2014

Energy-efficient and cost-effective reliability design in memory systems

Long Chen
Iowa State University

Energy-efficient and cost-effective reliability design in memory systems

by

Long Chen

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

Ying Cai

Morris Chang

Arun Somani

Joseph Zambreno

Iowa State University

Ames, Iowa

2014

DEDICATION

Dedicate to my parents, my parents-in-law and my wife. I have so so many thanks that I would like say to you....

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 consistent encouragement and patient guidance through my five years Ph.D study. He leads me to the exciting research area, gives me freedom in seeking research topics, guides me to resolve the research challenges, and teaches me how to present the idea in words and to the audiences. Dr. Zhang is not only a great adviser in academy, he is also a mentor and a friend in my personal life. I remember at early stage that I was depressed by my frustrating research. It is his full belief in me and constant encouragement that helped me out of the difficult time. He invited me to play table tennis to release my pressure and encouraged to not hesitate to develop research ideas. He always encourages me to aim high to compete with students in top level universities. Without his full trust and encouragement, I could not have reached this far. Thank you!

I would like to thank the rest of my thesis committee - Professors Ying Cai, Morris Chang, Arun Somani and Joseph Zambreno for their feedback and suggestions to improve this dissertation.

I would also like to thank Dr. Zhichun Zhu from University of Illinois at Chicago for her insightful suggestions on my research. I would like to thank Kun Fang from University of Illinois at Chicago for discussing ideas with me, guiding me to integrate the simulation tools and clarifying my misunderstanding of memory systems. I thank Yanan Cao for helping me out of challenging coding bugs. He is always the right person for resolving a coding issue. I would thank Jim Robertson, Manas Mandal, Gilberto Contreras, Major Bhadauria from NVIDIA for the guidance and for the great time I have during my internship.

I would additionally like to thank all my friends. Bo Sun teaches me how to drive, helps me

a lot with my five-kilometer running goal and corrects my gestures for swimming, bowling and pooling. Yuqing Chen, Meng Li, Yinan Fang, Shuren Feng, Xinying Wang, Zhiming Zhang, Liping Wu, Cheng Gong, Gengyuan Zhang, Haiding Sun, Junfeng Wang, Shu Yang, Hui Lin, Ziyue Liu, Wei Zhou, Yixing Peng are the names flashed across my mind when I pick the word friendship. Without you, I cannot have so much fun in Ames.

ABSTRACT

Reliability of memory systems is increasingly a concern as memory density increases, the cell dimension shrinks and new memory technologies move close to commercial use. Meanwhile, memory power efficiency has become another first-order consideration in memory system design. Conventional reliability scheme uses ECC (Error Correcting Code) and EDC (Error Detecting Code) to support error correction and detection in memory systems, putting a rigid constraint on memory organizations and incurring a significant overhead regarding the power efficiency and area cost.

This dissertation studies energy-efficient and cost-effective reliability design on both cache and main memory systems. It first explores the generic approach called embedded ECC in main memory systems to provide a low-cost and efficient reliability design. A scheme called E^3CC (Enhanced Embedded ECC) is proposed for sub-ranked low-power memories to alleviate the concern on reliability. In the design, it proposes a novel BCRM (Biased Chinese Remainder Mapping) to resolve the address mapping issue in page-interleaving scheme. The proposed BCRM scheme provides an opportunity for building flexible reliability system, which favors the consumer-level computers to save power consumption.

Within the proposed E^3CC scheme, we further explore address mapping schemes at DRAM device level to provide SEP (Selective Error Protection). A general SEP scheme has been proposed by others to selectively protect memory regions taking into account of both reliability and energy efficiency. However, there lacks detailed DRAM address mapping schemes which are critical to the SEP scheme. We thus explore a group of address mapping schemes at DRAM device level to map memory requests to their designated regions. All the proposed address mapping schemes are based on modulo operation. They will be proven, in this thesis, to be efficient, flexible and promising to various scenarios to favor system requirements.

Additionally, we propose *Free ECC* reliability design for compressed cache schemes. It

utilizes the unused fragments in compressed cache to store ECC. Such a design not only reduces the chip overhead but also improves cache utilization and power efficiency. In the design, we propose an efficient convergent cache allocation scheme to organize the compressed data blocks more effectively than existing schemes. This new design makes compressed cache an increasingly viable choice for processors with requirements of high reliability.

Furthermore, we propose a novel, system-level scheme of memory error detection based on memory integrity check, called MemGuard, to detect memory errors. It uses memory log hashes to ensure, by strong probability, that memory read log and write log match with each other. It is much stronger than SECDED (Single-bit Error Correcting and Double-bit Error Detecting) in error detection and incurs little hardware cost, no storage overhead and little power overhead. It puts no constraints on memory organization and no major complication to processor design and operating system design. In consumer-level computers without SECDED protection, it can be coupled with memory checkpointing to substitute ECC, without the storage and power overhead associated with ECC. In server computers or other computers requiring strong reliability, it may complement SECDED or chipkill-correct scheme by providing even stronger error detection. In the thesis, we prove that the MemGuard reliability design is simple, robust and efficient.

CHAPTER 1. INTRODUCTION

Memory system plays pivotal role in computer systems with either Von Neumann or Harvard architecture. In these architectures, data and programs are stored in memory system and loaded into processors for program execution. The reliability of memory system is therefore critically important to correctness of the execution flow. However, physical threats to memory storage have been observed decades ago and memory system reliability is increasingly a concern as the fabrication technology scales down to sub-nanometer regime. Without error protection, a single upset of memory cell can cause memory and data corruptions, application and system crashes, system security vulnerabilities [108, 27], and others. The more data memory holds and the longer it maintains, the higher chance that programs may experience upsets and mishaps. It has been reported from real machine failure statistics that main memory is responsible for about 40% of system crashes caused by hardware failure [65].

Another major concern of memory system is performance and power consumption in this era of multi/many-core per system and multi-thread per core. For years, memory technologies have been pushing forward to chase the performance of processors. Cache memory is further optimized to reduce its leakage power and on-chip cache capacity is growing to improve system performance. As for main memory, the mainstream DDRx (Double Data Rate) DRAM (Dynamic Random-Access Memory) frequency has evolved from 800MHz to 933MHz and 1066MHz to provide a higher bandwidth. Although supplying voltage of each memory generation decreases, the power consumption of memory system grows to a serious concern in computer system design. A typical 2GB to 4GB DDR3 memory can consume 5 to 13 Watt power varied for different configurations with different workloads [36]. As memory capacity in a server system is up to 32GB, 64GB and even higher, the power consumption of memory system is substantially large. Studies on real machines have revealed that DRAM memory

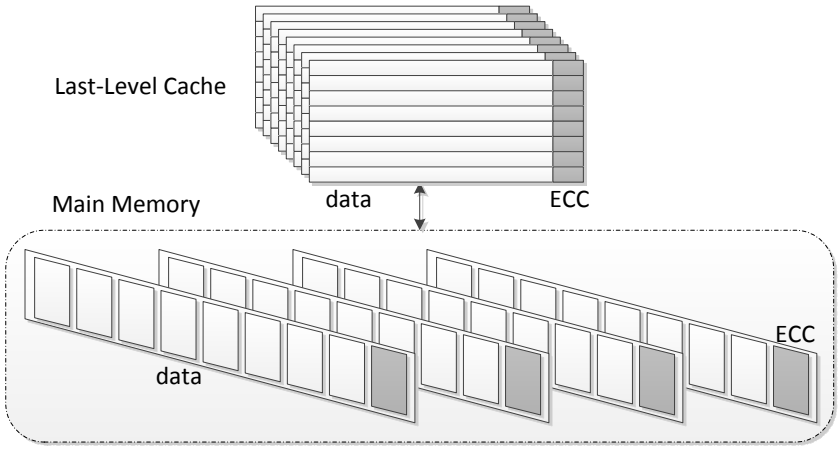


Figure 1.1: Conventional cache and main memory organization with ECC supported. The dark gray storage is used for ECC.

system can consume more power than processors for memory intensive workloads; and it has been predicted that future memory systems may consume more than 70% of system power [9].

However, conventional memory error protection scheme introduces significant overhead in terms of area cost and power consumption. Traditionally, (72, 64) Hamming based [28] or Hsiao code [31] is applied to provide Single-bit Error Correcting and Double-bit Error Detecting. Such a code is thus also called SECDED, following its error protection capability. Figure 1.1 shows the memory organization with SECDED ECC (Error Correcting Code) supported. In both cache and main memory system, extra 1/8 storage is appended to original data arrays to maintain ECC, which is shown in dark gray fields in the figure. As the applied SECDED code strictly presents 8:1 ratio for data and ECC code word for a typical 64-byte data block, it limits the rigid organization of memory system. For example, an eight (x8) DIMM (Dual-Inline Memory Module) requires one extra DRAM device for ECC and a sixteen (x4) DIMM requires two more devices. However, it is challenging for a memory module or mobile memory system with one, two or four devices to preserve the 8:1 ratio. Such a rigid organization of conventional ECC design limits the adoption of new memory techniques, presenting challenges on designing a reliable and power-efficient system.

This dissertation studies energy-efficient and cost-effective reliability design on both cache and main memory systems. It first proposes E^3CC (Enhanced Embedded ECC) for power effi-

cient sub-ranked and other narrow-ranked memory organizations. By embedding ECC together with data storage, E^{3}CC design decouples the connection between number of DRAM devices and error protection code. E^{3}CC design presents the opportunity of flexibility in reliability design and makes sub-ranked memory organization a viable choice for systems with requirements of power efficiency and high reliability. Secondly, it further explores E^{3}CC scheme to implement SEP (Selective Error Protection). SEP framework design is proposed by others [68] for energy efficient reliability design consideration as it protects solely the critical data. However, there lacks DRAM device level address mapping schemes to support SEP. We thus explore a group of address mapping schemes based on efficient modulo operation. The proposed mapping schemes are flexible, efficient and promising to various scenarios to favor the system requirements. Thirdly, it proposes *Free ECC* design for compressed LLC (Last Level Cache), relying on the observation that substantial idle fragments are left unused in compressed caches. *Free ECC* design maintains ECC in those fragments to save the dedicated storage required in conventional ECC design to further improve cache capacity utilization and power efficiency. Additionally, it proposes MemGuard, a system-level scheme with lightweight hardware extension to support or enhance memory reliability for a wide spectrum of computer systems including consumer computers with or without ECC and large-scale, high-performance computing applications. The design is simple, efficient and strong in error detection capability.

In detail, Chapter 3 develops E^{3}CC. It explores the generic approach of embedded ECC on power-efficient sub-ranked and other narrow-ranked memory systems. It embeds ECC together with data to avoid using the extra DRAM devices to maintain ECC, which are required in conventional ECC DIMM. In the design, it proposes a novel BCRM (Biased Chinese Remainder Mapping) to resolve the DRAM device-level address mapping challenge since effective memory capacity is reduced to a non-power-of-two size as ECC is embedded. It also identifies the issue of extra ECC traffic in DDR3 memories embedded ECC may cause, as a result of the burst length requirement of DDR3. A simple and effective cache scheme called ECC-cache is proposed to effectively reduce this traffic overhead. The E^{3}CC design enables ECC on conventional non-ECC DIMMs, which presents the opportunity of flexibility in reliability design. The design is demonstrated to be efficient and reliable.

Chapter 4 explores DRAM device-level address mappings for SEP (Selective Error Protection). SEP requires to partition memory space into a protected region and an unprotected region. In this case, a unique and effective address mapping scheme is needed to avoid the use of complex Euclidean division. It therefore explores the CRM (Chinese Remainder Mapping) scheme, and by grouping multiple-columns/rows as a *super-column/row*, it proves that CRM is effective for most cases to maintain either access parallelism or row buffer locality. For a corner case that CRM-based mapping is challenging to present row buffer locality, it proposes a *section-based* address mapping scheme, in which the address space is divided into sections based on greatest common divisor. Additionally, it proposes *adjustment-factor*s for the proposed mapping schemes to further tune the mapping layout in order to obtain required properties. The proposed schemes are all modulo based and they are flexible, efficient and promising to various scenarios that require memory partitioning.

Chapter 5 proposes *Free ECC* design to embed ECC into substantial idle fragments left, otherwise unused, in compressed LLC (Last Level Cache). It thus saves the dedicated storage for ECC in conventional reliable cache design and the power consumption. In the design, it first proposes a convergent allocation scheme to organize compressed data blocks in cache efficiently, targeting high compression ratio and low design complexity. Based on the proposed layout, it carefully examines the technical issues and design challenges in embedding ECC. Three cases are distinguished and discussed in detail and *Free ECC* design is demonstrated to be simple and efficient.

Chapter 6 presents MemGuard, a system-level error protection scheme to provide or enhance memory reliability for a wide spectrum of computer systems. The core part of MemGuard is a hash-checking based low-cost and highly-effective mechanism of memory error detection. In detail, a read log hash (ReadHash) and a write log hash(WriteHash) are maintained, 128-bit each for data correction check. The two hashes conceptually are hashed values of the log of all read and write accesses from or to main memory. By synchronizing to the same point of the two hashes periodically or at the end of program execution, the two match each other. Otherwise, errors are detected. The proposed MemGuard design can detect multi-bit errors of main memory, in very strong confidence. This reliability design is generic, which can

be applied to both consumer level devices with or without error protection and large-scale, high-performance computing applications to enhance their reliability.

The overall organization of the rest of this dissertation is as follows. Chapter 2 introduces background of main memory organization, memory error causes and consequences, and conventional memory error protection schemes. Chapter 3 develops the E^3CC design to support error protection for sub-ranked and other narrow-ranked main memories. Chapter 4 explores a group of mapping schemes at DRAM device level to support selective error protection for energy efficient concern. Chapter 5 proposes the *Free ECC* organization to reduce the cost of error protection for compressed cache schemes. Chapter 6 presents a system-level memory error protection with lightweight hardware extension, to support or enhance memory reliability for a wide spectrum of computer systems. Chapter 7 concludes this dissertation and outlines future research directions.

CHAPTER 2. BACKGROUND

This chapter introduces the background of main memory organizations, basic operation commands and its power model, which is applied in this thesis for memory power evaluations. The detailed causes and consequences of memory errors are presented to identify the issue and conventional memory error protection schemes are introduced. Their rigid organizations and significant overheads motivate our study.

2.1 Main Memory Organization

Conventional main memory in desktop computers and servers may consist of multiple channels and each channel connects to one to four DIMMs (Dual Inline Memory Module). A DIMM is a PCB (Print Circuit Board) that contains one or two ranks, which is formed by eight (x8 or sixteen x4) DRAM devices. A rank of DRAM devices are operated in tandem to form a 64-bit data path, which is connected to memory controller through a 64-bit data bus. A memory request is served by eight bursts on the data bus to pump a 64-byte data block, which is the same size to the last-level cache data block. In detail, each DRAM device typically consists of eight DRAM banks and the requested data block is fetched from DRAM bank to row buffer and then to memory controller for a read request. For a memory write request, the data flow direction is reversed. Figure 2.1 presents an example modern DDRx DRAM memory organization, where memory controller issues memory commands and memory request addresses to DRAM devices and data is transferred in between the two through the data bus.

There are up to three commands in DDRx memory system for serving a request, namely row activation, column access and precharge [87, 13]. The activation command is issued to the entire rank of banks to drive data from DRAM device cells to a bank sense amplifier, also called

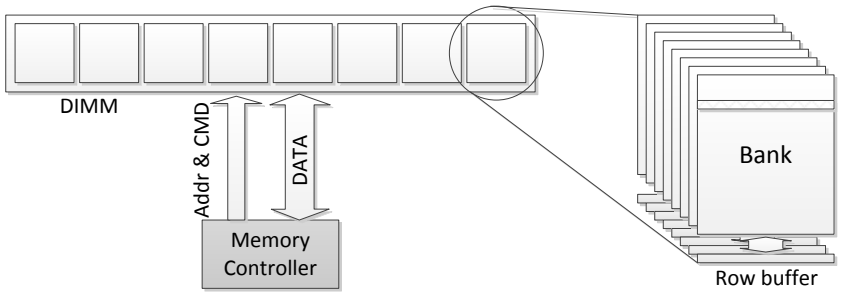


Figure 2.1: A conventional DDRx memory organization without error protection.

row buffer as one of its main functions is to temporarily retain data. A DRAM row is also called opened once it is activated. With a row opened, the following column access command can fetch a 64-byte data block directly from the row buffer. Typically, the row buffer is large, 8KB for example, so that continuous column access commands can be issued targeting the opened row. Such a case is called row buffer hit as it saves an extra row activation command to open the row for the column access. When a memory request is completed, a precharge command can be issued to restore data from row buffer back to DRAM device cells and the bank is said to be closed. As all rows in one bank share the same row buffer, solely one row can be opened in each bank. With the bank precharged, a DRAM access cycle is said to be completed and the bank is ready for another activation command.

2.2 DDRx DRAM Power Model

DDRx DRAM power is classified into four categories: background, operation, read/write, and I/O [70]. Operation power is consumed when bank precharge and activation commands are executed. Read/write power is for memory read and write while I/O power is to drive data bus for a rank and to terminate data on other ranks in the channel. DDRx DRAM supports multiple power levels to reduce background power as it is consumed consistently with or without operations. Figure 2.2 presents the three power levels and six power modes that a DRAM device can stay. The three levels are standby, powerdown and self-refresh. A DRAM rank is said to be in precharge standby if all the banks in the rank are closed; otherwise, it is active standby. The two standby modes consume high background power but the rank is ready for accesses.

