesses, so the two have overlapped gains in read traffic reduction but still complement each other. Sub-ranking is very effective in reducing traffic.

4.5 Experimental Methodology

4.5.1 Benchmarks

We use the SPEC CPU2006 suite [20] with the reference input data set as the benchmark programs to construct our single-core and multi-core workloads. The benchmark programs are categorized by their compressibility and memory access intensity. We build Flexible Memory on top of sub-ranked scheme, more specifically 32-bit sub-rank and 16-bit sub-rank. Their corresponding minimum transfer units are 32 bytes and 16 byte, respectively. As for compressibility, we set the categorization thresholds at 50% and 75%, leading to three categories: High ($CR \leq 50\%$), Medium ($50\% < CR \leq 75\%$) and Low ($CR > 75\%$). Here the compression ratio (CR) is the ratio of the compressed size over the uncompressed size. The benchmarks are also categorized by their memory access intensity. Memory intensive benchmarks (MEM) is defined as those of $MPKI > 10$ while other benchmarks ($MPKI \leq 10$) are classified as Instruction

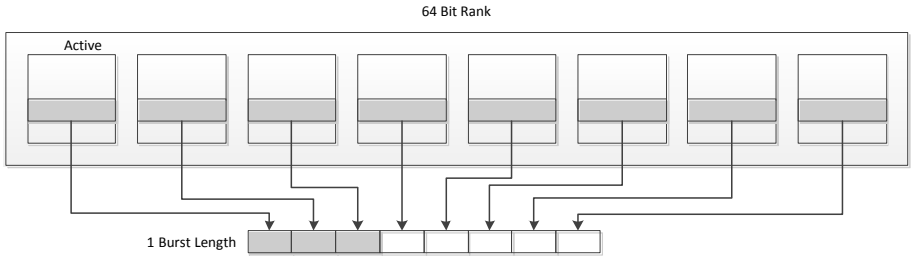


Figure 4.11: Full-rank compressed memory system

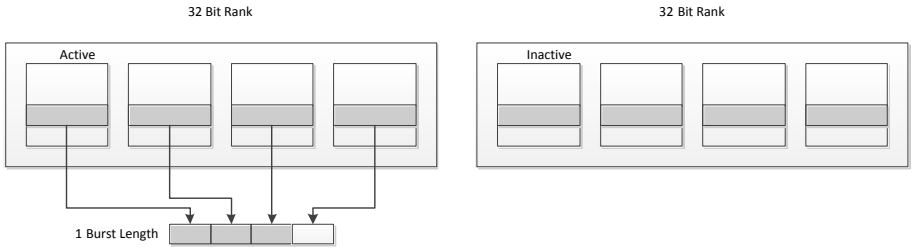


Figure 4.12: Sub-rank compressed memory system

Level Parallelism (ILP) benchmarks which are not bottlenecked by memory access. Table 4.1 lists the benchmark classifications.

The single-core workloads are constructed by running each benchmark in single core mode. Multi-core workloads are constructed by grouping four benchmark programs of similar property. Fifteen memory intensive workloads are constructed, with five in each MEM-x sub-category. Five ILP workloads are constructed. A list of these twenty workloads is shown in Table 4.2. They are used for power evaluation as power saving is directly related to compressibility.

In order to catch different behavior phases of each benchmark, multiple sampling phases are picked during execution of benchmark lifetime. For experiments without special setting, by default, they are interleaved by 2B and 4B instructions for single-core and multiple-core workloads.

4.5.2 Simulator

We use Marssx86 [44], a cycle-accurate full system x86_64 simulator. Due to nature of this work, it is necessary to extract all memory content belonging to the running programs. However, doing so requires running an additional program that scans the page table and communicates with the simulator while benchmark program is running, adding an overhead that can skew

Table 4.1: Benchmark Classification: MEM-* are memory-intensive workloads and ILP-* are compute-intensive workloads; H, M and L refer to high, medium and low compressibility, respectively.

Id	Benchmark	Class	Id	Benchmark	Class
0	libquantum	MEM-M	e	gcc	ILP-M
1	leslie3d	MEM-M	f	gromacs	ILP-M
2	bwaves	MEM-M	g	calculix	ILP-M
3	xalancbmk	MEM-M	h	h264ref	ILP-M
4	lbm	MEM-L	i	gamess	ILP-M
5	soplex	MEM-L	j	perlbench	ILP-L
6	milc	MEM-L	k	astar	ILP-L
7	sphinx3	MEM-L	l	bzip2	ILP-L
8	mcf	MEM-L	m	namd	ILP-L
9	GemsFDTD	MEM-H	n	hmmer	ILP-L
a	cactusADM	MEM-H	o	povray	ILP-L
b	tonto	ILP-M	p	zeusmp	ILP-H
c	omnetpp	ILP-M	q	sjeng	ILP-H
d	gobmk	ILP-M			

simulation results. Thus we choose to only capture memory pages that are accessed at least once, or in other words, active pages. Although this is not 100% coverage of pages owned by a process, it is good enough for performance evaluation as inactive pages have little impact on performance.

An in-house detailed model of DDR3 memory system with the Flexible Memory support is integrated to provide DRAM statistics. All DRAM power statistics are calculated according to the Micron power calculator [23, 24]. Other detailed simulation parameters can be found in Table 4.3.

4.6 Evaluation

In this section, we will show experimental results regarding improvements of memory capacity, performance and power efficiency.

Table 4.2: 4-Core workloads composition, benchmarks represented by Ids.

Workload	Benchmarks	Workload	Benchmarks
MEM-H-1	9999	MEM-L-1	4567
MEM-H-2	aaaa	MEM-L-2	4578
MEM-H-3	9aaa	MEM-L-3	4678
MEM-H-4	a999	MEM-L-4	5678
MEM-H-5	aa99	MEM-L-5	4568
MEM-M-1	1230	ILP-1	bcde
MEM-M-2	1100	ILP-2	fghi
MEM-M-3	2211	ILP-3	jklm
MEM-M-4	3322	ILP-4	nopq
MEM-M-5	3300	ILP-5	pqbc

4.6.1 Memory Content Compression Ratio

We use CR to represent the compression ratio, i.e. the ratio of compressed size over uncompressed size. *A lower compression ratio means better compressibility.* We have implemented LCP for comparison. 27 SPEC 2006 benchmarks are tested on a single-core system. There are 10 sampling phases, each of 200M instructions. The sampling points are separated by 2B instruction intervals. For each simulation phase, we scan all pages that are accessed during simulation. The same combined FPC-BDI compression algorithm is used for both FM and LCP. In other words, the only factor that can make a difference on compression ratio is how compressed blocks are organized in a page.

Only full-rank configuration is evaluated in this section, as sub-ranking configuration does not have any impact on in-memory content compression ratio. All zero-pages are not included as they are compressed in a special and very efficient way in both schemes.

Figure 4.13 shows that 22 out of 27 benchmarks get better compression ratio with FM than LCP. As much as 31.59% relative improvement is observed. The other 5 benchmarks show only average 5% difference between two schemes. The average compression ratios for FM and LCP are 69% and 77%, respectively. This is equivalent to 1.5x capacity gain. Part of improvement in compression ratio comes from allowing more levels of predefined compressed page sizes. To analyze how much this affects the compression ratio, we have collected distribution of page sizes. Figure 4.14 shows the number of pages that are compressed to a certain size from a

Table 4.3: Simulated System Configuration

Processor 1-4 cores	x86_64 ooo core 3.2GHz
L1 Data Cache	Private, 64 Byte Block
	8-way 64 KB/Core
L2 Cache (LLC)	64Byte Block
	8-way, 4MB Shared
DRAM	2 Channels DDR3 800MHz
Model	MT41J256M8-32M3gx8x8
Bus Frequency tCL/tRP/tRCD tRRD/tRC/tRAS	800 MHz 11/11/11 5/39/28
VDD/IDD0 IDD2P0/IDD2P1/IDD2N IDD3N/IDD3P IDD4R/IDD4W/IDD5	1.575 V/95 mA 12/37/43 mA 50/55 mA 156/145/195 mA
OS Context Switch Latency	4μs [37]
Memory Scheduling Latency	48 cycles
Cps/Decps Latency	5 cycles
BMT/MetaData Cache Size	4k entries
Over-Fetch Cache Size	4 KB
Added Slack Size Per Page	512 B

typical benchmark. Figure 4.15 shows size distribution when FM uses same set of page size levels. We can see that FM still has compression ratio advantage over LCP. This advantage is mainly due to the ability of having a compact layout while allowing various block sizes.

For benchmarks that favor LCP, FM loses some compression ratio because sizes of blocks can only be a multiple of management granularity, potentially wasting some space. However, as seen in the results, this disadvantage is small enough on average.

4.6.2 Space Utilization Analysis

In this section, we analyze the space utilization in FM and LCP. As described earlier in Section 4.4.4, FM and LCP both have roundup fragmentation, which we denote by F_{FM_R} and F_{LCP_R} respectively. Besides, FM may potentially create slack spaces that are too small that is unlikely for them to be utilized. The structure of LCP determines its zombie (unusable) space as $F_{LCP\_Z} = num\_tot\_available\_slots * linear\_size$. We collected fragmentation sizes by

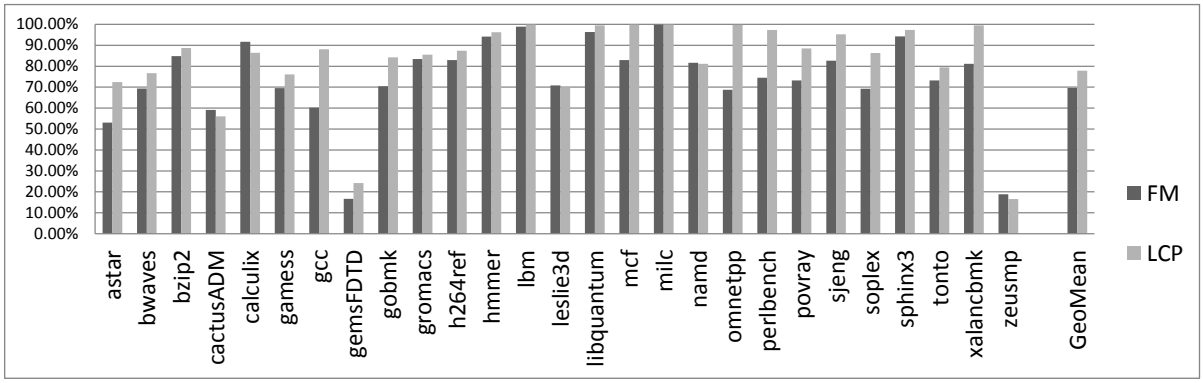


Figure 4.13: Single-core memory compression ratio. for each benchmark average compression ratio of 10 sampling points is presented

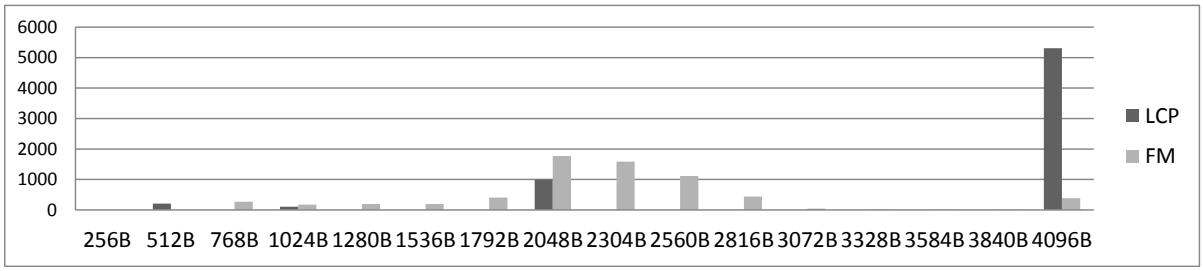


Figure 4.14: Page size distribution by number of pages that are compressed to each possible size (only gcc benchmark is shown)

taking 3 sampling points from each single core benchmarks, interleaved by 1B instructions, with length of 200M instructions per sampling point.

Figure 4.16 shows the breakdown of wasted space. The zombie/fragmentation space in FM is defined as a slack slot whose size is smaller than the average block size of its corresponding page. So not every piece of free space is fragmentation as they may likely be used. We can see that most (89.3%) of wasted space from FM comes from roundup, while only 10.7% is from the zombie space. This proves that FM is capable of efficiently utilizing allocated space. For LCP, 68.9% of wasted space is from zombie space. Overall, FM has 86% less wasted space compared to LCP.

4.6.3 Overhead Analysis

As stated earlier, the overhead of FM scheme composes of two parts. The first is the BMT access upon a BMT cache miss. In the simulation, most benchmarks show a BMT cache hit

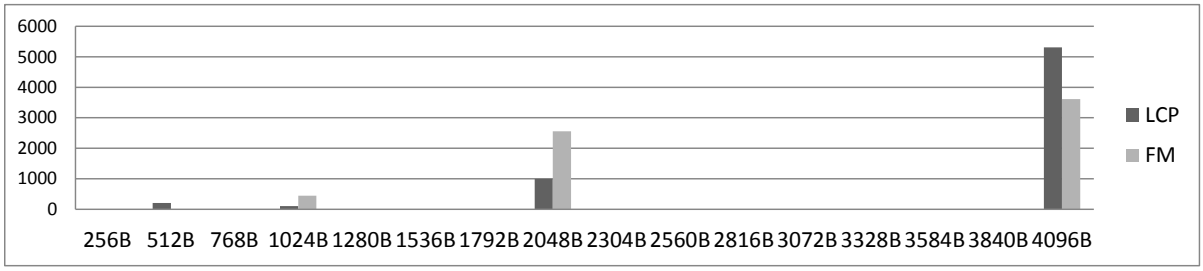


Figure 4.15: Page size distribution when applying page levels of LCP to Flexible Memory (only gcc benchmark is shown)

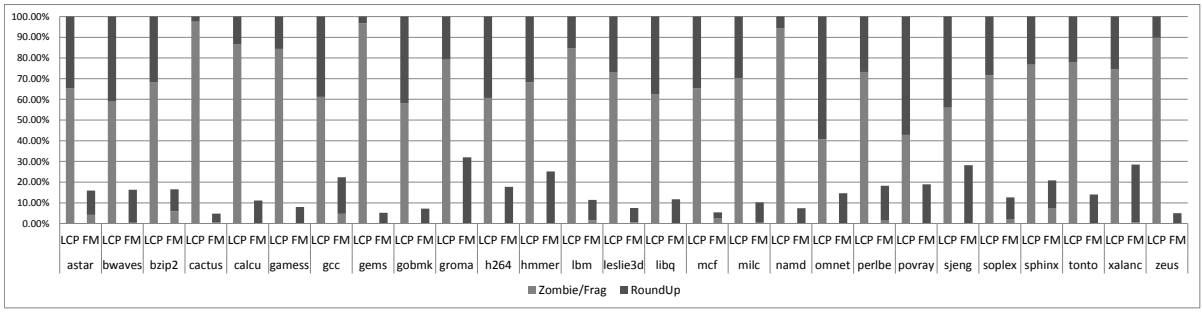


Figure 4.16: Wasted space of each benchmark normalized to that of LCP. RoundUp stands for roundup error in both FM and LCP. Zombie space refers small fragmentation for FM and Zombie space in LCP

rate over 95%, and for many benchmarks it can be as high as 99.5%. Even if a small 64-entry BMT cache is used, the hit rate is still over 80%. This is not a surprise given that a BMT entry covers a whole page of memory, similar to TLB design. Overflow caused by fat write is another main reason for the performance degradation in compressed main memory scheme. The overflow overhead comprises of two parts, namely the OS trap penalty and the data movement penalty. The former is set to 4 μs [37].

Figure 4.17 shows the possibility for an instruction to trigger a page overflow. For FM, page overflow includes page reorganization and page expansion. For LCP, it includes type-1 LCP overflow and type-2 LCP overflow. Any memory write handling that involves non-trivial operation or OS interrupt operation is counted as a case of overflow. In some benchmarks, like gamess, we didn't observe any overflow. Therefore they do not have a bar in logarithmic scale axis. In the figure, a longer bar signifies a lower possibility of overflow. On average, in LCP there are about 8 overflows per ten-million instructions, while in FM there are about 5 overflows per ten-million instructions, which is 37% less. Some benchmarks such as mcf do

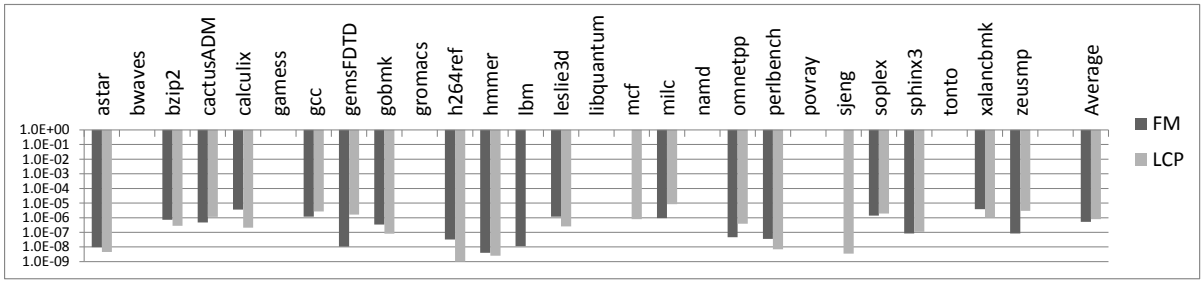


Figure 4.17: Possibility of triggering page overflow per instruction by each benchmark

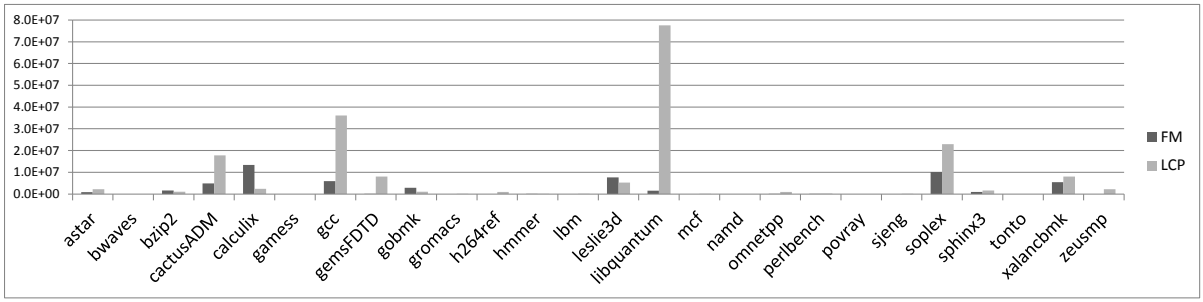


Figure 4.18: Memory traffic overhead generated by fat-write handling. Data presented shows how many bytes of extra traffic is needed across multiple simpoints

not have any overflow in FM but gives 10 overflows per ten-million instructions in LCP. In general, a more compressed page is likely to have more overflows as there are less free space available. However, FM is able to get both better compression ratio and lower possibility of overflow because of its flexibility in block layout.

Since LCP and FM handle fat-write with different methods and at different cost, it is unfair to use number of fat-writes to quantify the overhead. Instead we use bandwidth to quantify the overhead. Figure 4.18 shows amount of extra traffic generated by FM and LCP, respectively, to handle fat writes. For most benchmarks, both FM and LCP perform very well with little bandwidth overhead. In some other cases, the bandwidth overhead has a strong correlation with the frequency of heavy events in the fat write, such as page moves. Of all benchmarks, libquantumn shows the most difference between FM and LCP. A careful inspection of the detailed stats shows the program has very poor compressibility, thus LCP only has few exception slots in each page, leading to a high page recompression frequency. LCP can be designed in such a way that compression be turned off for benchmarks with high overhead so that libquantumn could be removed from comparison. Without removing libquantumn, the

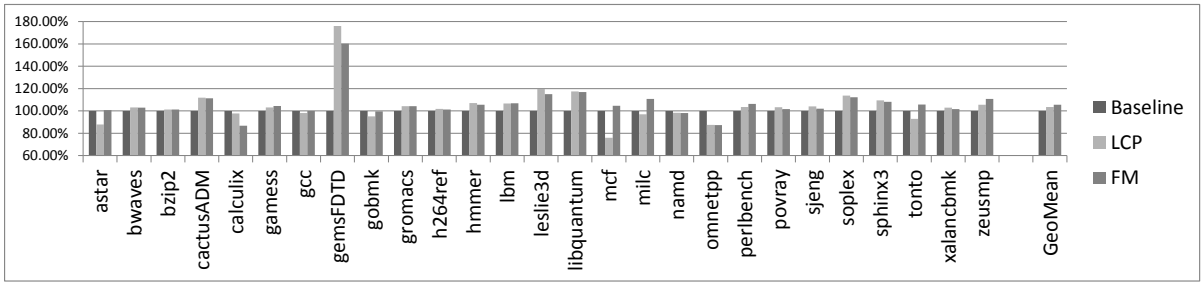


Figure 4.19: Normalized IPC of single-core workloads. All IPCs are normalized to baseline system that has uncompressed main memory

overall weighted bandwidth overhead of FM is only 29.3% of LCP. After removing libquantumn, the ratio is 48.5%. In short, FM incurs significantly less overhead in handling fat writes.