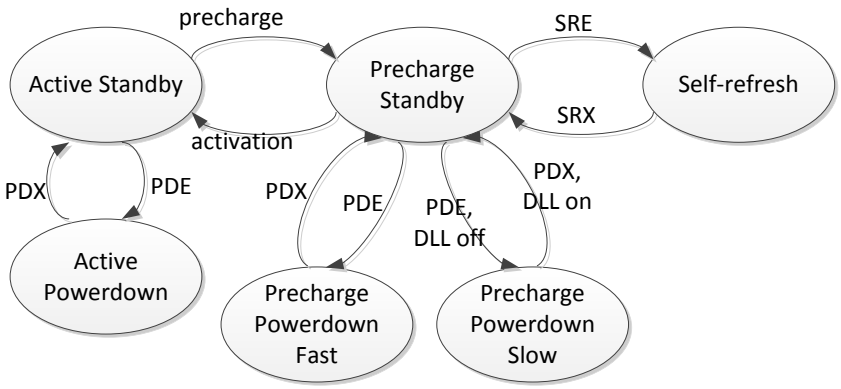


Figure 2.2: DDRx memory background power states tensions. PDE: PowerDown Enter; PDX: PowerDown eXit; SRE: Self-Refresh Enter; SRX: Self-Refresh eXit.

When the clock is disabled, the two standby modes transit to their corresponding powerdown modes, namely precharge powerdown fast and active powerdown. These two powerdown modes consume less power than standby modes with penalty of longer access latency to wake up the rank. There is one extra mode for precharge powerdown, namely, precharge powerdown slow. It is entered with DLL (Delay Lock Loop) frozen and it consumes less power than precharge powerdown fast but takes longer time to be ready for memory requests. The last power level is called self-refresh that consumes the least power as it just performs the periodic refresh operations to retain the data. It takes the longest time to wake up for memory accesses.

2.3 Memory Error Causes and Consequences

2.3.1 Causes of Memory Errors

Memory soft error was first reported by May and Woods [67] in 1979. Since then considerable efforts have been spent on memory error studies [45, 72, 10, 6, 113, 114, 103, 73]. Memory errors are mainly caused by unexpected charge intrusions due to particle injections. For years, multiple mechanisms have been observed to induce memory errors [6, 39, 40]. Alpha particles emitted by impurities in packaging materials are proved to be the dominant cause of soft errors decades ago. Then high-energy and low-energy neutrons from cosmic rays are observed to tamper memory data through two different mechanisms. As memory technology scales down to nanometer regime, complex physical effects are increasingly challenging to memory reliabil-

ity [39, 40]. In detail, ITRS 2011 presents the following effects. First, the thinner MOS gate dielectric sustains less time to breakdown. Second, the scaling effect raises the issues of process variations and random charge fluctuations, which exacerbate memory cell reliability. There are also many other effects, such as p-channel negative bias temperature instability (NBTI), the random telegraph noise (RTN) and others [39] that are increasingly severe.

2.3.2 Memory Error Rate and Consequences

Recent studies on large field real machines have found that main memory error rate is orders of magnitude higher than previously reported. One of the most recent studies on Google's fleet of servers reports about 25,000∼70,000 FIT per Mbit of DRAM systems [92]; FIT represents failures in a billion operation hours. The ratio of correctable error is 8.2% per year among ECC DIMMs, i.e. 8.2 of every 100 DIMMs detect and correct at least one error. The probability of uncorrectable error is 0.28% per year among ECC DIMMs, and 0.08% per year with Chipkill correct. Another study based on IBM Blue Gene (BG) supercomputers at multiple national laboratories also reports high error rate [33]; for example, FIT of 97,614 on BG/L computers at the Lawrence Livemore National Laboratory and 167,066 on BG/P computers at the Argonne National Laboratory. In addition, it observes error correlation on the same row or column, indicating high rate for multi-bit errors. Some other studies [59, 10] also show the increase of memory error rate. ITRS 2011 [39] proposes the requirement of 1,000 FIT for future DRAM error ratio, which is challenging to meet.

Cache memory reliability has also been a serious concern for decades. It is increasingly severe for several reasons as manufacture technology scales down. First, data stored in shrunk cells becomes more vulnerable to energetic particles, such as alpha particles and neutrons. Even if single cell memory error rate keeps constant, number of errors in entire cache grows with the increase of cache capacity [6]. The cache memory error rate is further exacerbated as operating voltage decreases and processor operating frequency increases. Meanwhile, the aggressive optimization for reducing leakage power worsens cache error rate [14]. On-chip cache usually occupies a large portion of chip area, which presents a high probability of upset. In addition, the errors in cache can propagate easily, corrupting applications and even crashing

the system as cache is close to the processing units and register files [118].

Memory errors may have unpredictable and sometimes serious consequences to computer users. It may lead to user data errors, program or system crashes, and in the worst case security vulnerabilities [108, 27]. It has been reported that, among all computer components, memory error is the top cause of system crashes [91]. As for security, a memory error may cause illegal and unpredictable control flow. A study [27] on Java program execution shows that a single-bit error has a chance as high as 70% to induce the execution of arbitrary code.

2.4 Memory Error Protection

A large amount of studies have focused on main memory error protections [45, 72, 10, 6, 113, 114, 103, 73]. The most widely adopt scheme is to use ECC (Error Correcting Code), such as SECDED [83, 24], i.e. Single-bit Error Correcting and Double-bit Error Detecting. Specifically, it employs a (72, 64) Hamming [28] based or Hsiao Code [31], which encodes a 64-bit data into a 72-bit ECC word with 8 parity bits. A single-bit error within the 72-bit ECC word is correctable and a double-bit error is detectable but not correctable. Another simple code is EDC (Error Detecting Code), which requires merely one even/odd parity bit for a 64-bit data block. It can be used to detect all odd-bit errors.

In DDRx memory systems, a non-ECC DDRx DIMM usually has one or two ranks of DRAM devices, with eight x8 or sixteen x4 devices (chips) in each rank to form a 64-bit data path. A conventional ECC DIMM, with SECDED, has nine x8 or eighteen x4 devices per rank to form a 72-bit data path, with the extra one or two devices to store the parity bits. The DDRx data buses are 64-bit and 72-bit, respectively, without and with ECC. Therefore ECC DIMMs are not compatible with a motherboard designed for non-ECC DIMMs. In addition, the 8:1 ratio of SECDED code presents a rigid organization of DIMM structures. With DDRx memory modules of four devices in a rank, the ECC overhead is higher to maintain the 8:1 ratio. Cache memories also adopt SECDED protection in general. For a 64-byte cache block, it attaches 8 bytes ECC codeword, which incurs 12.5% storage and power overhead. As cache organization is more flexible than DDRx memory system, there are many other types of codes for higher error protection, such as DECTED (Double-bit Error Correcting and Triple-bit Error Detection),

SNCDND (Single-Nibble error Correction and Double-Nibble error Detection) [11] and Reed Solomon [84] codes. However, they are rarely used due to high overheads. For example, the overheads are 23% and 22% for DECTED and SNCDND, respectively, given a 64-byte data word.

Chipkill Correct [64, 35, 15], a stronger but more costly design, has the capability of Single Device Data Correction (SDDC) in addition to SECDED in DRAM systems. Previous Chipkill design uses eight DRAM devices to tolerate single-device error in 64 devices following SECDED coding structure. It introduces significant power overhead as 72 devices are grouped to work in lockstep. A recent design groups multiple bits from one memory device as a symbol and applies symbol-correction code to recover a device failure [113, 114]. It typically organizes 36 x4 DRAM devices in a particular way to form a 144-bit data path, transferring both data and symbol-correction code. Such a scheme involves 36 DRAM devices in one memory access, still with significant memory power consumption. The most recent Chipkill design [103] proposes multi-tier error detection and correction schemes to form a Chipkill level protection. It uses nine x8 DRAM devices to carefully organize data, EDC and ECC bits at each tier, but with an increased ratio of storage overhead. The downside of chipkill, however, is excessive memory power consumption because in most Chipkill schemes each memory access may involve many more memory devices than conventional ECC memories.

All these reliability designs on DRAM systems limit error protection in DRAM itself and the designs are tightly coupled with DRAM device types and organizations. The schemes are inefficient in terms of storage, cost and power consumption.

CHAPTER 3. E^{3}CC: RELIABILITY SCHEME FOR NARROW-RANKED LOW-POWER MEMORIES

This chapter presents and evaluates E^{3}CC (Enhanced Embedded ECC), a full design and implementation of a generic embedded ECC scheme that enables power-efficient error protection for subranked memory systems. It incorporates a novel address mapping scheme called BCRM (Biased Chinese Remainder Mapping) to resolve the address mapping issue for memories of page interleaving, plus a simple and effective cache design to reduce extra ECC traffic. Our evaluation using SPEC CPU2006 benchmarks confirms the performance and power efficiency of the E^{3}CC scheme for subranked memories as well as conventional memories.

3.1 Introduction

Memory reliability is increasingly a concern with the rapid improvement of memory density and capacity, as memory holds more and more data. Without error protection, a single upset of memory cell may lead to memory corruption, which may have further consequences including permanent data corruption, program and system crashes, security vulnerabilities, and others. However, the majority of consumer-level computers and devices have not yet adopted any memory error protection scheme. The more data in unprotected memory and the longer they stay there, the higher chance that users may experience those mishaps.

Meanwhile, memory power efficiency has become a first-order consideration in computer design. DRAM memory systems can consume more power than processors [60] on memory intensive workloads; and it has been predicted that future systems may spend more than 70% of power in memory [9]. Recently proposed sub-ranked DDRx memories [2, 1, 121, 105] reduce memory power consumption significantly by using memory sub-ranks of less than 64-bit data

bus and less number of devices than conventional DDRx memories; for example, with two x8 devices in a sub-rank of 16-bit bus width. Additionally, mobile devices such as iOS, Android, and Windows phones and tablets have started to use low-power DDRx (LPDDR, LPDDR2, LPDDR3 and the incoming LPDDR4) memory with 32-bit data bus, which is similar to 32-bit sub-ranked memory. Those mobile devices have been increasingly used in applications that require reliable computing to an extent; for example, medical care, mobile banking, construction, and others.

The two trends lead to a conflict between memory reliability and power efficiency. Conventional ECC memory employs a memory error protection scheme using a (72, 64) SECDED code, of which an ECC word consists of 64 data bits and eight ECC bits. A memory rank in ECC DDRx memory may consist of eight x8 memory devices (chips) dedicated for data and one x8 device dedicated for ECC, or sixteen x4 devices dedicated for data and two x4 devices dedicated for ECC. Sub-ranked and those narrow-ranked low-power memories are incompatible with this memory rank organization, leaving the question how to implement error protection in them[1]. The study of Mini-Rank [121], one type of sub-ranked memory organization, proposes a generic approach called *Embedded ECC* to alleviate the concern. In Embedded ECC, data bits and ECC bits of an ECC word are mixed together, and therefore dedicated ECC devices are no longer needed. It essentially decouples memory organization and the choice of error protection code. With the mixing, however, the *effective size* of DRAM row is no longer power-of-two, which complicates memory device-level address mapping. If an efficient address mapping is non-existent, Embedded ECC will not be a practical solution. Embedded ECC does identify CRM (Chinese Reminder Mapping) [25] to be a potential solution to the address mapping issue, but there lacks design and implementation details.

This chapter presents E^{3}CC (Enhanced Embedded ECC), which is a full design and implementation of Embedded ECC, but with its own innovations and contributions. E^{3}CC shares the following merits with the original Embedded ECC (most of which were not identified in the previous study [121]):

- E^{3}CC can be integrated into sub-ranked memory, yielding a power-efficient and reliable

[1] If a shorter ECC word is used, ECC storage overhead will increase significantly.

memory system. It can also be applied to those mobile devices using low-power DDRx memory.

- Although it is proposed for sub-ranked memories, it may also be used to provide ECC protection to non-ECC DDRx DIMMs by an extension to memory controller, with no change to the DIMMs or devices. Such a system can be booted in either ECC memory mode or non-ECC memory mode.

- It is now possible to use other error protection codes with more extension in memory controller; for example, a long BCH(532, 512) code that corrects 2-bit error and detects 3-bit error using 532-bit word size and with 3.9% overhead. By comparison, the conventional ECC memory corrects 1-bit error and detects 2-bit error using 72-bit word size and with 12.5% overhead.

This study on E^{3}CC has its own contributions and innovations beyond the original idea of Embedded ECC:

- The E^{3}CC is a thorough design and a complete solution. Equally important, its performance and power efficiency have been thoroughly evaluated by detailed simulation of realistic memory settings. Both are critical to prove that E^{3}CC will be a working idea when applied. By comparison, the original Embedded ECC was a generic idea, with little design detail and no evaluation at all. It was a supplemental idea to the Mini-Rank design.

- A novel and unique address mapping scheme called *Biased Chinese Reminder Mapping* (BCRM) has been proposed to resolve the address mapping issue for page-interleaving address mapping scheme. BCRM is an innovation by itself. It can be used in memory systems where the address mapping is not power-of-two based and spatial locality is desired.

- The study of Mini-Rank [121] identified CRM [25] as a potential method of efficient address mapping, which motivated the search for BCRM. *However*, this study has found

out that cacheline-interleaving in Embedded ECC does not really need CRM; and for page-interleaving, CRM may destroy or degrade the row-level locality of page-interleaving.

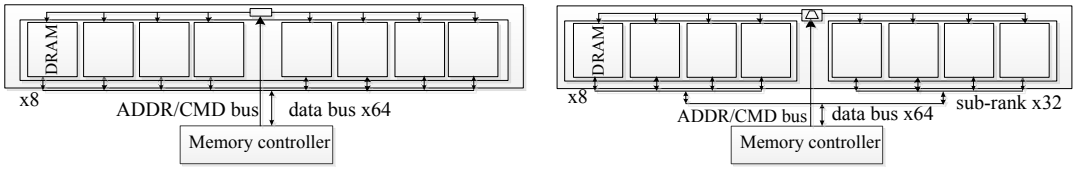
- This study identifies the issue of extra ECC traffic in DDR3 memories that Embedded ECC may cause, as a result of the burst length requirement of DDR3. A simple and effective scheme called *ECC-cache* is proposed to effectively reduce this traffic overhead.

E^3CC and the original Embedded ECC idea are related to but different from recent studies on memory ECC design. In particular, Virtualized ECC [113, 114] also makes ECC flexible as E^3CC does, but by storing ECC bits into separate memory pages. It relies on a processor's cache to buffer unused ECC bits to avoid extra ECC traffic. E^3CC has more predictable worst-case performance than Virtualized ECC, because it puts data bits and the associated ECC bits in the same DRAM rows. Accessing uncached ECC bits is merely an extra column access and additional memory bursts following those for data, with only a few nanoseconds added to memory latency, and no extra power spent on DRAM precharge and row activation. LOT-ECC [103] provides chipkill-level reliability using nine-device rank (assuming x8 devices). LOT-ECC also stores ECC bits with data bits in the same DRAM rows; however, it is otherwise very different from E^3CC in design. LOT-ECC provides stronger reliability than E^3CC but at the cost of more storage overhead (26.5% vs. 14.1%). The LOT-ECC design as presented is not compatible with sub-ranked memories, and thus LOT-ECC memory may not be as power-efficient as E^3CC memory with sub-ranking.

The rest of this chapter is organized as follows. Section 3.2 introduces background of sub-ranked memory organizations and related work. Section 3.3 presents novel address mapping and the design of Embedded ECC. Section 3.4 describes the performance and power simulation platform for DDRx memory. The performance and power simulation results are presented and analyzed in Section 3.5. Finally, Section 3.6 concludes the chapter.

3.2 Background and Related Work

Sub-ranked Memory Organization A memory access may involve up to three DRAM operations, namely precharge, activation (row access), and read/write (column access). The



(a) Conventional registered DIMM organization without ECC.

(b) Sub-ranked DIMM organization with x32 sub-rank size without ECC.

Figure 3.1: Comparison of the conventional registered non-ECC DIMM and a sub-ranked non-ECC DIMM organization. The sub-ranked DIMM consists of four devices per rank in this example.

memory controller issues commands to all devices in a rank to perform those operations. Sub-ranked memory organization [121, 2, 1, 105] is proposed to reduce the number of devices in a rank so as to reduce DRAM operation power spent on precharge and activation. It also increases the number of sub-ranks, which helps reduce DRAM background power. Figure 3.1 illustrates two DIMM organizations, a conventional registered DIMM and sub-ranked DIMM with 32-bit rank size. The difference is that the incoming commands and addresses are buffered and de-multiplexed to the devices in the destination sub-rank. This difference makes them sub-ranked, i.e. a full-size rank is now divided into sub-ranks of four, two or even one device. Figure 3.1b shows an example of sub-ranked DIMM with four devices per rank. For each memory access, the number of DRAM devices involved is effectively reduced, thus effectively reducing the row activation power and background power, which dominate memory power consumption.

Related Work There have been many studies on memory system reliability [113, 114, 90, 94, 103, 73, 115] and memory power efficiency [16, 32, 121, 104]. Most of those reliability-related studies focus on error correction for phase change memory and cache memory. Two studies closely related to our work are Virtualized ECC [113, 114] and LOT-ECC [103], which have been discussed in Section 3.1.

Recent studies show another trend of adopting sub-ranked memory architecture for power concern. The sub-ranked memory architecture divides conventional memory rank into multiple sub-ranks, each with a smaller number of DRAM devices involved. Mini-rank DIMM [121] breaks conventional DRAM rank into multiple smaller mini-ranks to reduce the number of

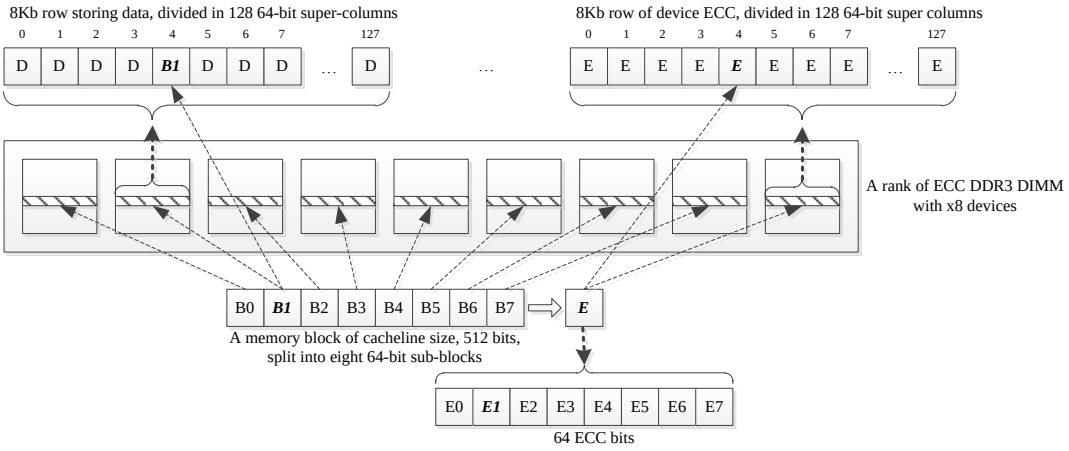


Figure 3.2: An example layout of a memory block of cacheline size (512-bit) inside a rank of a conventional ECC DDR3 DIMM using x8 devices. Each rank has eight data devices plus an ECC device. Each device has multiple banks, which is not shown. D represents a generic data super-column, and E represents a generic ECC super-column. One super-column is 64-bit, or eight columns in a device row. B_i represents a 64-bit sub-block of the memory block, and E_i represents the corresponding eight ECC bits. The memory block occupies eight 64-bit super-columns that are distributed over the eight devices; the ECC bits are stored in a column in the ECC device. An 8K-bit row consists of 128 super-columns. The row layouts of the second device and the ECC device are selectively highlighted, and so are the mappings of sub-block B1 and the ECC bits.

DRAM devices involved in one memory access, which thereby reduces the activation and row access power. The introduced bridge-chip works as a transfer to send 64-bit data to the 64-bit data bus. Multi-core DIMM [2, 1] is similar to sub-rank but uses split data bus with shared command and address bus to reduce the design complexity.

3.3 Design of E^{3}CC

In this section, we present the design of E^{3}CC and discuss in detail its design issues including memory block layout, page-interleaving and BCRM, ECC traffic overhead and ECC-cache, and the use of long error protecting code. All discussions are specific to DDR2/DDR3 DRAM memory; however, most discussions are also valid to other types of memory such as phase-change memories.

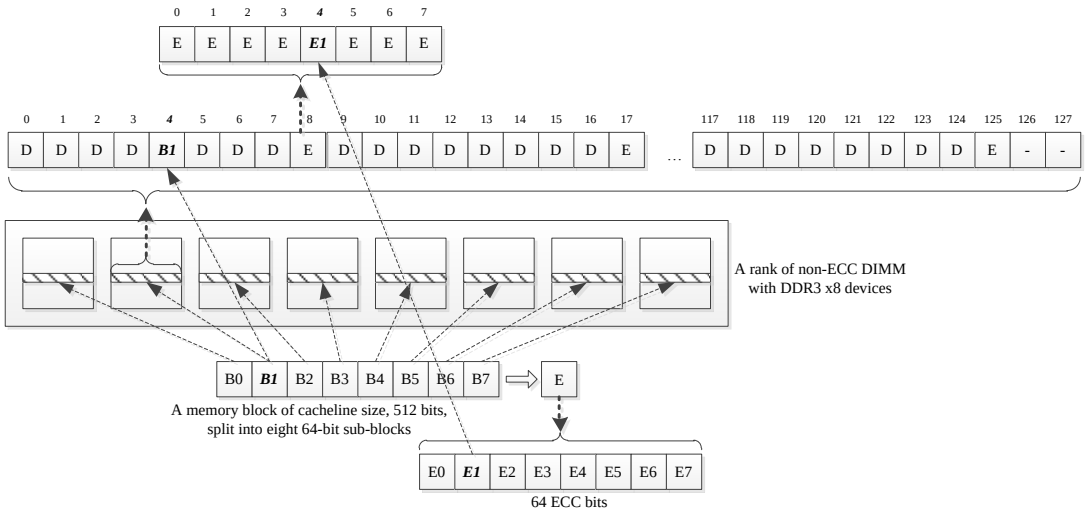


Figure 3.3: An example layout of a memory block of cacheline size (512-bit) inside a rank of E^{3}CC DDR3 DIMM of the full rank size. Other assumptions are the same as in Figure 3.2. The row layout of the second device is selectively highlighted, and so are the mappings of the second sub-block B1 and the ECC sub-block E1.

3.3.1 DIMM Organization and Intra-Block Layout

Figures 3.2 and 3.3 contrast the difference of conventional ECC memory and E^{3}CC memory. They show the mapping of memory sub-blocks inside a memory block of cacheline size; note that a memory block may be mapped to multiple memory devices. Most memory accesses are caused by cache miss, writeback, or prefetch. The examples assume Micron DDR3-1600 x8 device MT41J256M8 [69], which is 2G-bit each and has 8 internal banks, 32K rows per rank, 1K columns per row and eight bits per column. DDR3 requires a minimum burst length of eight, so each column access involves consecutive eight columns. We call an aligned group of eight columns a *super-column*. The ECC is a (72, 64) SECDED code made by a (71, 64) Hamming code plus a checksum bit, so each ECC word consists of 64 data bits and 8 ECC bits. In the conventional ECC DIMM, a memory block of the 64B cacheline size may occupy eight 64-bit super-columns distributed over the eight data devices. The ECC bits occupy a single super-column in the ECC device.

E^{3}CC DIMM is physically the same as conventional non-ECC DIMM. However, the ECC bits are mixed with data bits in the same DRAM rows. To maintain the 8:1 ratio of data and ECC, there is one ECC super-column for every eight data super-columns. The data bits of the

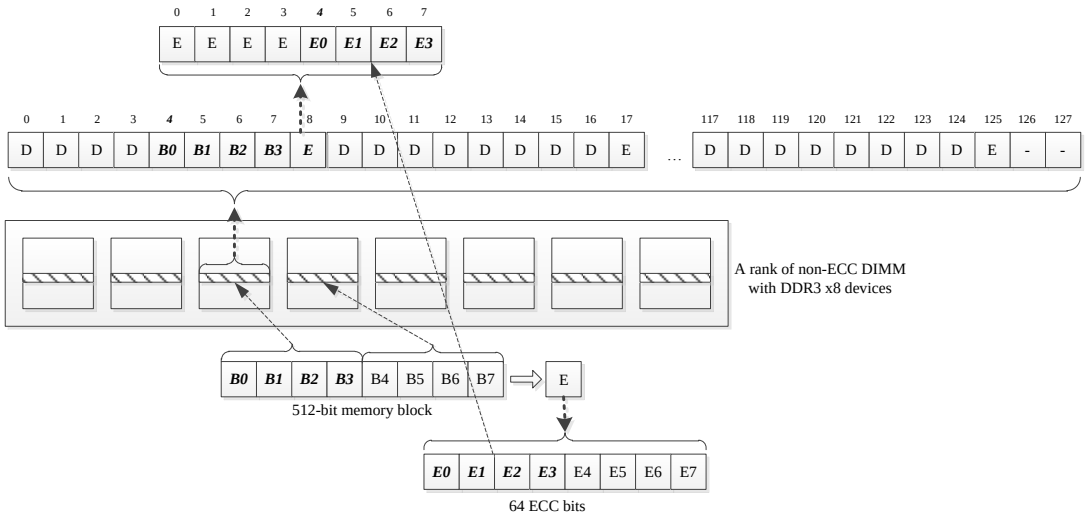


Figure 3.4: An example layout of a memory block of cacheline size (512-bit) for 16-bit sub-ranked E^{3}CC DDR3 DIMM. The other assumptions and notations are the same as in Figures 3.2 and 3.3. The row layout of the third device is selectively highlighted, and so are the mappings of sub-blocks B0-B3 and the ECC sub-blocks E0-E3.

memory block are distributed over the eight devices, one super-column per device. The ECC bits are also evenly distributed, occupying 1/8 super-column per device. An ECC super-column in the row is made by ECC bits from eight memory blocks. For this particular device, each row has 112 data super-columns and 14 ECC super-columns. The last two super-columns are not utilized, which are 1.6% storage overhead, in addition to the 12.5% ECC storage overhead.

The layouts can have different variants as long as the data and ECC bits of a memory block are evenly distributed over all devices in the rank and using the same row address. The intra-block layout of E^{3}CC DIMM is designed to be compatible with the burst length of eight of DDR3 devices[2]. A DDR3 device uses an internal buffer of eight bits per I/O pin to convert slow and parallel data into high-speed serial data; therefore, for a x8 device, each column access (data read/write) involves 64-bit memory data. Accessing a memory block of 64B requires only a single column access. However, accessing the associated ECC bits requires another column access (but no extra precharge or activation), of which only 1/8th of the bits are needed. We proposed ECC-cache to address this issue, which will be discussed in Section 3.3.4. DDR2 also

[2]DDR3 has a burst chop 4 mode, in which the burst length is reduced to four but with extra performance overhead introduced.

supports a burst length of four (in addition to eight), for which the layout can be different.

Figure 3.4 shows the intra-block layout for 16-bit sub-ranked E^{3}CC DIMM. The DIMM organization is the same as 16-bit sub-ranked, non-ECC DIMM. The memory block is now mapped to two x8 devices, using four data super-columns and 1/2 ECC super-column. Each memory access involves two devices, drastically reducing the power and energy spent on device precharge and activation. Accessing the ECC bits incurs less potential waste of bandwidth than before, as two memory blocks share one ECC super-column instead of eight in fully ranked DIMM. The last two super-columns are still leftover.

We do not show the layouts of 8-bit and 32-bit sub-ranked organizations as they are similar to that of the 16-bit, except that the memory block is mapped to one and four devices, respectively. The leftover ratio stays at 1.6% as the last two columns cannot be utilized. For 8-bit sub-ranked DIMM, the ECC bits of a memory block occupy a whole ECC super-column and therefore there is no potential waste of bandwidth when accessing the ECC bits.

3.3.2 Interleaving Schemes and Address Mapping

Memory interleaving decides how a memory block is mapped to memory channel, DIMM, rank and bank. It is part of the memory address mapping, i.e. to translate a given physical memory block address to the channel, DIMM, rank, bank, row and column indexes/addresses. In conventional ECC DIMM, the logic design is simply a matter of splitting memory address bits. With E^{3}CC, the number of memory blocks in the same row of the same rank and bank is no longer power-of-two, which complicates the memory device-level address mapping.

Two commonly used address mapping schemes are cacheline-interleaving and page-interleaving, and each has its variants. Figure 3.5 shows representative address mappings of the two types in DDRx memories. The main difference is that in page-interleaving, consecutive memory blocks (within page boundary) are mapped to the same DRAM rows, where in cacheline-interleaving the consecutive blocks are distributed evenly over channels, DIMMs, ranks and banks (not necessarily in that order). Page-interleaving is commonly used with open page policy, in which the row buffer of a bank is kept open after access; and cacheline-interleaving is commonly used with close-page policy. If another access falls into the same memory row, the data is still in the

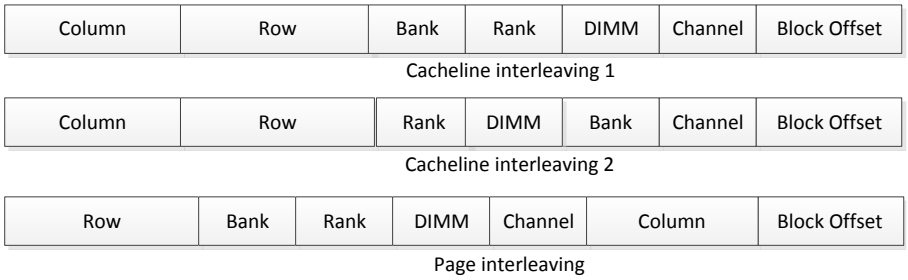


Figure 3.5: Representative memory address mappings for cacheline-interleaving and page-interleaving in DDRx memory systems. Cacheline interleaving 1 may be used in systems of memory power and thermal concerns, and cacheline interleaving 2 may be used to minimize the impact of rank-to-rank switching penalty on memory throughput.

row buffer of the bank and thus the read/write operation may start immediately. Otherwise, the bank must be precharged and then activated before the read/write operation, increasing memory access latency. Cacheline-interleaving with close-page policy is generally more power-efficient than page-interleaving with open-page policy, because it consumes more power to keep a row buffer open. However, the latter can be more power-efficient if there is a high level of page locality in memory accesses.

A major concern of E^{3}CC is regarding the address mapping scheme, because the effective size of DRAM rows, e.g. the number of data blocks/bits (excluding ECC bits) they hold, is no longer power-of-two. That means *the column address may not be extracted directly from the physical memory address bits*; and all address components to the left of the column address bits will be affected. We find that, however, for column-interleaving schemes whose column address is the leftmost bits in the physical memory address, the address mapping is not an issue. Those address components can be broken down as shown in Figure 3.5; the difference is that a subset of high-end column addresses becomes invalid, as they would represent invalid physical memory addresses on E^{3}CC memory. Page-interleaving, however, is affected because the column address appears in middle of the physical memory address. An integer division may have to be used, unless a unique and efficient address mapping scheme is found.

However, using integer division may negatively affect memory latency and throughput. It was not acceptable in 1980s and 1990s [54, 101, 25], and still very questionable in modern processors. A 32-bit integer division takes many clock cycles (56∼70 in Pentium 4 processor).

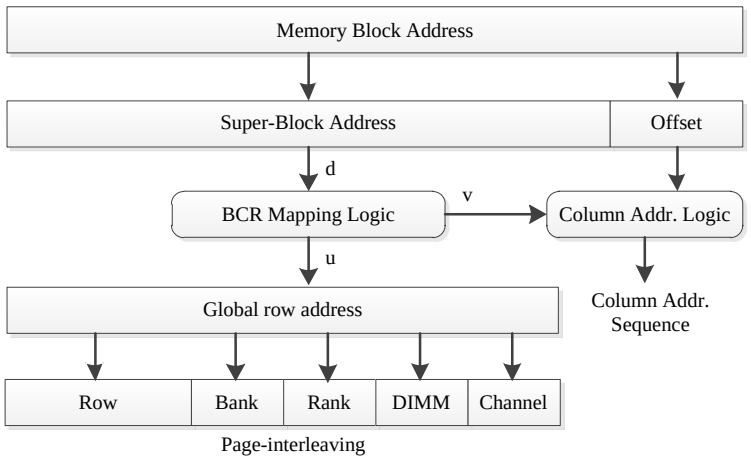


Figure 3.6: The logic flow of BCRM-based address mapping with page-interleaving.

It can be even more expensive than double-precision floating-point division (38 in Pentium 4 processor) [100]. A recent manual from Intel [37] reports that the latency of IDIV (Integer Division) instruction is from 20 to 70 cycles for Intel 64 and IA-32 architectures. We did simulation to evaluate the performance impact of adding 5ns, 10ns and 15ns for integer division delay to the memory controller overhead, using the baseline simulation setting in Section 3.4. The performance degradation is 3%, 5% and 7%, respectively. The simulation assumes that the division unit is fully pipelined. In reality, it is difficult to pipeline a division unit; that means multiple division units may have to be used, or this will be another factor limiting memory throughput. In our simulation setting, the peak memory throughput is one access per 2.5ns. Finally, for mobile processors or SoCs (System-On-Chip) of relatively low speed, the latency of integer division may be even longer than 15ns.

3.3.3 Page-Interleaving with BCRM

CRM and BCRM We have found a novel and efficient address mapping scheme called BCRM (Biased Chinese Reminder Mapping). We start with an existing scheme called CRM (Chinese Reminder Mapping), which was proposed based on Chinese Reminder Theorem [25], in which a memory block address d is decomposed into a pair of integers $\langle u, v \rangle$, with $0 \leq d \leq$

	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 3.1: An example of layout for the address mapping from d to $\langle u, v\rangle$ based on the CRM, with $m = 8$ and $p = 7$. For example, 11 is mapped to $\langle 3, 4\rangle$ as 11 mod 8 = 3 and 11 mod 7 = 4. A row represents a value of u and a column represents a value of v. The numbers in bold type highlight the mapping for the first 14 blocks. In CRM, each column is intended to represent a memory bank, and consecutive blocks should be mapped evenly to multiple banks. Row-level locality does not matter.

$pm - 1$, $0 \leq u \leq m - 1$ and $0 \leq v \leq p - 1$, using the following formula[3]:

$$u = d \bmod m, \qquad v = d \bmod p$$

CRM ensures that the mapping from d to $\langle u, v\rangle$ is "perfect" if p and m are coprimes, where being "perfect" means all possible combinations of (u, v) are used in the address mapping. Two integers are coprimes if their greatest common divisor is 1. The formal proof of the "perfect" property is given in the previous study [25]. Table 3.1 shows the layout of block address d under this mapping, assuming $p = 7$ and $m = 8$. As it shows, the numbers are laid out diagonally in the two-dimension table. In a prime memory system, p is intended to be the number of memory banks and m the number of memory blocks in a bank, and p is also intended to be a prime number. However, when used with page-interleaving, CRM will disturb the row-level locality as it was not designed for maintaining the locality.

BCRM adds a *bias factor* to the original CRM formula:

$$u = ((d - (d \bmod p)) \bmod m, \qquad v = d \bmod p$$

The bias factor $-(d \bmod p)$ adjusts the row position of each block; it "draws back" consecutive blocks to the same row. Table 3.2 shows an example of BCRM using the same parameters as

[3] We have reversed the notations of u and v from the equations in [25].

	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**

Table 3.2: An example of layout for BCRM, which maps d to $\langle u, v\rangle$ with $m = 8$ and $p = 7$. For example, 11 is mapped to $\langle 7, 4\rangle$ as $(11 - (11 \bmod 7)) \bmod 8 = 7$ and $11 \bmod 7 = 4$. A row represents a value of u and a column represents a value of v. The numbers in bold type highlight the mapping for the first 14 blocks. In BCRM, each row is intended to represent a memory row (all blocks in the same row resides in a single bank, and different rows may or may not be mapped to the same bank), so maintaining row-level locality or not makes a critical difference.

in Table 3.1. In our application scenario, u is the global row address and v is super-block index (to be discussed) in the row. The global row address is further decomposed as channel, DIMM, rank, bank, and row addresses as Figure 3.6 shows.