4.6.4 Overall Performance

The performance gain from memory compression in DDR3 memory comes from several sources. First, page faults are reduced because of the increase of effective memory capacity. Second, the over-fetch effect may combine reduce the number of cache misses (because of prefetching), and it also improves bandwidth utilization. Note that we do not include page fault simulation in our evaluation work. Any speedup shown here are results from over-fetch effect. In other words, on top of performance gain from page faults reduction.

Figure 4.19 shows the IPC speedup of single-core workloads. Uncompressed main memory is used as baseline and all IPCs are normalized to it. On average, single core benchmarks show 3.5% improvement with LCP scheme and 5.5% improvement with FM. This is purely from over-fetch effect. We also constructed 100 4-core random workloads using SPEC benchmarks. As bandwidth is more likely to be the performance bottleneck in multi-core system and the over-fetch effect helps reduces bandwidth pressure, the IPC speedup is expected to be more prominent in 4-core workloads. Both LCP and FM have performance gain, reaching 5.1% and 7.5% SMT speedup, respectively.

4.6.5 Over-Fetch Cache

In this section, the performance of OFC (Over-Fetch Cache) is evaluated on single-core benchmarks. A small 4KB is used as OFC to store additionally fetched data. When a memory

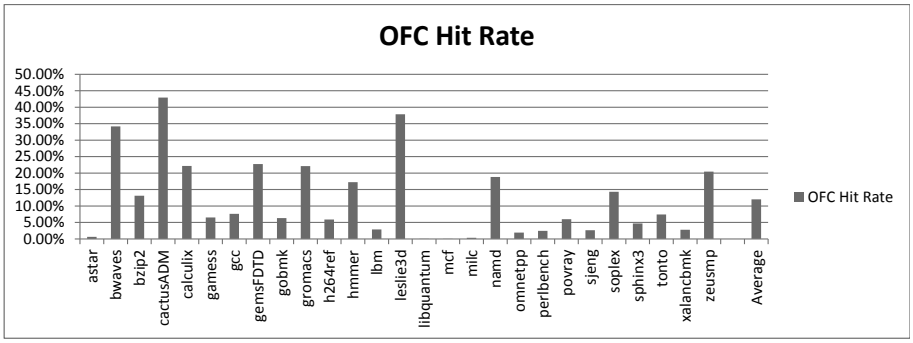


Figure 4.20: Hit rates of OFC (Over-Fetch Cache) for each benchmark

read request reaches memory controller, OFC is checked first to see data is already in the cache. If so, a read request can be served immediately without actual DRAM access.

Figure 4.20 shows the hit rate of OFC. The hit rate can reach as high as 43% for some benchmarks like cactusADM. However, for benchmarks like aster and mcf, the hit rate is virtually zero, possibly because their memory access pattern causes thrashing in the OFC. For other benchmarks including libquantumn and milc, their hit rate is also zero because of their poor compressibility. The average OFC hit rate is 12%.

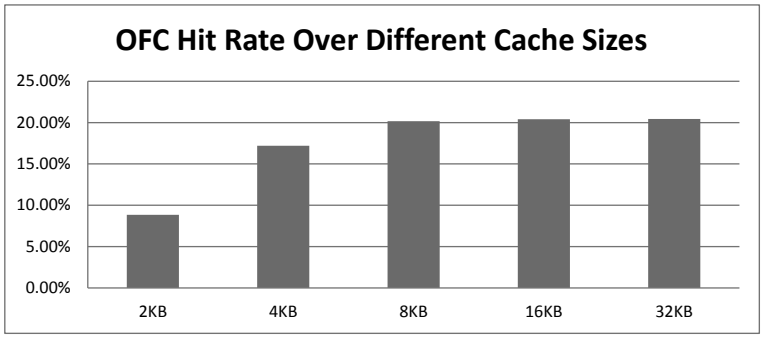


Figure 4.21: OFC hit rate of bwaves benchmark over when OFC size ranges from 2KB to 32KB

Intuitively, hit rate of the over-fetch cache increases with its size. Figure 4.21 shows the OFC hit rate of bwaves benchmark when the size of OFC changes from 2KB to 32KB. As expected, the hit rate improves from 8.85% to 20.44% when cache size increases. Using the cacti cache simulator [32], we calculate that even if a large 32KB OFC is used, the cache only occupies 0.36 mm^2 chip area assuming 32nm fabrication technology. Therefore, OFC is effective with low cost.

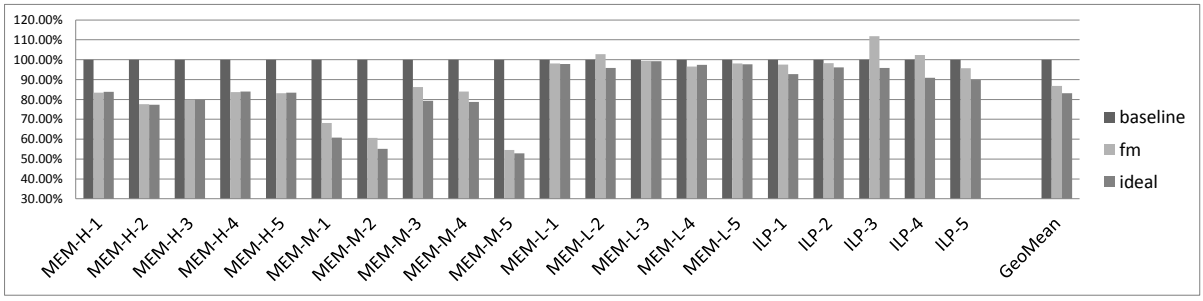


Figure 4.22: Normalized power (full-rank Memory), normalized to full rank non-compressed memory scheme (baseline)

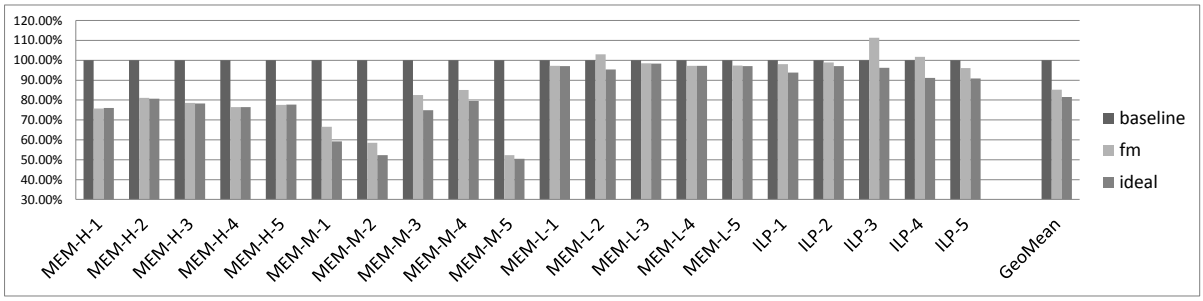


Figure 4.23: Normalized power (32-bit sub-rank Memory), normalized to 32-bit sub-ranked non-compressed memory scheme (baseline)

4.6.6 Memory Power Evaluation

In this section, we will show how much power/energy efficiency is gained through combining sub-ranked DRAM and compressed main memory. Power saving is mainly affected by sub-ranking configuration and block compressibility, and these techniques/factors apply to both FM and LCP without being affected much if at all by different page structures. Thus we only include the FM and FM-ideal simulation. The FM-ideal mode is an ideal case of FM with no overhead operation, thus it presents theoretical upper bound of power saving.

Figure 4.25 shows average power breakdown from all workloads. From the figure, we can tell that the memory background power stays about the same across different sub-ranking configurations with FM. The slight decrease of background power across different sub-ranking configuration is caused by uneven memory requests distribution, bringing in slightly longer power-down time. The I/O termination power is mostly decided by number of data-bus transactions and their sizes. Across all configurations, we see visible saving in this part due to reduced memory traffic. The operation power is determined by the number of micro operations

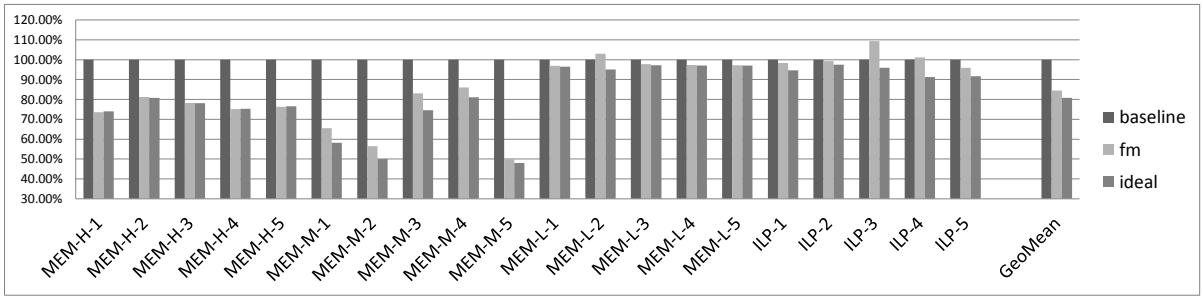


Figure 4.24: Normalized power (16-bit sub-rank Memory), normalized to 16-bit sub-ranked non-compressed memory scheme (baseline)

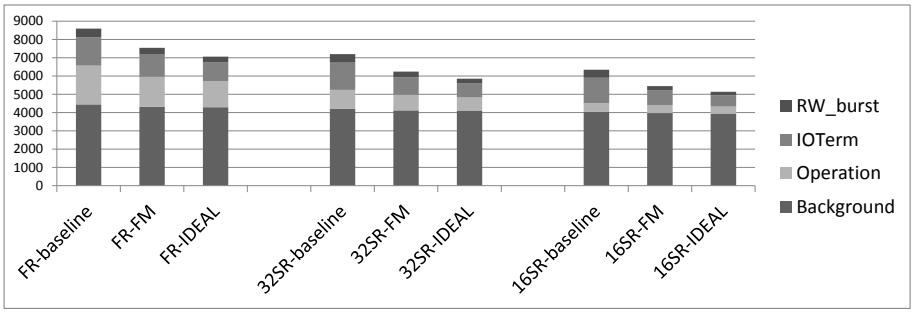


Figure 4.25: Memory power breakdown of multi-core workloads; y-axis is power consumption in micro-watts

in DRAM, like precharge and activation. Whenever a request is fulfilled by the OFC or request combining, a series of such operations can be saved, leading to considerable power savings. The read/write burst (RW-burst) power is saved in similar manner.