Proof of Correctness and Row Locality BCRM is a "perfect" address mapping, i.e. the mapping is mathematically a one-to-one function from $d \in [0, mp - 1]$ to $\langle u, v\rangle \in [\langle 0, 0\rangle, \langle m - 1, p - 1\rangle]$. This can be proven as follows. CRM has been proven to be a perfect address mapping [25], i.e. it can be formulated as a one-to-one mathematical function cr from $[0, mp - 1]$ to $[\langle 0, 0\rangle, \langle m - 1, p - 1\rangle]$. BCRM can be formulated as a mathematical function bcr defined as following:

$$bcr(d) = f(cr(d))$$

$$f(\langle u, v\rangle) = \langle (u - v) \bmod m, v\rangle$$

Function f is obviously a one-to-one function from $[\langle 0, 0\rangle, \langle m-1, p-1\rangle]$ to $[\langle 0, 0\rangle, \langle m-1, p-1\rangle]$. Therefore, bcr is a one-to-one function from $[0, mp - 1]$ to $[\langle 0, 0\rangle, \langle m - 1, p - 1\rangle]$.

BCRM has the property of "row locality": assume x is an integer that satisfies $(x \bmod p) = 0$, then $x, x + 1, \ldots, x + p - 1$ will be mapped to the same row, i.e. the value of u is the same

for those consecutive d values.

Using BCRM in Memory Address Mapping We let u be the global row address and m be the number of all DRAM rows in the whole system, which is a power of two. We make v be the index of memory *super-block*, which is so defined such that each row hosts exactly 7 memory super-blocks. Correspondingly, d is the global memory super-block address. The scheme is feasible because in E^{3}CC the number of utilized columns in a row is a multiple of 7, as 63 columns are utilized for every 64 columns. For example, for the example in Figure 3.3 a memory super-block is $112/7 = 16$ blocks, and for the example in Figure 3.4 a super-block is $28/7 = 4$ blocks. We assume that each row has 64 or a multiple of 64 super-columns, which is valid for today's memory device. As device capacity continues to increase, it will continue to hold. After v is generated, the memory controller generates a sequence of column addresses from v (plus the removed bits) for the corresponding data and ECC columns.

A merit of those non-power-of-two mappings using modulo operations is that fast and efficient logic design exists for modulo operation ($d \bmod p$), particularly if p meets certain property. In general case, assume d is an n-bit number of bits $d_{n-1}...d_1d_0$, then

$$d \bmod p = (\sum_{i=0}^{n-1} ((d_i \cdot 2^i) \bmod p)) \bmod p$$

The logic operations for $(d_i \cdot 2^i) \bmod p$ can be done in parallel and so can be the operations to calculate their sum. The logic design can be further optimized if p is a preset value, and even more if p is a certain specific value [101]. Finally, with m being a power of two, "$x \bmod m$" is simply to select the least-significant $log_2(m)$ bits of x.

3.3.4 Extra ECC Traffic and ECC-Cache

The E^{3}CC design may require an extra column access (read or write operation) to retrieve the ECC bits for a given memory request. For DDR3 DIMM of full rank size and 64B memory block access, it takes a single column access to transfer 64B data, and therefore in the worst case transferring the ECC bits may double the number of column accesses (effectively more memory bursts). Since only 72B of the 128B data are immediately needed, the potential bandwidth

overhead from the extra 56B transfer is 77.8% from the baseline. For 32-bit and 16-bit sub-ranked DIMMs, for each memory access it takes 3 and 5 column accesses for a device to fetch 32B and 16B data, respectively, and their ECC bits. The amount of extra bandwidth usage is 24B and 8B per request and the percentage of traffic increase is 33% and 11%, respectively, from the baseline in the worst case. For 8-bit sub-ranked DIMM, there is no bandwidth overhead. DDR2 also supports burst length of four, which can be used to reduce the overhead. Low-power DDR2 supports burst length of four and the rank size is 32-bit, so the percentage of worst-case bandwidth overhead starts from 11% (transferring 80B for 72B) and disappears at 16-bit sub-ranking, if burst of four is used.

To reduce this overhead, we propose to include a small ECC-cache in the memory controller to buffer and reuse the extra ECC bits. The cache is a conventional cache with a block size of 8B, and it utilizes the spatial locality in memory requests. We find that a 64-block, fully-associative cache is sufficient to capture the spatial locality for most workloads; set-associative cache can also be used.

It is worth noting that a partial write of an ECC super-column in DRAM does not require reading the ECC column first. DDRx supports the use of *data mask* (DM) for partial data write. The DM has eight bits. On a write operation, each bit may mask out eight data pins from writing, and thus the update can be done by 8B granularity. Furthermore, our design utilizes the burst-chop-4 feature of DDR3 on ECC writes to reduce the memory traffic overhead and power overhead from writing ECC bits.

3.3.5 Reliability and Extension

E^{3}CC can correct single random bit error and detect double random bits error happening in DRAM cells. The study on LOT-ECC [103] identifies other types of failures and errors, including row-failure, column-failure, row-column failure, pin failure, chip failure and multiple random bits error, which can be covered by LOT-ECC. The first four types are stuck-at errors, and can be covered by flipping the checksum bit in the error protection code. If the same idea is used in the error protection code of E^{3}CC, E^{3}CC can also detect a single occurrence of those failures, but may not be able to correct them. E^{3}CC cannot guarantee the immediate detection

of a pin failure, because it may cause multiple-bit errors in an ECC word on E^{3}CC. However, a pin failure also causes stuck-at errors, which with a high probability will be quickly detected by consecutive ECC checking. Similarly, a chip failure may also be quickly detected but cannot be recovered. Note that LOT-ECC is stronger than E^{3}CC at the cost of higher storage overhead (26.5% vs. 14.1%) and lower power efficiency. We argue that the E^{3}CC design is suitable for consumer-level computers and devices that require memory reliability enhancement with low impact on cost and power efficiency.

The flexibility of E^{3}CC enables the use of other error protection code for improved storage overhead, reliability, power efficiency, and performance. Conventional ECC memory using the Hamming-based (72, 64) SECDED code has a storage overhead of 12.5%. Hamming code is actually a special case of BCH codes with minimum distance of 3 [71]. For memory blocks of 64B cacheline size, one may use very long BCH code such as BCH(522, 512), BCH(532, 512), and BCH(542, 512) codes, which are SEC, DEC and TEC (Single-, Double- and Triple-bit Error Correcting), respectively, for 512-bit data block. The storage overhead ratio is 1.95%, 3.9% and 5.9%, respectively. To use BCH code in E^{3}CC, another set of address mapping logic may be needed.

3.4 Experimental Methodologies

We have built a detailed main memory simulator for the conventional DDRx and the sub-ranked memory system and integrated it into the Marssx86 [78] simulator. In the simulator, the memory controller issues device-level memory commands based on the status of memory channels, ranks and banks, and schedules requests by hit-first and read-first policy, using cacheline-interleaving and page-interleaving. We did experiments with the two cacheline-interleaving schemes in Figure 3.5; they are very close in the average performance. The first cacheline-interleaving scheme is used in the presented results. The basic XOR-mapping [119] scheme is used as the default configuration for page interleaving to reduce the bank conflict.

Table 3.3 shows the major parameters for the simulation platform. The simulator models Micron device MT41J256M8 [69] in full detail. To model the power consumption, we integrate a power simulator into the simulation platform using the Micron power calculation

methodology [70]. It also models the DDR3 low power modes, with a proactive strategy to put memory device in power-down modes with fast exit latency. We follow the method of a previous study [121] to break down memory power consumption into background, operation, read/write and I/O power. The details of memory power model are introduced in Section 2.2. The power parameters are listed in Table 3.4.

Parameter	**Value**
Processor	4 cores, 3.2GHz, 4-issue per core,14-stage pipeline
Functional units	2 IntALU, 4 LSU, 2 FPALU
IQ, ROB and LSQ size	IQ 32, ROB 128, LQ 48, SQ 44
Physical register num	128 Int, 128 FP, 128 BR, 128 ST
Branch prediction	Combined, 6k Bimodal + 6k Two-level, 1K RAS, 4k-entry and 4-way BTB
L1 caches (per core)	64KB Inst/64KB Data, 2-way, 16B line, hit latency: 3-cycle Inst,3-cycle Data
L2 cache (shared)	4MB, 8-way, 64B line, 13-cycle hit latency
Memory	DDR3-1600, 2 channels, 2 DIMMs/channel, 2 ranks/DIMM, 8 banks/rank
Memory controller	64-entry buffer, 15ns overhead for scheduling and timing
DDR3 DRAM latency	DDR3-1600:11-11-11

Table 3.3: Major simulation parameters.

Parameters	**Value**
Normal voltage	1.5V
Active precharge current	95mA
Precharge power-down standby current	35mA
Precharge standby current	42mA
Active power-down standby current	40mA
Active standby current	45mA
Read burst current	180mA
Write burst current	185mA
Burst refresh current	215mA

Table 3.4: Parameters for calculating power for 2Gbit, x8 DDR3-1600(11-11-11) device.

Workload	Applications	Workload	Applications
MEM-1	mcf, libquantum, soplex, milc	ILP-1	gobmk, sjeng, gcc, namd
MEM-2	sphinx3, soplex, libquantum, mcf	ILP-2	gobmk, sjeng, gcc, perlbench
MEM-3	lbm, mcf, soplex, sphinx3	ILP-3	gobmk, sjeng, deaIII, perlbench
MEM-4	mcf, libquantum, lbm, sphinx3	ILP-4	gobmk, deaIII, namd, perlbench
MEM-5	soplex, milc, lbm, milc	ILP-5	sjeng, gcc, namd, perlbench
MEM-6	libquantum, lbm, milc, sphinx3	ILP-6	gcc, deaIII, namd, perlbench
MIX-1	lbm, sphinx3, deaIII, perlbench	MIX-2	lbm, milc, deaIII, namd
MIX-3	libquantum, gcc, milc, namd	MIX-4	soplex, sjeng, libquantum, gcc
MIX-5	gobmk, sphinx3, mcf, perlbench	MIX-6	mcf, soplex, gobmk, sjeng

Table 3.5: Workload specifications.

3.4.1 Statistical Memory MTTF Model

Intuitively, SECDED will improve the memory reliability compared with non-ECC DIMM. However, few studies directly report the Mean Time To Failure (MTTF) in academy. We have used a statistical MTTF model suggested by a previous study [75]. For a memory system using SEC (Single-bit Error Correcting) codes, the system is considered operational until there are two bits of error occurred within a single codeword. Assume the generation of failure bit in memory is independent, random and follows Poisson process within time period T. We build a fault simulator based on this error model. Monte Carlo method is used and random errors are generated and injected to a memory system. When an error is injected, a check function is called to estimate if the number of error bits in one codeword exceed the ECC capability. If so, the simulator reports a system failure and records the failure time. Each experiment is repeated for a sufficient number of times, and then the failure time records are used to calculate the MTTF using the above MTTF model.

We simulate a quad-core system with each core running a distinct application from the SPEC2006 [98] benchmark suits. We follow the method used in a previous study [52] to group those benchmarks into three categories: MEM (memory-intensive), ILP (compute-intensive) and MIX (mix of memory-intensive and compute-intensive) based on their L2 cache misses per 1000 instructions (L2 MPKI). The MEM applications are those benchmarks with L2 MPKI greater than 10 and ILP applications are those with less than one L2 MPKI.

Table 3.5 shows 18 four-core multi-programming workloads randomly selected from those

applications. The MEM workloads consist of memory-intensive applications, the ILP workloads contain compute-intensive applications, and the MIX workloads have the MEM benchmarks and ILP benchmarks mixed together. For each workload, we create a checkpoint after all the benchmarks in one workload running to their typical phases. Then we collect the detailed simulation results from user space for 200 million instructions.The performance is characterized using SMT weighted speedup [96] $\sum_i^n (IPC_{multi}[i]/IPC_{single}[i])$, where n is the total number of applications running, $IPC_{multi}[i]$ is the IPC value of application i running under multi-core environment and $IPC_{single}[i]$ is the IPC value of the same application running alone.

3.5 Experimental Results

In this section, we present and analyze the evaluation results of E^{3}CC, including performance, memory traffic overhead, and power efficiency. We have conducted experiments with the full rank, 32-bit sub-ranked, and 16-bit sub-ranked memories, and with cacheline-interleaving and page-interleaving.

3.5.1 Overall Performance of Full-Rank Memories

It is straightforward to use a conventional DDR3-1600 non-ECC memory as a baseline for performance comparison. However, we also simulated a pseudo DDR3-1422 ECC DRAM with a 72-bit width data bus running at 711MHz since it provides the same raw data bandwidth with E^{3}CC with a 64-bit data bus running at 800MHz. Note that the data rate of 1422MHz is an artificial setting not used in practice.

Figure 3.7a compares the performance of the full-rank E^{3}CC DDR3-1600 memory, without and with ECC-cache, against full-rank DDR3-1600 baseline and the full-rank pseudo DDR3-1422 memory. We first focus on the memory-intensive workloads. Without ECC-cache, E^{3}CC incurs an average performance loss of 9.1% and 13.7% with cacheline- and page- interleaving, respectively. The performance loss comes from extra column access for ECC bits, which fetches more ECC bits than necessary and thus incurs bandwidth overhead. It will be analyzed in Section 3.5.3. ECC-cache effectively reduces the bandwidth overhead. With ECC-cache, the performance loss is reduced to 5.1% with cacheline-interleaving and 7.4% with page-interleaving.

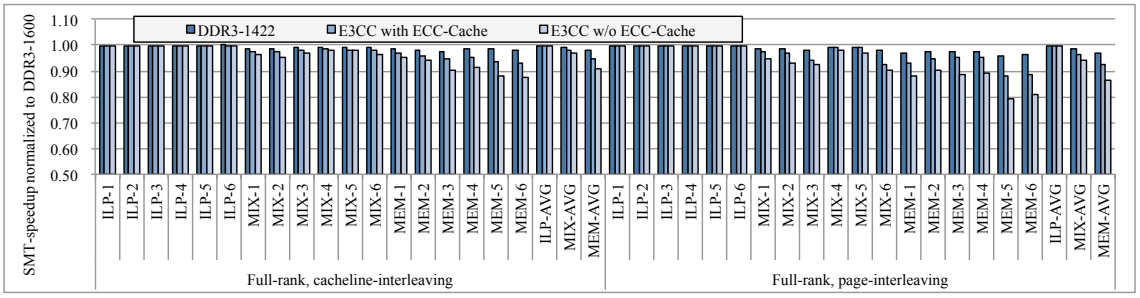
When E^{3}CC is compared with pseudo DDR3-1422, the performance overhead is 3.4% and 4.5% for cacheline- and page- interleaving, respectively.

For mixed workloads, E^{3}CC without ECC-cache incurs an average performance overhead of 3.2% and 5.7% with the cacheline- and page- interleaving, respectively. With ECC-cache, the performance overhead is reduced to 2.0% for cacheline-interleaving and 3.4% for page-interleaving. When E^{3}CC is compared with DDR3-1422, the performance losses are 1.0% and 2.0%, respectively. For ILP workloads, the average performance overhead is under 0.3% for both interleaving schemes, as ILP workloads are not sensitive to memory performance from the beginning.

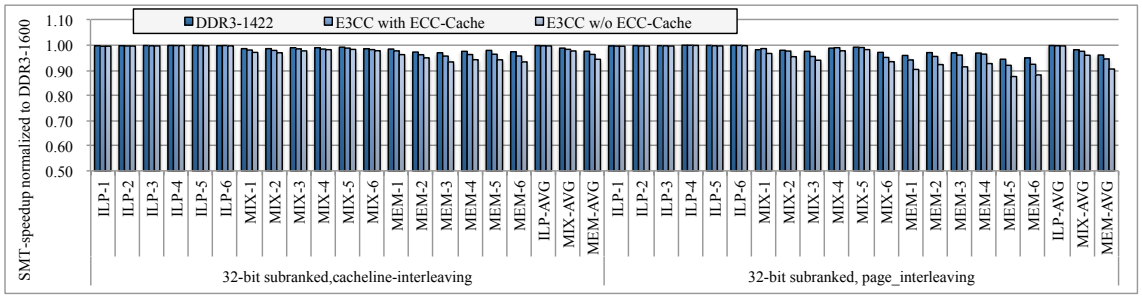
3.5.2 Overall Performance of Sub-Ranked Memories

Figure 3.7b compares the performance of 32-bit sub-ranked E^{3}CC DDR3-1600 memories with sub-ranked DDR3-1600 baseline and pseudo DDR3-1422 memories. With sub-ranking, the bandwidth overhead from fetching ECC bits is reduced significantly as discussed in Section 3.3.4. For memory-intensive workloads and without ECC-cache, the average performance overhead is 5.6% and 9.5% with cacheline- and page-interleaving, respectively. With ECC-cache, the overhead is reduced to 3.7% and 5.5%, respectively. When E^{3}CC is compared with DDR3-1422, the overhead is further reduced to 1.3% and 1.6%, respectively. For mixed workloads and without ECC-cache, the performance overheads are 2.3% and 4.0% from DDR3-1600 with cacheline- and page-interleaving, respectively; and with ECC-cache, the overhead is reduced to 1.7% and 2.5%, respectively. When E^{3}CC is compared with DDR3-1422, the performance difference is almost negligible.

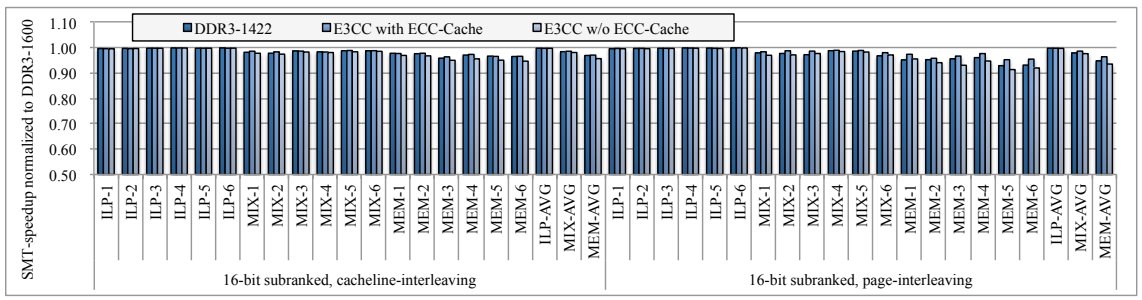
Figure 3.7c compares the performance of 16-bit sub-ranked memories of the E^{3}CC DDR3-1600 type against the sub-ranked DDR3-1600 baseline and the pseudo DDR3-1422 type. For memory-intensive workloads and without ECC-cache, the performance overhead is 4.3% and 6.5% for cacheline- and page-interleaving, respectively. With ECC-cache, the overhead is reduced to 2.9% for cacheline-interleaving and 3.6% for page-interleaving. When compared to DDR3-1422, E^{3}CC even improves the performance on page-interleaving. Although this scenario seems to be counter-intuitive, it is possible because the fetch of extra ECC bits has an



(a) Full-rank E^{3}CC memory vs. full-rank DDR3-1600 memory, with full-rank DDR3-1422 included.



(b) 32-bit sub-ranked E^{3}CC memory vs. 32-bit sub-ranked DDR3-1600 memory, with 32-bit sub-ranked DDR3-1422 included.



(c) 16-bit sub-ranked E^{3}CC memory vs. 16-bit sub-ranked DDR3-1600 memory, with 16-bit sub-ranked DDR3-1422 included.

Figure 3.7: Performance of the E^{3}CC DDR3-1600 memories and the baseline DDR3-1600 memories of different rank sizes. E3CC denotes E^{3}CC.

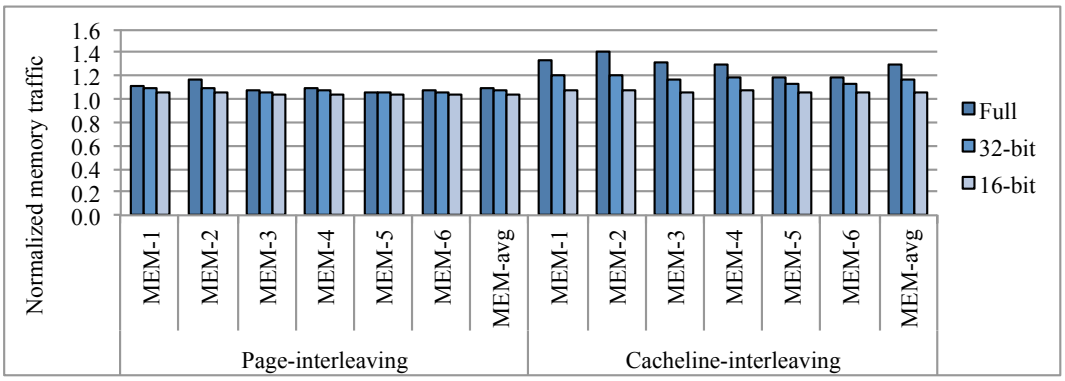


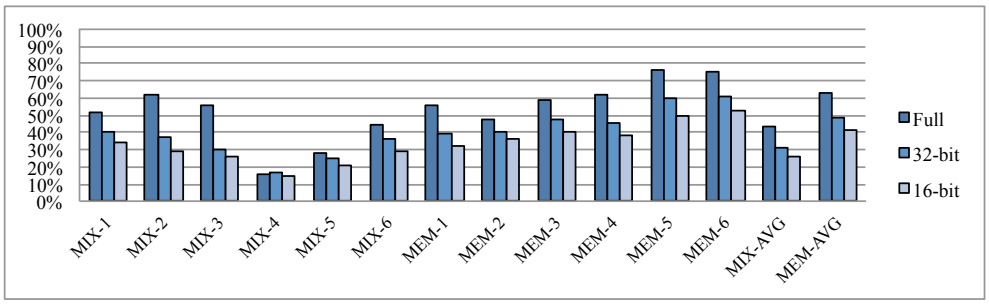
Figure 3.8: The extra read memory traffic caused by E^{3}CC when ECC-cache is used.

effect of prefetch.

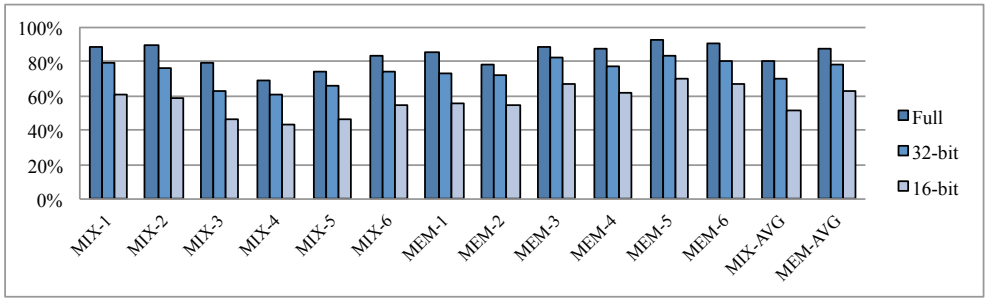
3.5.3 Memory Traffic Overhead and ECC-Cache

Figure 3.8 presents memory traffic overhead caused by E^{3}CC when ECC-cache is used. As memory read is critical to the system performance and only memory-intensive workloads are sensitive to the memory traffic overhead, the figure presents only the memory read traffic for the memory-intensive workloads. The average overhead is 29.0%, 17.1% and 6.5% for full-rank, 32-bit sub-ranked and 16-bit sub-ranked memories, respectively, with cacheline-interleaving. And with page-interleaving, the overhead is 9.8%, 7.3% and 4.1% for those three rank configurations, respectively. The overhead is less with page-interleaving than with cacheline-interleaving because page-interleaving retains row-level spatial locality in the program address space and the ECC-cache can well capture the spatial locality. ECC-cache is still effective for cacheline interleaving, as it still captures the spatial locality within multiple blocks sharing the same ECC column.

Figures 3.9a and 3.9b show the ECC cache read hit rates for cacheline- and page-interleaving, respectively. Overall, ECC-cache has a high read hit rate for memory-intensive workloads. The average hit rates are 62.5% and 87.2% for x64-rank DIMM with cacheline-interleaving and page-interleaving, respectively. Cacheline-interleaving distributes the memory requests among all the ranks, which improves the accessing parallelism and reduces the row buffer hit rate. Therefore, ECC cache hit rate for cacheline-interleaving is smaller than that of page-interleaving mode.



(a) Cacheline-interleaving



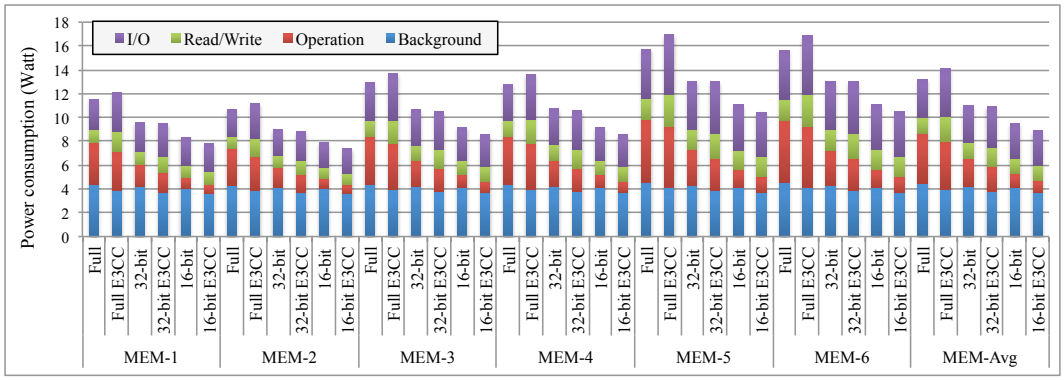
(b) Page-interleaving

Figure 3.9: ECC-cache read hit rate for mixed and memory-intensive workloads with cacheline- and page-interleaving.

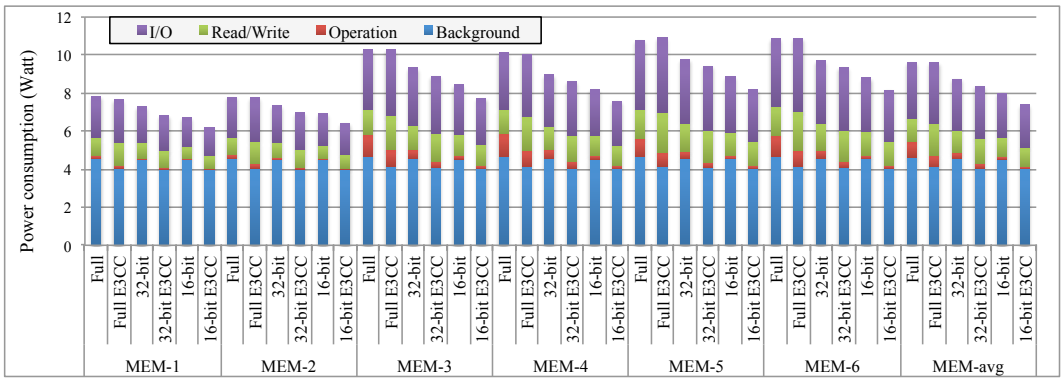
As the rank size decreases to x32 and x16, the cache hit rate is reduced because the program locality is further broken as row buffer size shrinks. On average, the row buffer hit rates are 46.4% and 40.3% with cacheline-interleaving with x32 and x16 rank size, respectively; and 78.1% and 62.5% with page-interleaving, respectively.

3.5.4 Power Efficiency of E^{3}CC Memories

As memory-intensive workloads are power hungry applications, we focus on the power efficiency of these workloads. Figure 3.10 compares the power breakdown of full-rank and sub-ranked memory without and with ECC. Sub-ranked memories without ECC are shown to give a baseline reference for the corresponding sub-ranked E^{3}CC memories, so as to show the power efficiency of E^{3}CC in each type of sub-ranked memory. Both are DDR3-1600 type. Consistent with previous studies [121, 2, 1], sub-ranked memories reduce memory operation power significantly as the sub-rank size decreases. The background power is also reduced moderately because of the increased number of sub-ranks over the number of ranks, as more sub-ranks can



(a) Cacheline-interleaving



(b) Page-interleaving

Figure 3.10: Memory power breakdown into operation, read/write, I/O and background power for full-rank and sub-ranked memories with and without ECC. E3CC denotes E^{3}CC; ECC-cache is used. Only memory-intensive workloads are shown. Sub-ranked memories without ECC are used as reference baseline, with power calculated as that of regular sub-ranked memories plus 12.5% overhead. Both the baseline and E^{3}CC are DDR3-1600 type.

be put into fast power-down mode to save background power. For page-interleaving, operation power is not a significant component, because the row-buffer hit rate of those workloads is high and therefore a less number of precharges and activations are needed.

E^{3}CC achieves good power efficiency when compared with the baseline. The full-rank baseline is actually the conventional full-rank ECC DIMM. The sub-ranked baselines correspond to a design that extends sub-ranked memories ideally to implement ECC; for example, for 32-bit sub-ranked memory of four x8 device per sub-rank, it extends the sub-rank bus to 36-bit and adds an x4 device to store ECC bits, assuming that the x4 device consumes half power of a x8 device.

We first focus on the full-rank memory. For cacheline-interleaving, E^{3}CC increases the overall power rate of full-rank memory by 5.6% from the baseline. Regarding each power component, the read/write power increases by 52.4% and I/O power by 23.7% but the background power decreases by 9.9% and operation power by 4.5%. For page-interleaving, E^{3}CC is almost the same as the baseline, with the overall power rate decreased by 0.1%. It increases the read/write power by 34.0% and I/O power by 9.1% but decreases the background power by 10.8% and operation power by 25.1%. The read/write and I/O power increases come from accessing extra ECC bits. ECC-cache is more effective on page-interleaving than on cacheline-interleaving, and therefore the increases are less on the page-interleaving than on the cacheline-interleaving. The background power is reduced because, compared with the baseline, there are 8 devices per rank instead of 9. Operation power is reduced partially for a similar reason, as there are 8 devices per rank to precharge and activate instead of 9. Additionally, we have found that the average row buffer miss rate of page-interleaving is 15.9% with E^{3}CC and 17.5% with the baseline, which means less precharges and activations with E^{3}CC. For all settings, because of the BCR mapping, E^{3}CC tends to have lower row-buffer miss rate than the baseline.

The power efficiency of E^{3}CC vs. the baseline increases with sub-ranked memory because of the decreased memory traffic overhead. For 32-bit sub-ranked memory with cacheline-interleaving, E^{3}CC on average reduces the overall power rate by 1.1%, with 10.7% decrease of background power, 9.9% decrease of operation power, 22.3% increase of read/write power, and 8.0% increase of I/O power. With page-interleaving, on average it reduces the overall power rate by 4.6%, with 10.9% decrease of background power, 20.2% decrease of operation power, 13.6% increase of read/write power, and 0.6% increase of I/O power. For 16-bit sub-ranked memory with cacheline-interleaving, E^{3}CC on average reduces the overall power rate by 7.7%, with changes of -11.0%, -14.7%, +1.8% and -5.2% on background, operation, read/write and I/O power, respectively. With page-interleaving, it on average reduces the overall power rate by 6.1%, with changes of -10.5%, -13.4%, +5.4% and -2.1% on background, operation, read/write and I/O power, respectively.

We have also found that while ECC-cache reduces the memory ECC read traffic efficiently, it does not reduce the memory ECC write traffic as well because there is much less spatial

locality in the write traffic than in the read traffic. The memory ECC write traffic is caused by last-level cache writebacks, whose spatial locality is mostly lost because the relative timing of writebacks is different from the relative timing of demand misses on the same set of memory blocks. Although the extra write traffic only has a moderate impact on the performance of those workloads, it does increase the read/write and I/O power. Our design uses the burst-chop-4 to reduce the ECC write traffic and power consumption. The use of a different processor cache writeback policy, for example eager writeback [55] or a revised scheme, may restore the spatial locality in cache writebacks and therefore may help reduce this power increase. If so, E^3CC will be even more power-efficient.

3.5.5 Evaluation of Using Long BCH Code

As discussed in Section 3.3.5, the flexibility of E^3CC may allow the use of very long BCH code for improved storage efficiency and reliability. Because the storage overhead is reduced, power efficiency and performance will also be improved. We have applied a statistical MTTF model suggested by a previous study [75] and built a Monte-Carlo simulator that assumes the generation of failure bit is independent, random and follows the Poisson process. The simulation results have been further verified with mathematical model for evaluation. We have simulated a 4GB memory system using BCH(512,522), BCH(512,532) and BCH(512,542), which are SEC, DEC and TEC codes, respectively, and with 2.0%, 3.9% and 5.9% storage overhead, respectively. We found that those long BCH codes may improve MTTF by two, four, and five orders of magnitude, respectively.

3.6 Summary

In this chapter, we have presented E^3CC, a complete solution of memory error protection for sub-ranked and low-power memories. A novel address mapping scheme called BCRM has been found to implement page-interleaving on E^3CC without using expensive integer division. A simple and effective solution called ECC-cache is presented to reduce extra ECC traffic. We have thoroughly evaluated the performance and power efficiency of E^3CC. E^3CC is suitable for consumer-level computers and mobile devices used in applications that require a certain degree

of reliable computing but desire a low impact on cost and power efficiency. E^3CC does not require any changes to memory devices or modules, further reducing the cost of manufacturing. Furthermore, using ECC or not can be configured at system booting time, giving users the flexibility to make trade-off between reliability, performance, and power efficiency.

CHAPTER 4. EXPLORING FLEXIBLE MEMORY ADDRESS MAPPING AT DEVICE LEVEL FOR SELECTIVE ERROR PROTECTION

Memory error protection is increasingly important as memory density and capacity increase; however, the protection comes with inherent storage and energy overhead. Selective error protection (SEP) is desirable for reducing the overhead, particularly on mobile computers and devices. In this chapter, we further explore flexible and efficient memory address mapping schemes at DRAM device level to support SEP on commodity memory devices and conventional modules in a reconfigurable manner, such that the physical memory address space can be split into two memory regions without and with memory error protection, respectively.

4.1 Introduction

Diverse applications have been developed on mobile systems including tablets and smartphones. These applications react differently to memory errors. For most common smartphone apps, including web-browsing, gaming, video player, audio player, and others, the program data is generally non-critical or insensitive to memory errors. For example, an error in a frame buffer during video processing may cause a pixel disruption in one frame, which is tolerant to human eyes [95, 63]. In contrast, some apps like mobile banking and financing, or partial code of apps like loop-control variables and others, are sensitive to errors. A recent study proposes a methodology to quantify the difference; for example, a variable "ZCH" in *lex* benchmark is evaluated to be around 1188 times more vulnerable than another variable "yychar" in the same program [68]. For video processing, while errors in a frame buffer are not disruptive, errors in vertex arrays and texture contents may cause visual discomfort because they disrupt the

appearance of many frames [95].

Memory error protection in mobile systems must be done carefully. On one hand, in the worst cases memory errors may incur severe consequences to systems and users, i.e. data corruption, visual discomfort, security vulnerability [27, 108] and others. On the other hand, it is a waste to protect the entire memory, because memory error protection incurs extra power consumption and reduces battery usage time. Therefore, it is usually desired to only protect critical code and data and leave the rest unprotected. For this purpose, a recent study proposes a framework of Selective Error Protection (SEP) [68]. In that framework, the OS and compiler place critical data (including code thereafter) in a designated protected region in the physical memory address space based on a profile-based data criticality analysis. For example, data and variables of high access frequency may be selected. The evaluation confirms that SEP can provide high error coverage with low energy consumption.

However, the study does not fully investigate the memory system design to support an ECC (Error Correcting Code)-protected region and a non-protected region; it simply assumes such a mechanism exists, and then focuses on the higher layers of the framework. It is actually very challenging to design such a memory system for mobile computers and devices or any computer of a simple memory organization.