Figure 4.22, 4.23 and 4.24 demonstrate power saving of all configurations. Three MEM-M workloads give high power saving, namely MEM-M-1, MEM-M-2 and MEM-M-5. This is because their combined compression ratios are very close to the high compressibility range (50%). They all have considerably high OFC hit ratio ($> 40\%$), saving the DRAM operation power because of less memory accesses. The ILP-3 workload shows extra power consumption in the normal FM mode, mainly due to its benchmark combination. BZip2 and perlbench happen to be in this workload and they have extremely low compression ratio and increase memory traffic because of fat write overhead. Of all workloads, the highest power saving is 45%. On average, full-rank, 32-bit sub-rank and 16-bit sub-rank shows 13%, 15% and 16% power savings, respectively, compared to the corresponding sub-ranked and non-compressed memory.

4.7 Summary

In this paper we present a novel flexible main memory compression design. The main goal of this work is to find a practical memory architecture that adapts to modern computing needs. An advanced page header BMT loaded with pointers to memory blocks makes Flexible Memory as a lightweight memory management utility. By combining advantage of low-overhead compact memory structure, state-of-the-art compression algorithm and techniques to reduce overhead, Flexible Memory is able to obtain an average 1.5x memory effective capacity gain. We further studied its potential of reducing memory traffic by utilizing an Over-Fetch Cache, which also helps achieving an average of 14% of power saving.

CHAPTER 5. FLEXIBLE ECC IN COMPRESSED MEMORY

5.1 Introduction

As DRAM density and capacity scales, memory systems become more prone to memory errors that could lead to system crash or data corruption [28, 40, 5, 6, 64, 41]. One of the most effective way to detect and correct memory errors is to append ECC (Error Correction Code) to data in memory, which requires specially designed devices with extra DRAM chips, forming nine-device DIMM, to hold ECC parity bits. Extra chips incur higher power overhead for keeping and accessing parity bits. For most commonly used SECDED (Single Error Correction Double Error Detection) code, power overhead is about 12.5% of memory subsystem power consumption. Performance of today's portable computers (smart phones) is usually constrained by their battery life. Energy and cooling cost is also one of the major concerns in data centers. Reducing the power overhead without losing reliability protection is crucial to improving performance of these systems. Selective Error Protection presented in Chapter 3 is a effective solution that focuses on lowering overall protection cost. In this chapter, we are investigating in a design to lower the cost of ECC per protected unit capacity. It could be combined with SEP perfectly.

Block-level hardware memory compression is a promising technique in increasing the effective memory capacity and bandwidth without significantly costing significant or even any extra power [13, 48, 49]. Besides, we have designed and presented Flexible Memory in Chapter 4 that leverages advanced memory mapping structures to achieve high capacity gain and lower compression overhead. Other than improving memory capacity, compression can also be helpful in reducing cost of memory reliability protection schemes for several reasons. First, memory compression provides free space to hold ECC code without having too add extra device, like

conventional ECC DIMMs. Second, the ECC bits in freed space does not cost extra power to keep and transfer. Therefore, it is ideal if compression and memory protection can be combined together to provide a very low-cost ECC scheme on commodity DRAM modules.

However, combining ECC and compression is non-trivial. Compressibility of data blocks in memory varies greatly, resulting in very different free space size. Although most blocks are expected to free up enough space for ECC, the other

Previous works [64, 65, 59, 30, 41, 38, 8] attempt to store parity bits together with data to avoid overhead of extra chips for ECC code. By doing so, non-ECC DIMM can have protection without added power overhead. However, at the same time effective capacity is also reduced in order to make room for ECC. This lowers power efficiency per unit capacity. The root reason behind the trade-off is that ECC code space overhead is fixed no matter how or where it is stored. Thus, in order to solve this problem completely, ECC space overhead must be lowered. Memory compression has the potential to reduce or eliminate ECC space overhead, which then lowers power overhead.

It has been proved that memory data has considerable compressibility [49, 48, 13]. With simple compression algorithms like BDI [50], around 50% of capacity can be freed. The freed up space could be used to accommodate ECC code. In ideal case, this do not incur any extra space overhead for ECC. Therefore, there is no associated power overhead.

Some recent research works studied the possibility of having a protected and compressed main memory system without needing to add extra device. MemZip [54] is a compressed memory design with memory reliability in mind. It does not try to utilize memory space freed up by memory compression at all. Instead, all these spaces are dedicated for non-traditional purposes and holding ECC is one of them. Although MemZip only brings minimum overhead, it does not make any promise about protection coverage. Therefore, some memory regions may be unprotected if its data does not compress well. COP [43] cleverly leverages the fact that multiple-bit error in a single memory block is rare. And it is safe enough to use this as a criteria to tell compressed and protected blocks apart from incompressible blocks, thus saving storage overhead of adding extra flags. However, in rate occasions where they are indistinguishable, COP relies on last level cache to store such blocks and prevent them from entering main memory

at all. Also multiple-bit error, though rare, could still happen and cause silent data corruption in COP scheme. Frugal ECC [31] identifies that in a compressed and protected memory design, compression algorithm should be optimized towards high protection coverage instead of good compression ratio. Following this discovery, Frugal ECC optimized compression algorithm and gained an improved protection ratio. However, like MemZip Frugal ECC does not provide the ability to share extra free space to hold overflowed data, which would have to go to a reserved compression exception region and making memory protection more costly.

In a compressed and protected memory, protection coverage is a metric about what percentage of memory can be protected with *free space produced only by compression* without using dedicated memory space like extra chips or reserved storage to host its ECC data. A high protection coverage signifies a lower-cost memory protection system. Improving protection coverage is the one of the most important goals when designing a memory compression and protection scheme.

To further improve protection coverage, we propose **Flexible ECC** that not only utilizes state-of-the-art Coverage-oriented Compression algorithm, but also proposes and employees a new Coverage-oriented block/page structure in this study, to maximize free space sharing between different blocks and OS pages in order to ultimately make free space available for more ECC codes.

Flexible ECC is based on Flexible Memory compression scheme for several reasons. First, Flexible Memory scheme provides flexible memory block layout, i.e. the ability to place a memory block anywhere in memory page and its page header BMT that can be extended for various purposes, which all support flexible ECC code placement. We believe, this could help making best uses of any free space that maybe unevenly distributed to different blocks within same OS page. Second, FM provides the ability to have multiple OS pages to co-exist in same page container and potentially allow them to share part of their free space. We believe Flexible ECC could increase protection coverage with the help of Flexible Memory and state of the art Coverage-oriented Compression algorithm.

Even with all optimization towards improving protection coverage, it is still possible for some workloads to be less compressible and no space available at all for their ECC codes. It

is not acceptable to leave these memory data unprotected. Though Selective Error Protection also leaves part of memory unprotected, their case is completely different ours. SEP carefully picks which part of data needs protection according to their vulnerability or significance, thus partly protecting memory does not lead to proportional loss of reliability. However, unprotected region in our case is caused by compressibility of their data, not their vulnerability. It is entirely possible that some important data has poor compressibility. Therefore, Flexible ECC design must be able to accommodate the extreme cases while keep complication low. This calls for the use of a dedicated memory region with flexible size to hold incompressible pages in these extreme cases.

In this study,

- We discover that other than a compression algorithm designed towards high coverage, flexibility to combine data and free space for ECC code can also improve ECC coverage.
- We present implementation details of Flexible ECC and evaluate its protection and performance.

Rest of this chapter is organized as follows: Section 5.2 gives background information about previous compression only schemes and a few other studies that combines memory compression with protection. Section 5.3 describes ideas behind Flexible ECC design and its detailed implementation. Section 5.5 and Section 5.4 discuss about or experimental methodology and evaluation results, respectively. Lastly, Section 5.6 summarize this work.

5.2 Related Works

In this section, we will discuss previous works related to designing a compressed and protected memory system.

5.2.1 Memory Compression

There are several memory compression works that do not involve memory protection but targeting to capacity gain, like, MXT (Memory Expansion Technology) [57, 1, 58], RMCS (Robust Memory Compression Scheme) [13] and Pekhimenko et al. proposed LCP (Linearly

Compressed Pages) [49, 48]. All these works attempt to achieve great compression ratio, shooting for most capacity gain and minimizing performance/energy overhead associated with their proposed scheme. However, they are not designed to support holding ECC code in their free space.

5.2.2 Memory Compression and Protection

MemZip [54], however, abandons capacity enlargement as a design goal. Instead, it follows same layout of conventional memory system, meaning that even if a memory block is compressed to a smaller size, its allocated space is still 64 Byte (assuming cache-line size of 64 byte). This avoids layout complication of compressed memory and any performance overhead caused by it. Another merit of this design is its large amount of unused memory space, which is the gap between actual memory block size and allocated block size. MemZip could easily hold ECC code in these free spaces with minor modification. But, due to rigid memory layout, ECC code has to tightly follow the word it is protection. And some larger free memory space can not be shared between different blocks, so overall space utilization rate and protection coverage can not be guaranteed.

COP [43] tries to compress each block and embed ECC to saved space to form a compressed and protected block, while leaving other incompressible blocks untouched. Such a design usually requires some dedicated flag bits to indicate whether a block is compressed or not so that when reading it, memory controller knows whether it needs to decompress and verify ECC code or not, otherwise it would become a write-only storage. However, COP identifies that in many cases this flag is not needed because attempting to decompress and verify ECC code a compressed block would usually result in multiple-bit error, which is very unlikely. Thus, COP could accommodate both compressed and uncompressed blocks with little overhead. However, for a small percentage of blocks that can not be distinguished in this way, COP stores them in last level cache only and stop from being evicted into main memory. Performance of this approach is limited by size of last level cache. Besides, COP can potentially hurt system performance because it changes cache eviction and fetch cache behavior, which depends on compressibility of memory data.

Frugal ECC [31] is another scheme design to both compress and protect main memory. It optimizes compression algorithm to improve protection coverage instead of capacity gain. Frugal also provides different protection tiers to improve protection coverage. However, page structure of Frugal ECC prevents space sharing just like MemZip. No block can share their extra space to help compressing another block even if they are in close proximity.

5.2.3 Coverage-oriented Compression

Coverage-oriented Compression (CoC) is proposed in Frugal ECC [31], it is the current state-of-the-art memory compression algorithm that considers protection coverage as a major design goal.

CoC is composed of three major components, namely Fitting Base Delta for data with small value range, exponent compression for floating points data and frequent word pattern for heterogeneously-typed data. FBD is in fact a revised version of Base-Delta-Immediate compression that has larger delta ranges to increase amount of data covered under itself at the cost of losing some compression ratio. Floating point numbers hard to compress because of its avalanche effect, i.e. a slight change of its value may cause many binary bits to flip. This algorithm focuses on exponent and sign part of floating point data while gives up on compressing others parts in order to get a larger coverage. Frequent word pattern follows similar design principal to improve compression coverage for heterogeneously-typed data.