In this chapter, we investigate the mechanism to support an error-protected region and a non-protected region on *a simple memory system made by commodity memory devices*. Specifically, such a memory system can be viewed as a collection of memory devices organized in channels and ranks without builtin support for ECC. We first employ *Embedded ECC* [121] to support ECC in the protected region because embedded ECC is energy-efficient and does not require an extra device to store ECC bits. The new challenge, however, is to fully address the device-level memory address mapping with two memory regions.

This work is partially motivated by our previous work E^{3}CC, a form of Embedded ECC which stores ECC bits with data bits in the same DRAM row. In E^{3}CC, we design an efficient embedded ECC scheme for narrow-ranked low-power DRAM memories considering both reliability and energy efficiency. Because of ECC embedding, the effective memory capacity is no longer power-of-two. We therefore propose BCRM (Biased Chinese Remainder Mapping) for

efficient device-level memory address mapping without using division. BCRM is used for the entire physical memory address space. This chapter addresses a different problem, namely how to split the address space into two regions with efficient device-level memory address mapping. In our framework, we assume embedded ECC is used in the protected region. The non-protected region, however, may not use power-to-two mapping because the two regions co-exist in the same memory system and affect each other. With hardware support and software management, their boundary may shift to vary the size of the two regions. The device-level address mapping must accommodate this change.

We propose novel address mapping schemes to address the challenge. First, we extend the CRM (Chinese Remainder Mapping) scheme to non-prime memory systems [25]. CRM is designed for prime memory systems decades ago and it applies CRT (Chinese Remainder Theorem) to do address translation. Similar to CRM, the proposed scheme uses modulo operation to replace Euclidean division. Second, for cases that CRM-based mapping is not adequate to maintain DRAM accesses locality, we propose a section-based mapping scheme. It partitions the coming physical addresses into sections based on greatest common divisor and applies modulo-based mapping section by section. Generally, DRAM device-level address mapping either distributes memory requests evenly among memory units for high parallelism or maintains row buffer locality for low accessing latency. We thus propose *adjustment-factor*s for the proposed mapping schemes to maintain one of the two common properties[1]. The proposed schemes can have multiple variations by extending different *adjustment-factor*s or by combining these schemes together. They, therefore, are flexible to favor system requirements in practice.

We have made the following contributions in this chapter:

- We propose C-SCM (*CRM based Super-Column Mapping*), which groups multiple columns as a *super-column* to apply CRM.

- We propose C-SGM (*CRM based Super-Group Mapping*), which groups both rows and columns as *super-row* and *super-column*, respectively, to apply CRM.

[1]In study [47], it shows that four cacheline blocks interleaving is optimal for many-core system considering power, performance and fairness. It is not discussed in detail in our study but our proposed schemes can be extended to maintain such a property.

- We propose S-SRM (*Section based Shift-Row Mapping*), which is not based on CRM. It can be adjusted to present good row buffer locality.
- We further propose *adjustment-factor*s including *adjusting-factor*, *breaking-factor* and *shifting-factor*. The proposed mapping schemes can have multiple variations by applying different factors or by combining different schemes to present different properties. It hints the design flexibility in practice to favor system requirements.

The rest of this chapter is organized as follows. Section 4.2 introduces the background and related work of the study. In Section 4.3, the SEP design challenges are presented in detail by showing an example mapping layout. Section 4.4 devises multiple address mapping schemes for different system requirements of SEP. In Section 4.5, we discuss various scenarios that our proposed mapping schemes can be applied and Section 4.6 concludes this chapter.

4.2 Background and Related Work

4.2.1 Diverse Sensitivities of Data, Variables and Applications

Intuitively, data and variables of application programs may not be uniformly sensitive to memory errors, in the sense how an error may impact a program's output or behavior observed by users or other computer systems. First, within an application, data and variables may exhibit diverse sensitivities to memory errors. For example, pixel data in an image or a video clip is insensitive to errors as a single pixel failure is mostly invisible to users. However, errors in other data structures, like the texture contents or loop control variables, can lead to much more severe consequences. A recent study [68] shows an example that variable "ZCH" in *lex* benchmark is around 1,188 times more vulnerable to errors than variable "yychar" in 5,000 experiments. It conducts a profile-based criticality evaluation and shows a case that 35.8% error protection can achieve around 99% fault coverage. Another study [110] observes that the dynamic segment of program memory is 18 times more sensitive to errors than the static segment. Second, different applications show different levels of sensitivity to memory errors. For example, gaming, audio, video and other multimedia applications, and web browsers are insensitive to memory errors while banking, financing or mission critical applications are sensitive to errors. A study [97] has

proposed PVF (Program Vulnerability Factor) to quantify application vulnerability to errors and found diverse PVFs of programs; for example, $\text{PVF}_{applu} = 86.2\%$ and $\text{PVF}_{equake} = 48.9\%$ for the two SPEC CPU2000 benchmark programs.

4.2.2 DRAM Accessing Page Policies

There are two commonly used page policies: close-page and open-page. Close-page policy attempts to precharge 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 less row activations are required. These two policies are equally important in practice and their variants have been widely used among commodity machines.

4.2.3 Related Work

Memory error protection has been widely studied [113, 114, 103, 73, 68]. Virtualized ECC [113, 114] proposes to maintain ECC in memory data space and relies on last level cache to cache unused ECC to reduce memory traffic. The design is flexible and efficient compared to conventional SECDED DIMM. LOT-ECC [103] proposes a tiered ECC protection to provide Chipkill-level protect with merely nine (x8) DRAM devices. Archshield [73] proposes an architectural-level framework to protect fabrication faulty cells caused by extreme scaling of DRAM. All those studies apply error protection to the entire memory space. A recent study [68] proposes SEP for low-cost memory protection, which is closely related to our work. The details have been discussed in Section 4.1. Other works [12, 56] also discuss selective data protections. By comparison, this work is focused on device-level memory address mapping to support SEP.

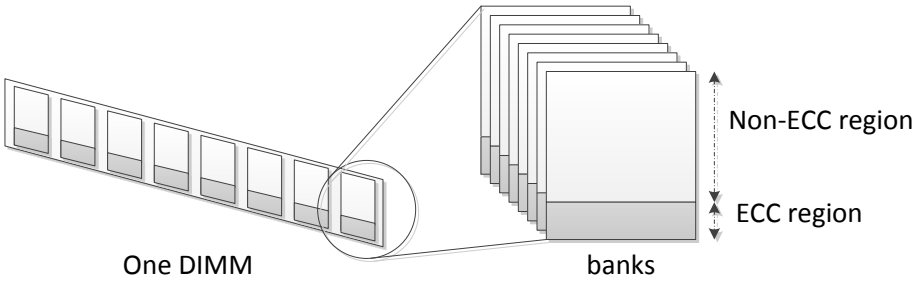


Figure 4.1: Example row-index based partitioning for selective protection. Part of the rows on all banks are protected by ECC while others are not.

4.3 Problem Presentation

When memory space is partitioned into two regions, the effective capacity of the two regions may no longer be power-of-two. It introduces challenges of DRAM device-level address decomposition given that complex Euclidean division should be avoided [25]. In this section, we analyze the problem in detail and show an example mapping layout to motivate our study.

4.3.1 DRAM Device-Level Address Mapping

In Chapter 3, the detailed DRAM address mapping of cacheline- and page- interleaving schemes are presented. As shown in Figure 3.5, cacheline-interleaving scheme maps the least significant bits of block address to channel index, followed by DIMM[2], rank, etc. It attempts to map continuous memory requests evenly to different channels, DIMMs, ranks and others. Therefore, it typically works with close-page policy for high parallelism. On the contrary, page-interleaving scheme breaks the least significant bits to column indexes. It therefore maintains DRAM row buffer locality as continuous addresses within a page boundary are mapped to the same DRAM row. With open-page policy, page-interleaving scheme retains the spatial locality in a program's access pattern and reduces memory access latency. These two interleaving-schemes are equally important in practice, similar to the two page policies, as they present either high parallelism or high locality. We thus denote the two properties as *access-parallelism* and *access-locality*, respectively, to simplify our presentation, and our proposed mapping schemes should retain these properties.

[2]If DIMM is not applicable like in smartphones, number of its index bits is 0.

4.3.2 Address Mapping Issue in SEP

In SEP, memory space is partitioned into two regions for different error protection strategies. In order to make the partitioning flexible and generic, we divide the two regions by *row index* in each bank. Figure 4.1 shows an example partitioning. For rows in each bank with row indexes greater than a given parameter, they are protected by ECC and the remaining rows are not. This partitioning is flexible as a typical DDRx device contains a large number of rows, 32K for example. One can alternatively partition the space by other indexes, i.e. channel, DIMM, rank, bank, etc. However, there are drawbacks. First, partitioning by channel, DIMM, rank or bank presents less flexibility as the number of those units can be limited in a system. Moreover, partitioning by column index is inappropriate as it reduces page size of one region and thus the maximum page locality. In addition, all those alternatives share the same issue with row-indexed partitioning in SEP implementation as one region might have a non-power-of-two size. Therefore, all of our following discussions assume *row-indexed* partitioning.

The major concern with partitioning is how to map a physical address to a designated region since its capacity may not be power-of-two. Conventional interleaving schemes require a power-of-two system for address decomposition. Without this prerequisite, the mapping can be wrong. To further clarify the problem, we show an example below. Without losing generality, all channel, DIMM, rank and bank indexes are ignored in the presentation for simplicity. The mapping issue is thus reduced to map a physical address $d \in [0, R \cdot C - 1]$ to a 2D array $L_{R \times C}$. Let $R = 8, C = 8$ in entire space. In a case that property of *access-parallelism* is required by the system, without partitioning, cacheline-interleaving scheme can be employed. The layout is shown in Table 4.1. With cacheline-interleaving scheme, the lower three bits of the address are the row index and the upper three bits are the column index. For example, address 49 (110001 in binary) maps to row 1 and column 6 (starting from 0) since row index and column index are 001 and 110, in binary, respectively.

However, conventional address mapping may not work when the space is partitioned into two regions. Assume an example partitioning ratio of 3 : 1 for non-ECC region and ECC region, the upper six rows are thus not protected while the remaining two rows are protected

row/col	0	1	2	3	4	5	6	7
0	0	8	16	24	32	40	48	56
1	1	9	17	25	33	41	49	57
2	2	10	18	26	34	42	50	58
3	3	11	19	27	35	43	51	59
4	4	12	20	28	36	44	52	60
5	5	13	21	29	37	45	53	61
6	6	14	22	30	38	46	54	62
7	7	15	23	31	39	47	55	63

Table 4.1: An example layout of the entire space with cacheline-interleaving scheme assuming $R = 8, C = 8$.

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

Table 4.2: An example layout with page-interleaving scheme for the region with six rows without ECC protection.

by ECC based on the *row-indexed* partitioning. With such partitioning, addresses $d \in [0, 47]$ are required to map to $L_{6\times8}$, namely row $[0, 5]$ by column $[0, 7]$. As there are merely six rows in the non-protected region, it is challenging to apply cacheline-interleaving scheme. If one simply takes the last three bits of physical address as row index, the addresses $d \in [0, 47]$ are mapped to $L_{8\times6}$ instead of the required $L_{6\times8}$. One can propose to use page-interleaving scheme to map addresses into $L_{6\times8}$ as shown in Table 4.2. However, it presents *access-locality* instead of the required *access-parallelism*. One can even propose complex division-based mapping scheme. However it is not allowed in practice due to division complexity and harm to system performance and power efficiency as presented in Chapter 3. We thus devise novel address mapping schemes to resolve this challenge for implementation of selective protection.

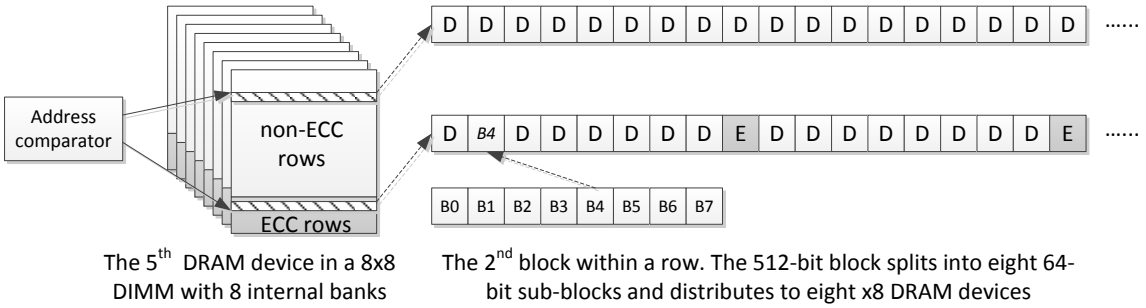


Figure 4.2: Overview of SEP design and data/ECC layout. Cacheline size is 512 bits inside a row and SECDED (72,64) code is applied for error protection.

4.4 Novel Address Mapping Schemes

In this section, we first present the particular SEP design and then devise generic address mapping schemes mathematically, considering various combinations of number of rows and columns. In the end, we show case studies for applying the proposed mapping schemes on real DDRx memory system for selective protection.

4.4.1 SEP Design Overview

In SEP memory, one region is reserved to be protected by ECC. Memory data is selectively transferred between memory controller and DRAM devices via ECC unit if its address is within the protected region. The memory controller maintains in a register the address of the boundary between the two regions. For addresses greater than or equal to the boundary address, the requests are protected by ECC. Otherwise, ECC circuitry is bypassed. The SEP framework can be referred to study [68].

At DRAM level, the error-protected region requires extra storage for ECC redundancy. As part of memory space is not protected, the conventional way to add an extra DRAM device is inappropriate. We employ the embedded ECC scheme [121], which stores ECC bits with data in the same DRAM row to reduce the energy consumption in accessing ECC bits. Figure 4.2 presents an example data and ECC layout of the SEP design. Given a positive output from address comparator, the memory request falls into ECC region and its ECC redundancy is placed following the data. With SECDED protection, 8-byte ECC is generated for each 64-

byte cacheline block. Therefore, there is one ECC block following every eight data blocks in a DRAM row. The in-block ECC layout is similar to that of E^{3}CC presented in Chapter 3.

4.4.2 Exploring Generic Address Mapping Schemes

As presented in Section 4.3, the DRAM device-level address mapping issue emerges when the space is split into two regions. We thus devise novel and efficient mapping schemes to resolve this challenge. First, we discuss CRM (Chinese Remainder Mapping) scheme as multiple of our proposed schemes are based on it. Then we illustrate the devised mappings case by case to cover all possible combinations of number of rows and columns.

CRM was proposed relying on CRT (Chinese Remainder Theorem) [25] for prime memory systems. It maps a physical address $d \in [0, RC-1]$ to a position in the 2D array $L_{R\times C}$ one to one using the following formula[3]:

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

, in which r and c are row and column indexes of the array and $r \in [0, R-1], c \in [0, C-1]$. CRM guarantees that the translation from d to a $\langle r, c\rangle$ pair is a one-to-one mapping given the prerequisite that R and C are coprime.

Table 4.3 shows an example layout of a block address d with this mapping, assuming $R = 7, C = 8$. Using the CRM formula, address $d \in [0, 55]$ is translated to a position in the 2D array correspondingly. For example, $d = 50$ maps to $\langle 1, 2\rangle$ (starting from 0). In the mapping, every address is translated to exactly one designated position and each position of the 2D array is addressed by exactly one d.

In SEP with a simple memory system of commodity devices, the address mapping is more complicated for two reasons. First, the number of rows and columns may not be coprime as required in CRM. Second, as discussed in Section 4.3.1, either *access-parallelism* or *access-locality* is usually required to maintain for practical system. For all the following discussions, define $g = gcd(R, C)$, in which g is the GCD (Greatest Common Divisor) of R and C. Our

[3]The notations are different from that in paper [25].

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

Table 4.3: An example layout of Chinese Remainder Mapping assuming $R = 7, C = 8$.

discussions focus on the case that g is 2^n, which is true in most cases in real DDRx system as shown in Section 4.4.3. Let $|g| = log_2(g)$. We thus have the following two cases based on g.

Case I: $g = gcd(R, C) > 1$ and $gcd(R, C/g) = 1$. In this case, R and C are not coprime but R and C/g are coprime. We thus propose C-SCM (*CRM based Super-Column Mapping*) using the following formula:

$$\begin{aligned} r &= d_s \bmod R \\ c &= (d_s \bmod t) \ll |g| + (d \bmod g) \end{aligned} \tag{4.1}$$

where $d_s = d \gg |g|$, which is called *super-address*, $t = C \gg |g|$, which is called *super-column*, and $d_s \in [0, Rt - 1]$. By grouping multiple columns together, the number of rows and *super-column*s are coprime and thus the CRM scheme can be applied. This mapping is rational as it simply extends CRM to an array with the same number of rows and a reduced number of *super-column*s. The left-shift operation in expression of c expands the *super-column*s to normal columns by a scaling-factor g and $(d \bmod g)$ calculates the offset of the mapping inside a *super-column*.

Table 4.4 presents an example layout with the proposed C-SCM scheme assuming $R = 8, C = 6$. In the example, $g = 2$ and $|g| = 1$. Thus two columns are grouped as a *super-column*. The addresses with same *super-address* are mapped to the same *super-column*. For example, *super-address* d_s of addresses $d = 2, 3$ is 1 and it is mapped to row 1, *super-column* 1. By scaling the *super-column*, addresses $d = 2, 3$ map to normal columns 2 and 3, respectively.

The example above maps addresses within range of g to the same row, which diverges

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

Table 4.4: An example layout of C-SCM with $R = 8, C = 6$.

slightly from *access-parallelism*. To maintain *access-parallelism*, one can apply XOR mapping scheme [119] first before this mapping. For this layout, we propose *breaking-factor* to break the small locality within range of g so that memory requests are evenly distributed into memory units. The selection of *breaking-factor* is to map the logically continuous addresses to different rows so that they are physically scattered. There can be multiple options of *breaking-factor*. In this example, we use "$+c$" as the factor and r is thus updated to $r = (d_s + c) \bmod R$. Table 4.5a shows the updated layout. As it shows, the small locality of size g is completely broken. For example, addresses 2 and 3 now map to rows 3 and 4, respectively. All the continuous addresses are now in different rows.

In case that *access-locality* is required in practice, the proposed C-SCM cannot be applied directly as it shortens row buffer locality to a smaller size g. Similar to *breaking-factor*, we propose to add an *adjusting-factor* to draw back the offset of the mapping. We observe that a continuous set of d_s's are mapped to physically continuous rows. Therefore, by adding an offset "$-(d_s \bmod t)$" in calculating row index, the biased row indexes are adjusted. r is thus updated to $r = (d_s - (d_s \bmod t)) \bmod R$. Similar method is proposed in Chapter 3. However, we explicitly write it with *super-column* assumption and BCRM (Biased Chinese Remainder Mapping) is just a particular case of this study. Table 4.5b shows the updated layout. All the continuous addresses within a row boundary are now mapped to the same row and the maximum row buffer locality is maintained.

The rationality of the two adjustment factors introduced in C-SCM is similar to that of the

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

(a) An example layout of C-SCM with *breaking-factor* c.

row/col	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

(b) An example layout of C-SCM with *adjusting-factor* $-(d_s \bmod t)$.

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

(c) An example layout of C-SCM with both *adjusting-factor* $-(d_s \bmod t)$ and *breaking-factor* $+c$.

Table 4.5: Example layouts of C-SCM with *breaking-factor* and *adjusting-factor*, assuming $R = 8, C = 6$.

bias-factor in Chapter 3. In addition, one can apply both *breaking-factor* following *adjusting-factor* to the mapping scheme. It presents a different layout as shown in Table 4.5c by expression $r = (d_s - (d_s \bmod t) + c) \bmod R$. In practice, one can explore more variations to meet particular requirements.

Similar to C-SCM, one can have C-SRM (*CRM based Super-Row Mapping*) as long as the number of *super-rows* is a coprime with the number of columns. The detail of C-SRM is not presented and it can be applied straightforwardly following C-SCM. Table 4.6 shows an example layout with C-SRM assuming $R = 6, C = 8$. Addresses with same *super-address* now map to the same *super-row*. For example, addresses 32 and 33 are mapped to *super-row* 1 as their *super-address* are both 16. This mapping scheme can be one solution for the problem presented

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

Table 4.6: An example layout of C-SRM with $R = 6, C = 8$.

in Section 4.3.

Case II: $(g = gcd(R, C) > 1)$ **and** $(gcd(R, C/g) > 1)$**.** In this case, the number of rows and the number of *super-columns* are non-coprime. The proposed C-SCM can not be applied. If $gcd(R/g, C) = 1$, C-SRM is feasible for this case. However, as shown in Table 4.6, it is challenging to maintain the row buffer locality. In addition, it does not work if $gcd(R/g, C) > 1$. We therefore propose a new scheme called C-SGM (*CRM based Super-Group Mapping*) which works even if $gcd(R/g, C) > 1$. The mapping is done with the following formula, in which $s = R \gg |g|, t = C \gg |g|$; and $d_s = d \gg (|g| \ll 1), d_s \in [0, s \cdot t - 1]$.

$$\begin{aligned} r &= (d_s \bmod s) \ll |g| + ((d \gg |g|) \bmod g) \\ c &= (d_s \bmod t) \ll |g| + (d \bmod g) \end{aligned} \tag{4.2}$$

The rationality of this scheme is that it groups multiple rows and columns separately into *super-row* and *super-column*, denoted as s and t, respectively. As s and t are coprime, the mapping from d_s to $L_{s \times t}$ can apply CRM. The second step in Formula 4.2 is to expand the *super-row* and *super-column* to normal rows and columns by the scaling-factor g and map addresses to the inner-group array $L_{g \times g}$. This is obtained by recursively applying the proposed C-SCM by grouping multiple columns as a *super-column*. In other words, it maps address $d \in [0, g \cdot g - 1]$ to $L_{g \times g}$ by C-SCM. With the two steps decomposition, address d is mapped to the 2D array one to one.

Table 4.7 shows an example layout of block addresses with the proposed C-SGM assuming $R = 6, C = 8$. In the mapping, $g = 2$ and $|g| = 1$. Thus, two rows are grouped as a *super-row* and two columns are grouped as a *super-column*. The addresses d's with same d_s are

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

Table 4.7: An example layout of C-SGM scheme with $R = 6, C = 8$.

mapped to same *super-row* and *super-column* index. For example, *super-address* d_s of addresses $d = 4, 5, 6, 7$ is 1, therefore all these addresses map to *super-row* 1 *super-column* 1. In the second step of proposed C-SGM scheme, we apply C-SCM to map to the inner *super-group* mapping and C-SRM can also be used for particular requirement.

The proposed C-SGM scheme can map address $d \in [0, 47]$ to designated region of $L_{6\times8}$. However, it does not maintain the property of *access-parallelism* or *access-locality*. One can explore XOR mapping [119] or adjustment factors with C-SGM. In this study, though, we explore a new scheme called S-SRM (*Section based Shift-Row Mapping*), which is not based on CRM. We divide the address space into sections based on the GCD g, in which $g = gcd(R, C)$. S-SRM can be written in the following formula:

$$\begin{aligned} c &= d \bmod C \\ r &= (d + i) \bmod R \end{aligned} \tag{4.3}$$

In above formula, $i = \lfloor (d \cdot g)/(R \cdot C) \rfloor$ and $i \in [0, g - 1]$. Table 4.8a shows an example layout of S-SRM. It is a one to one mapping that continuous addresses are mapped to different rows. Therefore, *access-parallelism* is maintained and this scheme can be a solution to the problem presented in Section 4.3. In case that *access-locality* is required, *adjusting-factor* can be applied to draw back row buffer locality. With *adjusting-factor* "$-c$", r is updated to $r = (d + i - c) \bmod R$. The adjusted layout is shown in Table 4.8b.

The rationality of S-SRM relies on the introduced i, which is called *shifting-factor*. It plays two major roles. First, it partitions the address space d into g sections, denoted as section i each. In above example, $g = 2$ and d is thus partitioned into two sections with $d_0 \in [0, 23]$ and $d_1 \in [24, 47]$, the subscript of d denotes i. Second, the *shifting-factor* i shifts the mapping of the

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

(a) An example layout of S-SRM without *adjusting-factor*.

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) An example layout of S-SRM with *adjusting-factor* "$-c$".

Table 4.8: Example layouts comparison using S-SRM with and without *adjusting-factor* assuming $R = 6, C = 8$.

entire section by an offset of i, which otherwise is mapped to the same positions with section of $i = 0$. For example, these two example sections will map to exactly the same row and same column in the 2D array without *shifting-factor*. Table 4.9 shows such layouts. The unmapped positions are marked as "x". It is obvious that address sets $d + i \cdot RC/g$ are mapped to the same positions for different integer $i \in [0, g-1]$. For example, addresses $d = 0, 24$ are mapped to $\langle 0, 0\rangle$; $d = 13, 37$ are mapped to $\langle 1, 5\rangle$. We thus come to S-SRM scheme to shift the row indexes of different address sets by its *shifting-factor*. With this adjustment, the translation maintains a one-to-one mapping.

We prove here that S-SRM is a one-to-one mapping. Given $R = mg, C = ng, gcd(m, n) = 1, d \in [0, mng - 1]$, formula

$$x = d \bmod R, y = d \bmod C$$

maps d to a pair of integers $\langle x, y\rangle$ one to one. To prove the statement, Let $d_p = pR + x_0$ and p is the integer quotient, therefore, $d_p \bmod R \equiv x_0$. Since $d \in [0, mng - 1]$, $p \leq n - 1$. Let $y_p = d_p \bmod C$. Now, we merely need to prove that for all p, d_p maps to different y_p. Assume

row/col	0	1	2	3	4	5	6	7
0	0	x	18	x	12	x	6	x
1	x	1	x	19	x	13	x	7
2	8	x	2	x	20	x	14	x
3	x	9	x	3	x	21	x	15
4	16	x	10	x	4	x	22	x
5	x	17	x	11	x	5	x	23

(a) An example layout of S-SRM with $d_0 \in [0, 23], i = 0$.

row/col	0	1	2	3	4	5	6	7
0	24	x	42	x	36	x	30	x
1	x	25	x	43	x	37	x	31
2	32	x	26	x	44	x	38	x
3	x	33	x	27	x	45	x	39
4	40	x	34	x	28	x	46	x
5	x	41	x	35	x	29	x	47

(b) An example layout of S-SRM with $d_1 \in [24, 47], i = 1$.

Table 4.9: Example layouts with S-SRM without applying *shifting-factor* i for the two sections of address $d \in [0, 47]$ assuming $R = 6, C = 8$.

there is a $p\prime$ different from p such that $y_{p\prime} = y_p$. Then

$$y_p = pmg + x_0 - qng$$

$$y_{p\prime} = p\prime mg + x_0 - q\prime ng$$

Since $y_{p\prime} = y_p$, we have

$$(p - p\prime)m = (q - q\prime)n$$

Note that $p, p\prime, q, q\prime$ are all integers and $p, p\prime \in [0, n-1]; q, q\prime \in [0, m-1]$, m and n are coprime. The only condition that the above formula is right is $p = p\prime$ and $q = q\prime$. Therefore, the assumption is invalid. We thus have the statement proved.

The proposed S-SRM can have multiple variations. One can shift column to adjust the layout instead of shifting row to have S-SCM (*Section based Shift-Column Mapping*). In addition, one can combine S-SRM with C-SCM to group multiple columns as a super-column and then apply S-SRM. Formula 4.4 shows an example equation combining C-SCM and S-SRM together. In the formula, $d_s = d \gg |g|, t = C \gg |g|$ which are *super-address* and *super-column*,

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

Table 4.10: An example layout with combination of S-SRM and C-SCM, assuming $R = 6, C = 8$.

respectively. Table 4.10 shows an example layout using this formula.

$$\begin{aligned} c &= (d_s \bmod t) \ll |g| + (d \bmod g) \\ r &= (d_s + i) \bmod R \end{aligned} \tag{4.4}$$

The proposed S-SRM can be adjusted to maintain row buffer locality. However, S-SRM requires complex multiplication and division operations to calculate the *shifting-factor* i. This can be resolved in real machine by adding a small RAM or ROM to maintain the address range for each i given that g is generally a small value. In above example, $g = 2$, value $B = 24$ can be maintained such that $i = 0$ for $d < B$ and $i = 1$ for $d \geq B$.

4.4.3 Case Study of Real DDR3 System With SEP

In previous section, we discuss all the address mapping schemes for different cases mathematically. In this section, we show case studies of applying the proposed mapping schemes in real DDRx memory system for selective protection.

Without losing generality, we assume an example memory system in our following discussions. The memory system has two DRAM channels with two DIMMs per channel and two ranks per DIMM. Each rank contains eight x8 DRAM devices, assuming Micron DDR3-1600 MT41J256M8 device [69] in this example. It is a 2Gbit chip with eight internal banks and each bank has 32K rows, 1K columns, and 8 bits per column. Assume SEP divides the space into two regions with ratio 3 : 1 and let the upper region, denoted as x, have $2^{15} \times \frac{3}{4}$ rows and the lower region, denoted as y, have $2^{15} \times \frac{1}{4}$ rows. In a case that region x is not protected by ECC

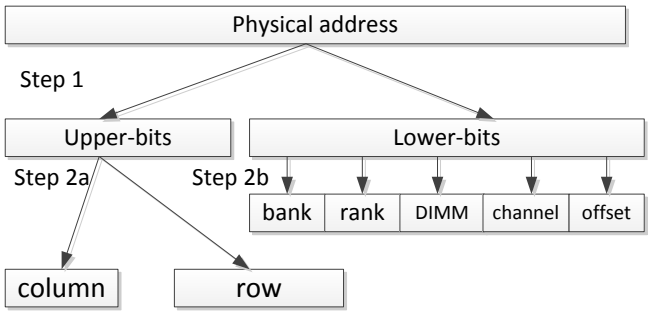


Figure 4.3: Address decomposition procedure.

and it requires *access-parallelism*, conventional cacheline-interleaving scheme cannot be applied to resolve its address mapping. In this case[4],

$$R_x = 2^{13} \times 3, C_x = 2^7, gcd(R_x, C_x) = C_x$$

Therefore, Formula 4.1 can be used. By grouping all the columns as a *super-column*, CRM based mapping scheme is applicable. Figure 4.3 shows the detailed address decomposition procedure. In step 1, the physical address can be decomposed directly into lower 12 bits and the remaining upper-bits. The lower-bits include bank, rank, DIMM, channel and block-offset. In step 2b, these lower-bits are split into corresponding indexes as all of them are power-of-two. Step 2a is completed by Formula 4.1 that $r = (d \gg 7) \bmod R_x, c = d \bmod C_x$. To maintain *access-parallelism*, a breaking-factor "$+c$" can be added in expression r, which is explained previously. The above formula requires modulo $2^{13} \times 3$ and modulo 2^n operations. Modulo 2^n is simply the last n bits of address d and we prove in Section 4.4.4 that modulo $2^{13} \times m$ can be converted to modulo m operation, which can be done efficiently.

In above case, if region x requires ECC protection, its number of columns is shrunk to a non-power-of-two value. As each 64-byte data block requires 8 bytes ECC with SECDED scheme, the effective number of columns is thus $C_x = 2^{10}/9 = 113$. A modulo operation of 113 can introduce certain complexity in hardware implementation in practice. We thus use merely 112 columns as a trade-off between complexity and capacity. Therefore,

$$R_x = 2^{13} \times 3, C_x = 2^4 \times 7, gcd(R_x, C_x) = 2^4$$

[4]In a 64-bit rank, eight columns are accessed continuously to burst a 64-byte block. C_x is thus 2^7 by removing 3 bits block offsets for a typical 64-byte block.

Similarly, the proposed C-SCM scheme can be applied and proper adjusting-factor can be added if needed. In this example, S-SRM can also be applied for address mapping. Given $gcd(R_x, C_x) = 2^4$, 15 reference addresses are required to divide the physical address space into 16 sections. By applying Formula 4.3, the mapping is done by modulo R_x and C_x.

The SEP design can introduce various combinations of R and C as the partitioning can be flexible and diverse forms of ECCs can be employed other than SECDED code. The proposed mapping schemes can be applied correspondingly. However, the flexibility may be limited slightly in practice considering the complexity of hardware implementation. One may also make trade-off between valid DRAM capacity and hardware simplicity.

4.4.4 Hardware Implementation of Modulo Operation

In real DDRx memory system with the proposed address mapping, the following modulo operation is generally required:

$$v \bmod (m \times 2^n)$$

where m is a small value. We prove that the above modulo $(m \times 2^n)$ operation can deduce to modulo (m) operation, which can be resolved efficiently. The deduction is obtained by the following formula:

$$v \bmod (m \times 2^n) = ((v \gg n) \bmod m) \ll n + (v \;\&\; (2^n - 1))$$

where & is a bit-wise AND operation. The core part of the proof is to regard the data in binary representation. Assume $v = h \ll n + l$, where l is the lower n bits of v and h is the higher bits. We thus have $l < 2^n$. Let $h = mk + r$, where $r = h \bmod m$. Given the two basic properties of modulo operation [101]

$$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$

$$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$$

The proof is done by following steps:

$$
\begin{aligned}
& v \bmod (m \times 2^n) \\
&= ((h \cdot 2^n) \bmod (m \cdot 2^n) + l \bmod (m \cdot 2^n)) \bmod (m \cdot 2^n) \\
&= (((mk + r) \cdot 2^n) \bmod (m \cdot 2^n) + l) \bmod (m \cdot 2^n) \\
&= (r \cdot 2^n + l) \bmod (m \cdot 2^n) \\
&= r \cdot 2^n + l
\end{aligned}
$$

Based on the basic properties of modulo operation, a modulo small number operation can be done as follows:

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

where v_i is the corresponding ith bit of v in binary representation. When m is set, the formula can be further simplified as $(2^i \bmod m)$ has finite outcomes. For example, Teng [101] shows that a simple logic can be implemented to resolve modulo 7 operation as $2^i \bmod 7 = 1, 2, 4, 1, 2, 4 \cdots, (i = 0, 1, 2, 3, 4, 5 \cdots)$. Therefore, a modulo m operation is deduced to addition operation. The corresponding ith bit of v is used as a filter to decide whether one of the three outcomes is added to obtain the result. Similar to modulo 7 operation, one can have repeated outcomes for other small numbers for 2^i. Therefore, the modulo operation can be done efficiently. In practice, one can add a small SRAM in memory controller for maintaining these remainders for a preset value m. The SRAM can be reconfigured based on the specified reference address in partitioning register.

4.4.5 Other Discussions

The proposed address mapping schemes can be extended for memory systems with more regions for various error protections. For example, a system can have three regions with one unprotected, one with SECDED protection and the remaining one 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.

4.5 Discussion of Application Scenarios

The proposed address mapping schemes make SEP a viable design in real DDRx systems. All these schemes can be applied but not limited to the following scenarios.

4.5.1 OS and Compiler Aided Selective Protection

There can be multiple ways to apply SEP in real machines. First, Mehrara et al. [68] propose a profile-based criticality analysis to determine the vulnerability of data, text, variables, etc. of a program. The liveness based profile study prioritizes code and data that are frequently accessed and places them in error-protected memory region. Second, Chen et al. [12] use a program slicing [106] based approach to optimize compiler to place the set of elements requiring protection in error tolerant region of SEP. Furthermore, Oz et al. [77, 76] propose TVF (Thread Vulnerability Factor) to quantify vulnerability of multithreaded applications and further propose core partitioning based on TVF considering both performance and reliability. It hints a core partitioning based policy with SEP hardware design. In addition, one can even apply user-specified data criticality to tell OS and compiler which part of data requires protection and SEP facilitates this purpose. All these strategies require OS and compiler for placing the sensitive data. The details are beyond our study.