5.3 Flexible ECC Design

Flexible ECC is designed with coverage-centered flexibility in mind. In other word, main goal of Flexible ECC design is to maximize protection coverage with the help of memory layout flexibility.

Protection coverage is defined as percentage of memory that is protected by specifically ECC codes stored in memory space freed by memory compression. Therefore, even if entire memory is protected, protection coverage might not be 100% because some part of memory might still be protected by a dedicated space without reduced cost. The significance is that those memory regions in compression protection coverage incurs very low or no storage overhead

and reduced energy overhead. Thus, increasing protection coverage is equivalent to lower cost memory protection.

Compared to previous memory compression and protection schemes, Flexible ECC improves protection coverage by introducing block/page level mechanisms to help incompressible data find enough available space for their ECC from memory locations that are hard to use because the space may not be adjacent to them. This can work on top of Coverage-oriented Compression because relying CoC alone might not be enough to always grantee high coverage. Any compression algorithm, no matter how good it is at compressing to reduce data size can not compress all possible data values, otherwise it would eventually compress any data value to size zero if such compression algorithm exists and is applied to a set of data over and over to ultimately reach zero, which is obviously unreal.Thus, there are bound to be some memory blocks being incompressible, i.e. not having enough space to hold both its data and ECC, which lowers protection coverage.

To illustrate the insufficiency of compression algorithm in the aspect of improving coverage, we can think of an example like follows: Suppose 8-bit ECC is needed to protect 64-bit of data block and a memory system contains 10 such blocks. It is possible that compression algorithm is not able to provide 80-bit of free space for all blocks. Even in a better case where 80-bit is generated by compression, the uneven distribution of it might cause problem. It is possible that only two of the blocks are giving out the 80-bit space, but this space is not made available to other eight blocks. This leads to a 20% low coverage.

Based on this observation, Flexible ECC shifts focus from compression algorithm to memory block/page management to seek improvement opportunity. We can still consider previous example. If those two blocks are able to share their free space to the rest eight blocks, then this memory system can reach 100% protection coverage instead of 20%. Or in other words, a major goal of Flexible ECC is to make space available to a wider range of consumers and give consumers more choices to store their ECC. It may seem simple to do, but most memory compression management scheme do not readily support flexibility like this. Therefore, A well designed memory compression framework with high flexibility is needed to meet this requirement.

Accordingly, we consider Flexible Memory as the best choice for Flexible ECC for several reasons: 1) Flexible Memory has the ability to provide a compact layout that makes efficient use of available space. 2) Flexible Memory compression scheme natively supports flexible layout of pages and blocks with high degree of freedom, it. 3) Flexible Memory is mostly insensitive to what compression algorithm is used.

Other than memory compression management framework, Flexible ECC has two other major components, namely compression algorithm and ECC code. To achieve best protection coverage, Flexible ECC compress data with state-of-the-art CoC from Frugal ECC. As for ECC code, we choose most popular SECDED to generality.

Even though the basic idea behind compressed and protected memory of applying compression on data to obtain free memory space to hold ECC is simple, there are quite some important design/implementation details unclear. In rest of this section, we will further discuss these details to make Flexible ECC a practical design..

5.3.1 Ordering of Compression and ECC Generation

In order to have a compressed and protected memory, compression operation and ECC generation are naturally needed. However, it is not straightforward as in what order they should be applied to data, nor is it clear whether ECC should be generated from compressed form of data or uncompressed form or whether ECC should be compressed at all, all of which could make non-trivial difference in both performance and reliability.

To append ECC parity code to data in a compressed memory, there are several possible options available to do so. These options vary from each other in subtle ways. For discussion in this section, we would use following notation to make the representation clearer. $cps()$ stands for compression algorithm while $ECC()$ represents chosen ECC coding method. $data$ stands for input data. Then we have four options to compress and protect data.

First option is $cps(data + ECC(data))$, namely generating ECC code off of uncompressed data and concatenating them together to form a regular uncompressed ECC word. At last, compress the ECC word altogether. This is the most natural way to compress and protect, the only difference between it and traditional protection scheme the last step of compression.

However, for many frequency-based algorithms like Huffman encoding or value/pattern-based compression algorithms like FPC, FPV and BDI, this method greatly decreases compressibility of data, because ECC code is heterogeneous to type of data it is protection and likely to bring in high entropy or irregularity. For algorithms that rely on having certain kind of pattern or small value range to compress, such impact can sharply reduce compressibility or make the data incompressible. Therefore, this option is not a good choice.

Second option is $cps(ECC(data) + cps(data))$. Which computes ECC parity code before compression of data but append to it after compressed data. Then last step is compress combined results. This does not hurt compressibility of data by adding ECC code like in previous option any more. However, when ECC parity bit is compressed, it is weakened. Each ECC parity bit is added to increase the Hamming distance between correct word and possible erroneous word. When compressed, although the ECC bits are packed into a smaller space, which is equivalent to having a shorter Hamming distance. Another more straightforward way to understand it is that ECC code could suffer multi-bit error when there is actually only one DRAM cell flip. This would ultimately hurt protection strength.

Besides, given generally poor compressibility of ECC parity bits, last compression would not see much decrease in word size. Even in case it does compress successfully, protection is weaker. Therefore, this option should not be considered either.

Third option can be represented as $ECC(cps(data)) + cps(data)$. The difference between this option and last one is that instead of applying ECC to uncompressed data, it tries to append ECC to compressed data and then append generated parity bits to it. This method requires compression of data only once before it is protected, maintaining the valuable data pattern that can be utilized to improve compression. Afterwards, ECC code is generated and stored in memory without being compressed. This option preserves both data compressibility and ECC protection strength.

5.3.2 Coverage-Oriented Page/Block Layout Design Principles

After determining the generation process of protected block including both data and ECC, next design target would be finding a place to store it in main memory. Their placement is a non-trivial design issue that could greatly affect both protection coverage and performance.

Based on flexibility in data placement provided by FM and reach satisfactory performance, we design a page/block layout following some principles.

Space sharing with high degree of freedom should be supported. This is the reason we choose to base Flexible ECC on FM, because of its great flexibility to place a memory block or page anywhere it is needed and the ability to extend its header BMT for reliability usages. This is mainly used to make space sharing easier between blocks and pages. Without this property, it would be hard to utilize all free spaces available and result in a lower coverage.

Both data and ECC should be easily addressable. Many compressed main memory scheme complicates data layout and makes addressing harder, FM-based Flexible ECC is no exception. This is the side effect of having great flexibility. However, FM embeds a extensible page header that helps make it easier to address blocks and a customized page table to help address pages and sub-pages. Flexible ECC should take advantage of them and keep data and ECC addressable.

Two times access to retrieve data and ECC separately should be avoided. It is possible for data and its ECC parity bits be far away from each other in terms of memory space. It is possible DRAM would need to issue two read commands to two different row buffer to read both ECC and data, which could potentially double access latency. Therefore, certain limitation and/or optimization should be included to avoid cases like this.

5.3.3 Block-level Layout

Block-level layout refers to a set of methods to manage blocks within same compressed page. In FM, a simple BMT is embedded into each compressed page. It is essentially a set of $< offset, size >$ pairs acting as pointers pointing to each block, either compressed or

uncompressed, in that page. Because it explicitly stores sizes and offsets of all blocks, each block can be placed anywhere in the page and can have arbitrary size as needed.

Such a feature is very useful to Flexible ECC because BMT can be used to point a data block to its ECC parity bits with some modifications. This gives a data block more options regarding where to store its ECC bits, which translates to high protection coverage.

Therefore, we enhance BMT to FECC-BMT that supports more advanced features related to improving protection coverage, like Easy Block Exception Handling and Block Space Sharing (More details will be given later).

A memory block in Flexible ECC be one of the following four types based on their compressibility and ECC code status:

- Compressible. A block that is compressible enough to hold at least its own ECC code and not sharing its space to other blocks.
- Space Borrower. A block that is not compressible enough thus is borrowing space from another block to hold its ECC code.
- Space Lender. A block that is very compressible such that is is able to lend its own space to other blocks.
- Easy Exception Block. A block that is not compressible and not able to find another memory block that has extra space.

Examples of these four blocks types are presented in Figure 5.2.

Figure 5.1 shows the structure of an FECC-BMT entry, all of its ECC flag value and correspondence to above four block types. Compared to original BMT, FECC-BMT adds two fields named ECC Flag and ECC Block ID. Note that ECC Block ID is different from its own Block ID, which is not explicitly stored in BMT. Combining ECC Flag and ECC Block ID, FECC-BMT could support space sharing between blocks within same page.

In the case of **Compressible Block**, as first block in Figure 5.2, ECC Flag are set to be '00', indicating that its ECC code are stored right after its data. ECC Block ID is not used in

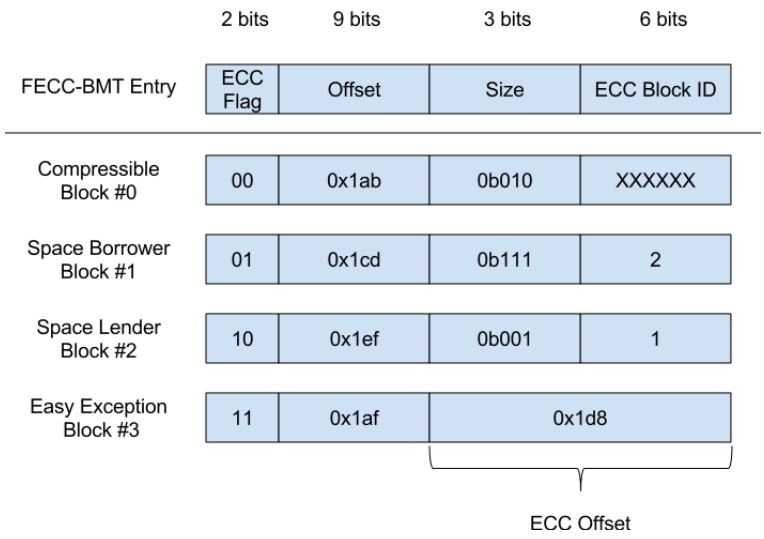


Figure 5.1: Structure of Enhanced BMT in Flexible ECC, and four examples of its use case with different ECC flag values.

this scenario. To find its ECC code, controller only needs to read out its trailing data without extra addressing overhead.

Another type of memory block is called **Space Borrower**. It is the second block example in Figure 5.2. A Space Borrower block does not have great compressibility that has to borrow space from another block in order to be covered by ECC protection. ECC Flag for a Space Borrower is set to '01' to show its status. ECC Block ID is used indicate which block is providing needed space for it. When memory controller reads a Space Borrower, it can find its ECC code using ECC Block ID.

Space Lender is actually the pairing part of a space borrower. It provides the space needed by **Space Borrower**. A space lender is very compressible such that compressing it frees up more space than its own EC need. So it could help another block by sharing that extra space and become a Space Lender. Its ECC Flag bits are set to '10' and ECC Block ID is pointing to its borrower. The reason we need to include ECC Block ID is because when fat-write happens to lender, its may lose its ability to share space any more. Thus, its borrower must be found and notified in order to find another lender to hold its ECC. Otherwise, error protection capability or even data would be lost.

Another type of block are called Easy Exception Blocks. Because Flexible ECC is based on FM, which allows free space be placed almost anywhere in the page. These free spaces

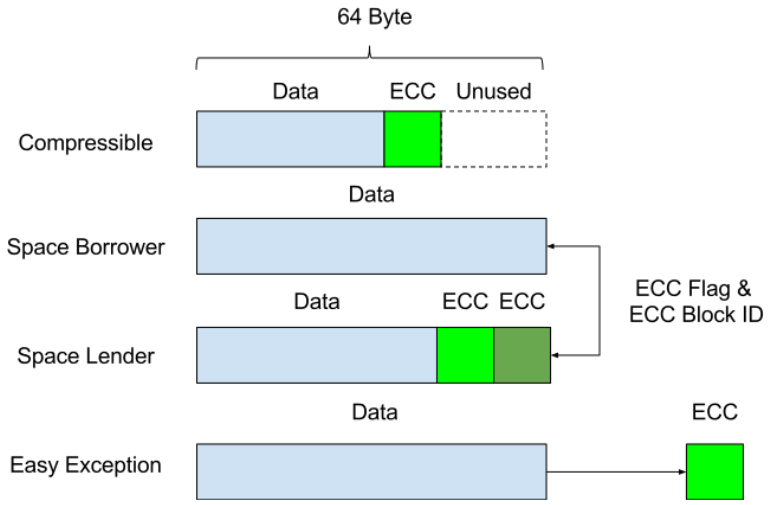


Figure 5.2: Examples of four types of blocks.

may be too small for entire memory blocks, but they act as buffer area for block expansions or reorganizations in FM. In Flexible ECC, we can make use of them by storing ECC code in them. They are especially useful in those cases where a block is incompressible and no other block is allocated enough space to share. An incompressible can then pick its nearest free space, even though it does not belong to any other block, and claim it as its dedicated ECC block. Nearer space is preferred because they are more likely to be stored in same DRAM page and can be accessible without having bank conflict. In previous works, such a block with no compressibility at all has no choice but be stored in exception region with high overhead, while Flexible ECC can handle cases like this easily with little or no overhead when there is any free space in same page. Therefore, we call blocks like this Easy Exception Blocks.

Easy Exception Blocks are indicated by ECC Flag value of '11'. Unlike other types of blocks, EEB does not need size field any more because its size is always 64 Byte, otherwise, it would not be incompressible. And 3-bit size field is then combined with 6-bit ECC Block ID to form a 9-bit ECC Offset field, which is enough to indicate any sub-block in same page. In the example shown in Figure 5.2, ECC of block #3 is placed in sub-block 0x1d8 indicated by ECC Offset field.

In a word, thanks to high degree of flexibility inherited from FM and appropriate support from FECC-BMT, Blocks within same page can freely share space with each other in order to improve overall coverage. Any small chunk of free space can also be fully utilized to cover

incompressible pages with very low cost. In other word, as long as there However, this is based on the presumption that their owner page could provide enough free space. Or in other word, if a page is not compressible enough, block-level coverage optimization within that page would not be able to help because of page size hard limit.

5.3.4 Page-level Layout

Given that block-level optimization efficiently utilizes available space within a page to effectively improve protection coverage, and this is based on the presumption that OS page could provide enough space. It is important to have page-level layout optimization too, which can work together with block-level layout optimization towards achieving higher coverage.

Therefore, we design a page-level structure such that pages with limited available space can utilize space from its peers efficiently to gain needed space.

Following the design of FM, Flexible ECC also manage pages in page container. FM requires a page to be stored only within one page container to avoid excessive bank conflicts when reading a page. However, this rule is in fact too tight when protection coverage is primary design goal instead of performance. In many cases, we could alter the design a little bit to allow a page go across page container boundary so that a less compressible page and a more compressible page could be paired together and share space.

Pairing two pages that are in same page container is trivial and usually unnecessary. Because if more than one pages are stored in a single page container, that means both of them are already compressed to smaller sizes. In order to find free space in either one of the page only requires a page expansion. Even in some cases, like one page is in a bad position that page expansion could cause undesired overhead, an extra page table can be added to indicate that certain sub-page is dedicated to hold overflow data and ECC with structure.

Therefore, we focus our discussion on inter page container page pairing, which includes pages that do not compress well. We try to find the best layout that can pair two pages together so that they can share free space to improve coverage while minimizing access overhead.

Figure 5.3 shows a layout to efficiently utilized extra space from one page to compensate for lack of space in another page, we call it page pairing. As its name suggests, it pairs two

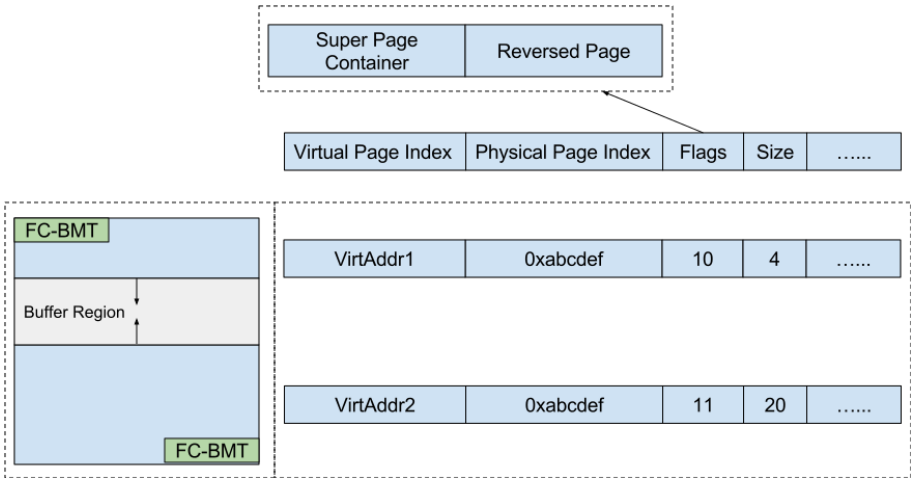


Figure 5.3: Page pairing in one super page container and their page table entries.

pages together and assign a 8KB super page container to them instead of 4KB regular page containers. One of these two pages should start from beginning of super page container whereas the other page should start from the end of it and grow reversely as we see in Figure 5.3 that both first and second page grow into the mutual buffer.

The page in reverse order would be structured slightly different from other pages in its BMT address and block offset calculation. Such a small difference should not cause a notable difference in performance or coverage.

There are several benefits of page pairing structure. The most important one is its better usage of free space for the page with less compressibility because mutual buffer area is always set in middle of the super page container and each page could claim or release part of it to gain or lose free space. Besides, the convergence-style layout keeps most part of each page within either upper of lower half 4 KB area, which is aligned with DRAM row size. This helps reduce bank conflicts.

Obviously, in order to support a page-pairing structure like this, page table must be revised. Traditionally when two virtual pages are mapped to same physical page, it is generally because of shared memory. However, page-pairing structure map two virtual pages to different parts instead of sharing same memory region. Therefore, some modification to OS memory management logic is necessary. Besides, two additional flags are needed. These flags include a Super-Page-Container flag to show if this page is mapped to an double size (8 KB) page

container, instead of regular (4 KB) container. Another flag (Reversed Page flag) is needed to show which of the two pages are placed at the end of page container with reversed ordering.

After enabling page-level space sharing through supporting page-pairing structure, a page with insufficient space can utilize space from another page if such a page with sufficient space exists. Even if OS could not find an appropriate pairing page at the moment, it could pre-allocate an super page container for it and wait for compressible page to appear. However, number of such pages should be strictly controlled otherwise a program with less compressible data set could take up twice as much memory as it should have needed.

After combining block-level and page-level structure optimization, space sharing is greatly improved. A block can utilize any space available in its page and a page is allowed to make use of extra space from other pages. We believe this can help improve protection coverage.

5.3.5 Exception Memory Region

Though page-pairing layout helps improve protection coverage, it still can not guarantee a 100% coverage even combined with block-level optimization discussed before. This is because compressibility and memory capacity poses a hard limit, which can not be overcome with enabling space sharing. As long as this hard limit exists, there is always the possibility that some data can not hold its own ECC data either for poor compressibility or simply lack of available memory capacity. In this case, incompressible page should be accommodated in a dedicated area in memory space together with its ECC in order to keep memory space 100% covered.

However, setting up an Exception Memory Region involves complications too, like how to minimize storage consumption and performance overhead. In fact, there are two major problems in this issue.

First is address mapping in EMR. Unlike other parts of memory where memory blocks are either addressed by FECC-BMT with explicit pointers or by implicit binary decomposition mapping, Memory blocks and their ECCs together form none-power-of-two 72-byte blocks and this causes trouble in address mapping. Since their sizes do not vary block by block, nor do they need out-of-order block placement, it is a waste of space to have FECC-BMT included in each

page. For OS and user-level applications, the existence of ECC code should be transparent, which makes address mapping even more difficult. Luckily, Segmented-BCRM memory system design for Selective Error Protection introduced in Chapter 3 would be a perfect solution for this problem only with some minor modifications.

Second problem is about sizing of EMR. On one hand, a small EMR may result in excessive re-sizing that could involve large chunks of data movement, causing high latency in some memory operations. One the other hand, a large EMR could waste memory space. We therefore, implement EMR in a balloon fashion that blows up gradually when a simple prediction shows that in the near future more space is needed and shrinks down when it is predicted that less space would be needed. The prediction is based on EMR utilization and delta of page numbers in EMR in unit time.

However, in the case of failing to increase size of EMR due to memory capacity or utilization issues, some pages would have to be evicted from physical memory space into hard disks as a last resort. In the future, if it becomes needed, Virtual Memory would pick another least used page and replace it with requested page.

5.4 Experimental Methodology

5.4.1 Benchmarks and Workloads

We construct single-core workloads with the SPEC CPU2006 suite [20] running with the reference input data set. Only single-core workloads are constructed because compressibility and protection coverage does not change according to number of cores or number or programs running simultaneously.

During simulation, all workloads are fast-forwarded to skip initialization period as well as a 500M instruction warm-up period. Semaphores are inserted to make sure simulations are deterministic regarding starting point.

Table 5.1: Simulated System Configuration

Processor 1 core	x86_64 ooo core 3.2GHz
L1 Data Cache	Private, 64 Byte Block
	8-way 64 KB/Core
L2 Cache (LLC)	64Byte Block
	8-way, 4MB Shared
DRAM	2 Channels DDR3 800MHz
Model	MT41J256M8-32M3gx8x8
Bus Frequency tCL/tRP/tRCD tRRD/tRC/tRAS	800 MHz 11/11/11 5/39/28
VDD/IDD0 IDD2P0/IDD2P1/IDD2N IDD3N/IDD3P IDD4R/IDD4W/IDD5	1.575 V/95 mA 12/37/43 mA 50/55 mA 156/145/195 mA
Cps/Decps Latency	5 cycles
ECC Code	(72, 64) SECDED
Memory Sub-Ranking	32-bit sub-ranks
Simulation point length	100M Inst

5.4.2 Simulator and Configuration

Our simulator is based on Marssx86 [44], a cycle-accurate full system x86_64 simulator. We also integrated modules to Flexible Memory and Flexible ECC. CoC (Coverage-oriented Compression) algorithm is implemented according to descriptions of it in Frugal ECC [31]. This way, Flexible ECC and Frugal ECC can be compared fairly such that only memory organization instead of difference in compression algorithm would make a impact on protection coverage. Besides, An in-house detailed model of DDR3 memory system is integrated to provide DRAM statistics. All DRAM power statistics are calculated according to the Micron power calculator [23, 24]. Other detailed simulation parameters can be found in Table 5.1.

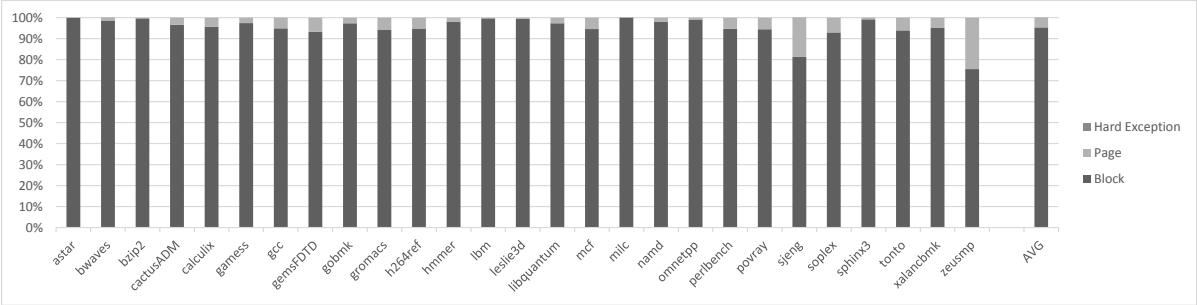


Figure 5.4: Distribution of compression exception handling in Flexible ECC. Block (blue bar) is exception handled by block-level methods; similarly, page means page-level methods; the rest are hard exceptions that are higher cost and need dedicated memory region.

5.5 Evaluation

5.5.1 Compression Exceptions

Compression exceptions are enemies to efficiency of memory compression and protection scheme. Each compression exception means extra resources including storage space and energy budget must be spent to cover the exception.

Frugal ECC relies on CoC to reduce number compression exceptions while Flexible ECC also has structural flexibility on top of CoC to reduce exceptions. Figure 5.4 shows the how compression exceptions are handled in Flexible ECC.

In this figure, we can clearly see that block-level handling, including block space sharing and easy exception (separate ECC block) takes care of most exceptions. On average, around 95.3% of them are handled by them. Out of the rest exceptions, 4.6% are handled by page-level methods, specifically page pairing with super page container. Overall less 0.01% of all exceptions are costly exceptions involving using reserved memory region and possibly cause two times error. On contrary, if Frugal ECC provides same level of protection (without considering half compression protection), all of these exceptions would be categorized as hard or costly exceptions.

We can see that by introducing another layer of memory management that helps evening compressibility and free space, cost of memory compression and protection can be reduced by a large margin with Flexible ECC.

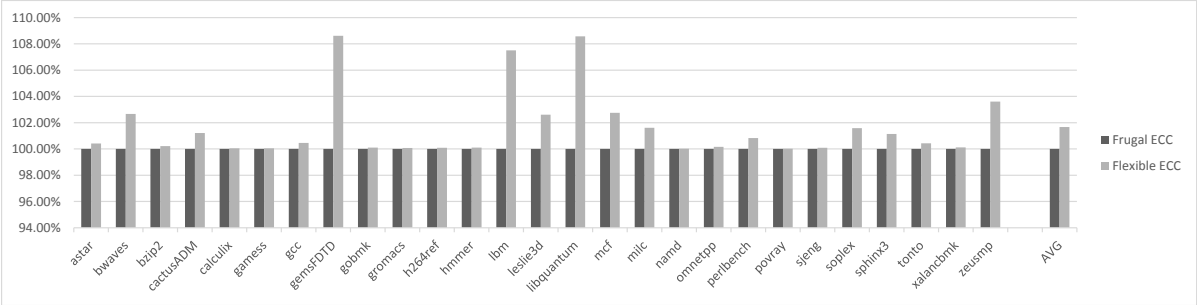


Figure 5.5: IPC of all workloads normalized to that of Frugal ECC.

5.5.2 Performance

Performance improvement is tightly coupled with the reduction of compression exceptions. Frugal ECC places overflow data from compression exception to a dedicated region, and accessing this region may cause an extra memory access than usual and second (extra) access might incur bank conflict which would bring in more latency. This latency can be anywhere between tens of nanoseconds to over a hundred nanoseconds.

Flexible ECC, however, provides multiple lower cost ways to handle compression exception and related overflow data. And we have seen in Section 5.5.1 majority of them can be handled by page or block-level flexible layout without causing significant overhead.

Figure 5.5 presents the performance of Flexible ECC in terms of IPC when normalized to Frugal ECC as baseline. IPC improvement varies from 0.04% to 7.51% depending on workloads compressibility and memory intensity. On average, this is a 1.67% IPC speedup. The source of speedup is from saving unnecessary memory operations, thus shortening memory access latency.

5.6 Summary

We believe that compression is a perfect match with ECC-based memory protection because it creates usable space out of crowded memory space to hold ECC codes with minimal or sometimes no cost. However, combining them is non-trivial because compression targets for high capacity while memory protection aims to provide high protection coverage with little focus on capacity. Therefore, we study the possibility of designing a Flexible ECC scheme based on Flexible Memory, which provides high degree of freedom to customize both layout of

blocks within an OS page and layout of pages in entire memory space. Such flexibility provides the ability to adjust memory layout to enable space sharing at many different levels so that Flexible ECC could make the most use out of freed space to improve protection coverage. Our experiments show that Flexible ECC greatly reduces high cost compression exceptions than previous state-of-the-art, this overall makes memory compression and protection schemes more practical and efficient.

CHAPTER 6. CONCLUSION AND FUTURE WORK

Large-scale computers and clouds based on them have become a much more important part in computing systems. Many computing tasks have been migrated from individual devices to large-scale computers located in dedicated data centers. There are many reasons behind this paradigm shift. Out of them, easier management high power-efficiency, strong reliability and availability are most dominating reasons. However, when large-scale computers keep scaling up by including more nodes, it has become challenging to maintain these properties. Memory system, as a crucial part to many computing systems, is also facing these challenges.

We first present an efficient memory SEP mechanism to support a memory SEP system using commodity memory modules and devices. It partitions the whole set of DRAM rows into two regions, a non-protected region and an ECC protected region. A new address mapping scheme called parameterized BCRM is proposed to map physical memory address into DRAM device address components, and two efficient logic designs are presented. With this support, the OS may dynamically adjust the sizes of the ECC protected region and the non-protected region according to application demands. Our evaluation shows that the design incurs negligible performance overhead and improves memory energy efficiency.

We then present Flexible Memory, a novel flexible main memory compression design. The main goal of this work is to find a practical memory architecture that adapts to modern computing needs. An advanced page header BMT loaded with pointers to memory blocks makes Flexible Memory as a lightweight memory management utility. By combining advantage of low-overhead compact memory structure, state-of-the-art compression algorithm and techniques to reduce overhead, Flexible Memory is able to obtain an average 1.5x memory effective capacity gain. We further studied its potential of reducing memory traffic by utilizing an Over-Fetch Cache, which also helps achieving an average of 14% of power saving.

To further reduce storage and energy cost of memory protection, we designed Flexible ECC that makes full use of available spaces in memory to hold ECC code for other blocks even though the space is not adjacent to blocks. This provides an extra layer of system design that can help improve protection coverage, which directly translates to lower cost of memory protection. Flexible ECC is insensitive to compression algorithm choice or ECC code choice, and thus is able to work with some priori compression and protection schemes. Our experiments show that Flexible ECC provides highest known protection coverage ratio when compared with previous state-of-the-art.

In summary, we propose three memory designs aiming to improve memory capacity, bandwidth and reliability while keeping power cost low. They can be applied on a wide range of computer systems. For small-scale systems like personal hand-held smart phones or tablets that do not have the luxury to include large memory capacity or strong memory protection because of hardware cost and battery lifetime concern, all three proposed schemes can help improve in both realms. For larger-scale computers including personal computers and desktop workstations, their growing concern over memory reliability can be resolved by implementing either SEP or Flexible ECC, that provides low-cost ECC protection on commodity devices. For extremely large-scale computers in data centers, Flexible Memory can be used to improve performance of their memory system by giving low-cost capacity enlargement, higher bandwidth resource and great energy efficiency. SEP and Flexible ECC can also be applied on them to potentially strengthen reliability by using stronger ECC code and reduce reliability-related energy cost. Overall, various types of computing systems could all benefit from these schemes to have larger memory capacity, higher effective bandwidth and become more reliable at very low cost.

In the future, we would like to extend our research topics to memory systems beyond DRAM to newer memory technologies especially Non-Volatile Memory. They are considered promising candidates to replace DRAM technology. However, they are at early stage of development and still troubled by problems like limited write endurance, long access latency and high energy cost per operation. We see an opportunity to have more sophisticated memory system design to have an impact in this area and potentially make new technologies more practical and useful.

BIBLIOGRAPHY

[1] Abali, B., Franke, H., Poff, D. E., Saccone, R., Schulz, C. O., Herger, L. M., and Smith, T. B. (2001). Memory expansion technology (mxt): software support and performance. *IBM Journal of Research and Development*, 45(2):287–301.

[2] Ahn, J. H., Jouppi, N. P., Kozyrakis, C., Leverich, J., and Schreiber, R. S. (2009a). Future scaling of processor-memory interfaces. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–12.

[3] Ahn, J. H., Leverich, J., Schreiber, R. S., and Jouppi, N. P. (2009b). Multicore dimm: An energy efficient memory module with independently controlled drams. *Computer Architecture Letters*, 8(1):5–8.

[4] Alameldeen, A. R. and Wood, D. A. (2004). Frequent pattern compression: A significance-based compression scheme for l2 caches. *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, 1500.

[5] Baumann, R. (2005). Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266.

[6] Borucki, L., Schindlbeck, G., and Slayman, C. (2008). Comparison of accelerated DRAM soft error rates measured at component and system level. In *Proc. of IRPS*, pages 482–487.

[7] Chen, G., Kandemir, M., Irwin, M. J., and Memik, G. (2005). Compiler-directed selective data protection against soft errors. In *Proc. of DAC*, pages 713–716.

[8] Chen, L., Cao, Y., and Zhang, Z. (2013). E3CC: A memory error protection scheme with novel address mapping for subranked and low-power memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):32:1–32:22.

[9] Corp., I. Intel 64 and ia-32 architectures software developer manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. Accessed: Oct. 1st 2014.

[10] Danilak, R. (2006). Transparent error correction code memory system and method. US Patent 7,117,421.

[11] Dell, T. J. (1997). A white paper on the benefits of chipkill-correct ecc for pc server main memory. *IBM Microelectronics Division*, pages 1–23.

[12] Denning, P. J. (1970). Virtual memory. *ACM Comput. Surv.*, 2(3):153–189.

[13] Ekman, M. and Stenstrom, P. (2005). A robust main-memory compression scheme. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 74–85.

[14] Fan, X., Weber, W.-D., and Barroso, L. A. (2007). Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 13–23.

[15] Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., and Brightwell, R. (2012). Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 78.

[16] Gao, Q. (1993). The Chinese remainder theorem and the prime memory system. In *Proc. of ISCA*, pages 337–340.

[17] Govindavajhala, S. and Appel, A. (2003). Using memory errors to attack a virtual machine. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 154–165.

[18] Haertel, M. J., Polzin, R. S., Kocev, A., and Steinman, M. B. (2012). Ecc implementation in non-ecc components. US Patent 8,135,935.

[19] Hamming, R. (1950). Error correcting and error detection codes. *Bell System Technical Journal*, 29(2):147–160.

[20] Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17.

[21] Hsiao, M. (1970). A class of optimal minimum odd-weight-column SEC-DED codes. *IBM Journal of Research and Development*, 14(4):395–401.

[22] Hwang, A. A., Stefanovici, I., and Schroeder, B. (2012). Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *Proc. of ASPLOS*, pages 111–122.

[23] Inc., M. T. Micron system power calculator. http://www.micron.com/~/media/Documents/Products/Technical%20Note/DRAM/TN41_01DDR3_Power.pdf. Accessed: Aug. 4th 2014.

[24] Inc., M. T. Micron system power calculator. http://www.micron.com/~/media/Documents/Products/Power%20Calculator/DDR3_Power_Calc.XLSM. Accessed: Aug. 4th 2014.

[25] Intel Corproation (2015). Intel 64 and IA-32 architectures optimization reference manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[26] Jacob, B., Ng, S. W., and Wang, D. T. (2008). *Memory Systems Cache, DRAM, Disk*. Morgan Kaufmann.

[27] Jian, X., Duwe, H., Sartori, J., Sridharan, V., and Kumar, R. (2013). Low-power, low-storage-overhead chipkill correct via multi-line error correction. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 24:1–24:12.

[28] Johnston, A. H. (2000). Scaling and technology issues for soft error rates. In *Proc. of Annual Conference on Reliability*.

[29] Kalampoukas, L., Nikolos, D., Efstathiou, C., Vergos, H. T., and Kalamatianos, J. (2000). High-speed parallel-prefix modulo 2n-1 adders. *IEEE Transactions on Computers*, 49(7):673–680.

[30] Kim, J., Sullivan, M., and Erez, M. (2015a). Bamboo ecc: Strong, safe, and flexible codes for reliable computer memory. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 101–112.

[31] Kim, J., Sullivan, M., Gong, S.-L., and Erez, M. (2015b). Frugal ecc: efficient and versatile memory error protection through fine-grained compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 12:1–12:12.

[32] Labs, H. Cacti 5.3. http://quid.hpl.hp.com:9081/cacti/index.y. Accessed: Aug. 2nd 2014.

[33] Lee, J.-S., Kim, S.-D., and Weems, C. C. (2002). Performance analysis of a selectively compressed memory system. *Microprocessors and Microsystems*, 26(2):63–76.

[34] Lee, K., Shrivastava, A., Issenin, I., Dutt, N., and Venkatasubramanian, N. (2006). Mitigating soft error failures for multimedia applications by selective data protection. In *Proceedings International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 411–420.

[35] Lefurgy, C., Rajamani, K., Rawson, F., Felter, W., Kistler, M., and Keller, T. (2003). Energy management for commercial servers. *Computer*, 36(12):39 – 48.

[36] Lei Fan, M. R. (2004). Implementation and energy analysis of base-delta-immediate compression. Technical report.

[37] Li, C., Ding, C., and Shen, K. (2007). Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*.

[38] Mehrara, M. and Austin, T. (2008). Exploiting selective placement for low-cost memory protection. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(3):14:1–14:24.

[39] Micron Technology, Inc. (2014). DDR3 SDRAM MT41J256M8-32 Megx8x8Banks. http://html.alldatasheet.com/html-pdf/506436/MICRON/MT41J256M8/9219/41/MT41J256M8.html.

[40] Mukherjee, S., Emer, J., and Reinhardt, S. K. (2005). The soft error problem: an architectural perspective. In *Proc. of HPCA*, pages 243–247.

[41] Nair, P. J., Kim, D.-H., and Qureshi, M. K. (2013). ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates. In *Proc. of ISCA*, pages 72–83.

[42] Pagiamtzis, K. and Sheikholeslami, A. (2006). Content-addressable memory (cam) circuits and architectures: a tutorial and survey. *Solid-State Circuits, IEEE Journal of*, 41(3):712 – 727.

[43] Palframan, D. J., Kim, N. S., and Lipasti, M. H. (2015). Cop: To compress and protect main memory. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pages 682–693.

[44] Patel, A., Afram, F., Chen, S., and Ghose, K. (2011a). Marss: A full system simulator for multicore x86 cpus. In *Proceedings of the 48th Design Automation Conference*, pages 1050–1055.

[45] Patel, A., Afram, F., Chen, S., and Ghose, K. (2011b). MARSSx86: A full system simulator for x86 CPUs. In *Proc. of DAC*, pages 1050–1055.

[46] Patel, R. A. and Boussakta, S. (2007). Fast parallel-prefix architectures for modulo 2n-1 addition with a single representation of zero. *Computers, IEEE Transactions on*, 56(11):1484–1492.

[47] Patterson, D. A. and Hennessy, J. L. (2008). *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., 4th edition.

[48] Pekhimenko, G., Mowry, T. C., and Mutlu, O. (2012a). Linearly compressed pages: A main memory compression framework with low complexity and low latency. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 489–490.

[49] Pekhimenko, G., Seshadri, V., Kim, Y., Xin, H., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. (2013). Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–184.

[50] Pekhimenko, G., Seshadri, V., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. (2012b). Base-delta-immediate compression: practical data compression for on-chip caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 377–388.

[51] Reed, I. S. and Solomon, G. (1960). Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304.

[52] Schroeder, B., Pinheiro, E., and Weber, W.-D. (2009). DRAM errors in the wild: A large-scale field study. In *Proc. of SIGMETRICS*, volume 37, pages 193–204.

[53] Semiconductor, L. (2012). Rd1025 - ecc module. *Reference Design 1025*.

[54] Shafiee, A., Taassori, M., Balasubramonian, R., and Davis, A. (2014). Memzip: Exploring unconventional benefits from memory compression. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 638–649.

[55] Sridharan, V., DeBardeleben, N., Blanchard, S., Ferreira, K. B., Stearley, J., Shalf, J., and Gurumurthi, S. (2015). Memory errors in modern systems: The good, the bad, and the

ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310.

[56] Teng, M.-H. (1983). Comments on "the prime memory systems for array access". *ACM Transactions on Computers (TC)*, C-32(11).

[57] Tremaine, R. B., Franaszek, P. A., Robinson, J. T., Schulz, C. O., Smith, T. B., Wazlowski, M. E., and Bland, P. M. (2001a). Ibm memory expansion technology (mxt). *IBM Journal of Research and Development*, 45(2):271–285.

[58] Tremaine, R. B., Har, D., Mak, K.-K., Smith, T. B., Wazlowski, M., and Arramreddy, S. (2001b). Pinnacle: Ibm mxt in a memory controller chip. *IEEE Micro*, 21(2):56–68.

[59] Udipi, A. N., Muralimanohar, N., Balasubramonian, R., Davis, A., and Jouppi, N. P. (2012). LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems. In *Proc. of ISCA*, pages 285–296.

[60] Udipi, A. N., Muralimanohar, N., Chatterjee, N., Balasubramonian, R., Davis, A., and Jouppi, N. P. (2010). Rethinking dram design and organization for energy-constrained multi-cores. *ACM SIGARCH Computer Architecture News*, 38(3):175–186.

[61] Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24.

[62] Yang, J. and Gupta, R. (2002). Frequent value locality and its applications. *ACM Trans. Embed. Comput. Syst.*, 1(1):79–105.

[63] Yang, J., Zhang, Y., and Gupta, R. (2000). Frequent value compression in data caches. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 258–265.

[64] Yoon, D. H. and Erez, M. (2010). Virtualized and flexible ECC for main memory. In *Proc. of ASPLOS*, pages 397–408.

[65] Yoon, D. H. and Erez, M. (2011). Virtualized ECC: Flexible reliability in main memory. *IEEE Micro*, 31(1):11–19.

[66] Zhang, Y., Yang, J., and Gupta, R. (2000). Frequent value locality and value-centric data cache design. *SIGARCH Comput. Archit. News*, 28(5):150–159.

[67] Zheng, H., Lin, J., Zhang, Z., Gorbatov, E., David, H., and Zhu, Z. (2008). Mini-rank: adaptive DRAM architecture for improving memory power efficiency. In *Proc. of MICRO*, pages 210–221.

[68] Zhou, P., Zhao, B., Du, Y., Xu, Y., Zhang, Y., Yang, J., and Zhao, L. (2009). Frequent value compression in packet-based noc architectures. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 13–18.

[69] Zimmermann, R. (1999). Efficient vlsi implementation of modulo (2 n±1) addition and multiplication. In *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*, pages 158–167.