4.5.2 Selective Protection to Lower Refresh Frequency

There have been multiple studies [62, 61, 107, 19, 48] focusing on reducing refresh frequency of DRAM memories to improve power efficiency and performance. Refresh command is essential to DRAM technologies to recharge memory cells in certain period to maintain data integrity. Otherwise, data loses as charges on a cell leak away. As DRAM capacity and density are growing, more refresh operations are required for each DRAM bank and they can introduce significant overhead in terms of performance and power consumption. Studies [107, 19, 48] have proposed to add ECC to tolerate errors in DRAM cache or main memory caused by reduced refresh rate. In detail, study [19] observes leakage errors are unidirection from 1s to 0s and therefore *Berger Code* can be applied to detect arbitrary number of errors (no correction is

possible). It further shows that DRAM refresh rate can be reduced to 4x time by applying single-bit error correction. With all these schemes, our proposed address mapping can be applied to organize and address ECC codeword. The implementation, though, requires careful design as SEP itself introduces overhead and error tolerant capability is limited.

4.5.3 Selective Protection to High Error Rate Region

Recent studies [92, 33] measure error rate and study error properties on DRAM systems on large scale real machines. They both observe that errors are not distributed uniformly on memory system. Particularly, study [33] observes that the top and bottom of the row/column space in a bank is more likely to experience errors. In addition, errors tend to cluster on same row, same column and their nearby rows and columns. That means the error rate of the rows and columns is high once they have experienced errors. All these observations hint for SEP scheme that selective error protection can be applied to these regions to tolerate errors while limit area and power cost by reliability design. As the regions with high error rate can physically distributed in a bank, it is challenging to apply SEP with the proposed address mapping straightforwardly. One can propose to add a simple logic to reorganize the physically scattered rows to be logically continuous. Therefore, the memory space can be divided into two continuous regions logically and our proposed address mapping schemes can be simply applied.

4.5.4 Balancing DRAM Access Locality and Parallelism

In conventional DDRx memory system, either cacheline-interleaving or page-interleaving scheme is deployed and the system either maintains *access-parallelism* or *access-locality*. Due to diversities of applications and workloads, such a system can be inefficient since merely one property is maintained. Study [44] proposes to partition internal memory banks between cores to maintain DRAM accesses locality since requests interferences from other cores are isolated. In addition, they propose to compensate the reduced bank parallelism with sub-ranking scheme. Similar to their study, one can have different partitions for both locality and parallelism in one system. For example, a low-level in-bank partitioning can be applied, in which requests with high locality are placed in same DRAM rows targeting high row buffer hit ratio while requests

with low locality are distributed into multiple banks for high parallelism. Such a design requires memory requests characterization of different workloads. As in-bank partitioning can form a non-power-of-two region, our proposed scheme can be applied to resolve the address mapping challenge.

4.6 Summary

In this chapter, we discuss in detail the DRAM device-level address mapping challenges when selective error protection scheme is applied. To resolve the challenges, the CRM based and Section based address mapping schemes are proposed. The proposed mapping schemes are flexible and can be selectively applied to either maintain high memory locality or high accessing parallelism by tuning the proposed adjustment factors. All these schemes are based on modulo operation, which is proved to be efficient. The proposed schemes can be applied to diverse scenarios balancing power and area overhead with reliability requirement. They are efficient, flexible and feasible in practice.

CHAPTER 5. FREE ECC: EFFICIENT ECC DESIGN FOR COMPRESSED LLC

Cache compression schemes have been proposed to increase the effective cache capacity of last-level cache, for which we found the conventional cache ECC design is inefficient. In this chapter, we propose *Free ECC* that utilizes the unused fragments in compressed cache design to store ECC. It not only reduces the chip overhead but also improves cache utilization and power efficiency. Additionally, we propose an efficient convergent cache allocation scheme to organize the compressed data blocks more effectively than existing schemes. Our evaluation using SPEC CPU2006 and PARSEC benchmarks shows that the *Free ECC* design improves cache capacity utilization and power efficiency significantly, with negligible overhead on overall performance. This new design makes compressed cache an increasingly viable choice for processors with requirements of high reliability.

5.1 Introduction

Recently, efficient compression techniques have been proposed [17, 109, 3, 80] for on-chip caches to increase the effective capacity without physically enlarging the storage. Typically, compressed cache doubles the number of tag fields and merely targets a 2x cache capacity. This parameter is selected mainly due to the limitation of average compression ratio viable, while avoiding unnecessary tag arrays on chip. Therefore, all of our following discussions use this 2x compressed cache as the baseline. Compression techniques save the real estate of data array when being compared with conventional method, which increases cache capacity by brute force. Thus, the number of on-chip transistors is effectively reduced and so is the leakage power.

However, the conventional cache ECC design is inefficient when used in compressed caches.

We formulate this efficiency as cache capacity utilization using the following formula:

$$e = \frac{\text{Combined Size of Effective Data Blocks}}{\text{Physical Cache Capacity}}$$

Without error protection, this factor is 2 in maximum for a compressed cache that targets 2x compression [109, 80]. The maximum utilization reduces to 1.78 if conventional ECC design is used. We have observed that, in recently proposed compressed cache schemes, a compressed cache has a significant amount of unused fragments in the cache data blocks as the result of compression. If done properly, those fragments can be utilized to store ECC. It not only reduces chip cost but also improves cache capacity utilization and power efficiency. It will not be straightforward, though, because the unused fragments are of variable sizes and they may not always be available for a given cache block.

In this chapter, we propose a design called *Free ECC* to embed ECC into the unused fragments in compressed caches to avoid dedicated ECC storage. The design is tightly coupled with the cache compression scheme. For a given cache block, there are three cases of embedding its ECC, which is distinguished by a compression encoding field of the block (the field exists in compressed cache and uniquely identifies a compression pattern). First, the ECC can be stored directly in the unused fragment if the fragment is large enough to hold the ECC, which is a common case. Second, if the fragment is not large enough, a lightweight EDC (Error Detecting Code) is embedded into the block, and the ECC is stored in a reserved block of the same cache set of the cache block. For a cache read, the ECC is accessed only if an error is detected by EDC checking, which is a rare case. In other words, almost all cache reads only incur accessing a single cache block, even if the ECC is stored in another block. The remaining space of the reserved block will be still used for data storage. We have revised the compressed cache scheme such that, for any compression pattern, if a cache block is compressed, the unused fragment is always large enough to embed the EDC. Finally, for a cache block that cannot be compressed, its EDC is maintained in the compression encoding field, which is not used when the block stores uncompressed data.

We have carefully examined the technical issues and challenges in *Free ECC* design and have resolved those issues with minimal impact on system performance. We also propose a

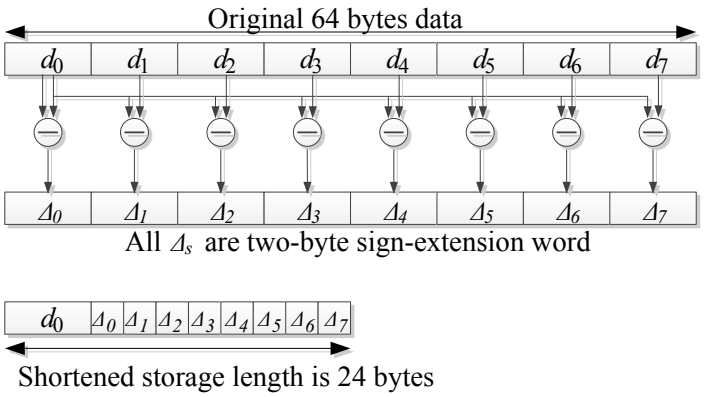


Figure 5.1: An example of $B\Delta I$ ($B_8\Delta_2$) compression algorithm, in which base size is eight bytes and each Δ is two bytes.

convergent cache allocation scheme to efficiently organize compressed data blocks in a set. It provides improved compression ratio or reduced complexity when compared with two existing cache compression schemes.

The rest of this chapter is organized as follows: Section 5.2 introduces the background of cache compression schemes and related work. *Free ECC* Cache design is presented in Section 5.3. Section 5.4 and Section 5.5 present simulation environment and analyze the simulation results, respectively. Section 5.6 concludes this chapter.

5.2 Background and Related Work

5.2.1 Cache Compression Schemes

Compression algorithms have been widely studied [80, 109, 3, 122, 123, 18] for decades. However, only those with simple and fast hardware implementations can be used in cache compression. Among existing compression algorithms, ZCA (Zero-Content Augmented Cache) [17], FVC (Frequent Value Compression) [109], FPC (Frequent Pattern Compression) [3] and recently proposed $B\Delta I$ (Base-Delta-Immediate compression) [80] are potential selections for cache compression due to the hardware simplicity. ZCA [17] focuses on reducing the storage of zero value in cache as it is frequent. FVC [109] based compression observes that a certain number of values are frequent and could be represented by smaller length of encoding bits. FPC [3] is proposed to compress data with certain patterns like zero runs, sign-extension, half-word and

others. It is built on the observation that some data patterns are frequent and representable by fewer bits.

In *Free ECC* design, we opt for $B\Delta I$ as the baseline compression algorithm due to its competitive compression ratio and low decompression latency. However, all the discussions can be extended to most of the other algorithms. $B\Delta I$ [80] explores the similarity of nearby data segments in a data block. It represents data with a base (B) and a set of differences (Δs) between data segments and the selected base. Figure 5.1 presents an example of the compression scheme. In the example, the 64-byte data is represented in eight 8-byte segments $\{d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$. d_0 is selected as the base and differences are calculated as $\Delta_i = d_i - d_0$ for $i = 0, 1, \cdots, 7$. As all the Δ_is ($i = 0, 1, \cdots, 7$) are two-byte sign extension words, the Δs can be represented in two bytes and the 64-byte data is thus compressed to $\{d_0, \Delta_0, \Delta_1, \cdots, \Delta_7\}$.

As B and Δ sizes vary, the compressed data is thus represented as $B_x\Delta_y[I]$, meaning the base size is x-byte and each Δ is y-byte. An optional I determines whether zero is selected as the second base, in which case a data mask is required to indicate whether Δ is based on the non-zero or zero base. The compressed data size thus can be calculated by $x + y \cdot (64/x) + I \cdot (64/x)/8$, I is 1 if zero is opted as a second base, 0 otherwise. In addition, a four-bit encoding is required to decompress the 15 compression patterns, including uncompressed data, as a combination of x, y and I.

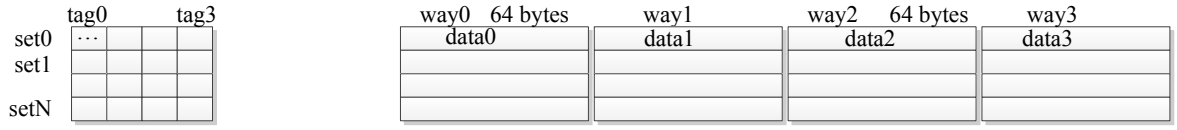
5.2.2 Fragments In Compressed Cache

The cache organization maintains a tag field for each data block as an identifier. Therefore, the number of tags determines the maximum data blocks that can be placed in the data entries. As the compression ratio for each application varies, it is highly possible that small storage fragments are left idle in compressed cache. Take $B\Delta I$ compression algorithm as an example, if the two compressed data lengths are 16 and 36 in one data block, a 12-byte fragment is left unused.

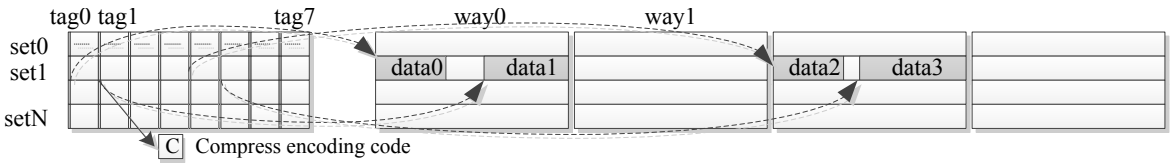
5.2.3 Related Work

Prior works [50, 118, 51, 112, 111] have studied cache error protection for uncompressed caches. Kim and Somani [50] propose to only protect the most frequently accessed data as they would propagate errors to other components more easily. The proposed parity caching, shadow checking and selective checking significantly reduce the cost required for the reliability concern. However, errors in the less frequently used data would potentially tamper other components. ICR (In-cache replication) [118] proposes a non-uniform reliability scheme to use the existing cache space to hold replicas of data that would be used in near future. It reduces the area cost significantly. However, the system potentially suffers high error rate since not all cache lines are protected. Kim [51] proposes another area efficient scheme that applies ECC to dirty blocks and EDC to clean blocks and it periodically evicts dirty blocks to DRAM to reduce the amount of ECC required in cache. However, this scheme increases the traffic to main memory.

The most recent studies propose to decouple error detection and error correction to minimize the overhead in tolerating errors [112, 111]. They propose to keep the light-weight EDC in on-chip cache while offloading the high-cost ECC code to remote off-chip DRAMs given the observation that error correction is a rare event. Such a separation of EDC and ECC significantly reduces the cost of the dedicated storage for redundancy but does not eliminate it, and it introduces the overhead for updating and retrieving ECC from/to DRAMs. In the case that a dirty cache line is updated and its ECC is not cached, a DRAM accessing is required to update the corresponding ECC. A continuous writing to the same dirty line would worsen the situation. Although they further propose ECC FIFO [111] to reduce overhead for accessing main memory for ECC, the scheme will still potentially increase main memory traffic and its power consumption. There are also many other studies [58, 117, 116] proposed to reduce the overhead of reliable cache designs. However, they are not targeting the specific compressed cache schemes and do not explore the fragments in compressed caches.



(a) A conventional uncompressed 4-way set associative cache organization



(b) A $B\Delta I$ compressed cache organization with double tag fields to maintain 2x data in data storage

Figure 5.2: Comparison of uncompressed cache and compressed cache organizations.

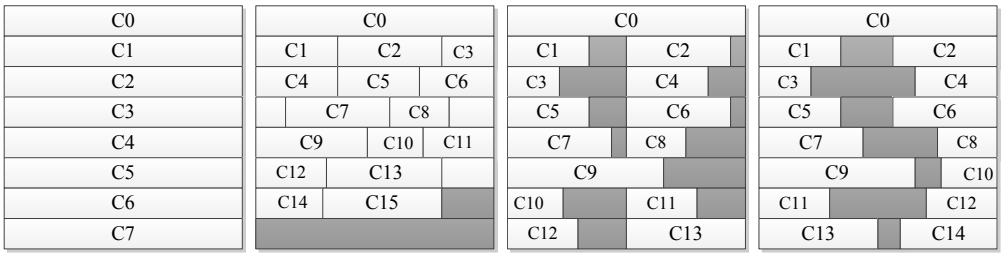
5.3 Design of Free ECC

In this section, we first present the compressed cache organization and the proposed convergent allocation scheme. Based on this layout, we discuss in detail the challenges and strategies adopted in *Free ECC* cache implementation.

5.3.1 Convergent Allocation Scheme

Figure 5.2 compares the compressed cache organization with its uncompressed form. Figure 5.2a presents the uncompressed four-way set associative cache. Compared with the uncompressed form, Figure 5.2b doubles the number of tag fields to allow two data blocks to reside in one cache block. Applying $B\Delta I$ compression algorithm, a 4-bit encoding C is attached to each tag to help decode and address the two data blocks in one cache line as illustrated in Section 5.2. Compared with Figure 5.2a, Figure 5.2b virtually doubles the cache capacity without physically enlarging the data storage.

In compressed caches, the layout of data blocks affects compression ratio and thus the performance. Figure 5.3 compares the uncompressed and three compressed cache allocation schemes. Figure 5.3a presents the eight data blocks in one set for an eight-way set associative cache. Figure 5.3b presents an example of consecutive allocation layout, which allocates space for compressed data blocks continuously in order to efficiently use the storage. However, this scheme is impractical due to multiple issues. First, the compressed data can cross two cache



(a) uncompressed cache (b) consecutive allocation (c) fixed-space allocation (d) convergent allocation

Figure 5.3: Comparison of cache allocation schemes for an example eight-way set associative cache. The dark gray fields are fragments left unused.

blocks, i.e. data C6, C9 and C14. This non-alignment complicates cache read access. Second, as the length of compressed data varies, the address of a data block in one set changes dynamically, which complicates the addressing. The calculation of starting position of a data block requires the information of compressed lengths of all the prior data blocks. Most importantly, it introduces data movement issue caused by data overflow, also called ***fat write*** for the case that the coming data cannot fit in its prior slot. In such cases, all the following data requires to move backwards. These excessive overheads hamper its adoption in on-chip caches though such allocation scheme uses cache capacity efficiently.

Conventionally, fixed-space allocation is applied to organize the data placement. An example layout is shown in Figure 5.3c. In such a scheme, every 64-byte block is regarded as two 32-byte storage. If compressed data can fit into a 32-byte block, it uses half of the entry. Otherwise, it uses the entire 64-byte cache block. For example, compressed data blocks C3 and C4 are placed in one entry as their sizes are both less than 32 bytes. However, data C9 is greater than 32 bytes and thus occupies the entire cache block. In the case of a fat write, the data block that shares the same cache block with the coming data is simply evicted. It thus eliminates the data movement issue and simplifies addressing data blocks. Such a simple scheme addresses the issues in consecutive allocation with penalty of compression ratio loss.

We propose convergent cache allocation to improve the compression ratio of fixed-space allocation. Figure 5.3d presents an example data layout. It allows two compressed data to place in one entry if the total size fits into one block. The two compressed data are placed

from the two ends of the entry and leave the fragment in the middle. It has the following advantages. First, it increases the compression ratio compared with fixed-space allocation. For example, data C9 and C10 are placed in one entry in Figure 5.3d while they reside in two blocks in Figure 5.3c. Second, it shares all the advantages of fixed-space allocation, i.e. eliminating data movement issue and simplifying data blocks addressing. In addition, the placement of two compressed data leaves one big fragment in the middle instead of two smaller ones in fixed-space allocation, which helps maintain ECC.

The details of cache access for convergent allocation are similar to that of fixed-space cache allocation. In fixed-space cache allocation, the start position of the second compressed data is always fixed to the half point of block size. It ends at multiple varied positions depending on compressed data lengths. For convergent cache allocation, the start position of second data in a block varies but it always ends at the last byte of the block. With $B\Delta I$ compression algorithm, the start position of the second compressed data is the two's complement of the corresponding compressed data length. For example, given a compressed data length of 16, the start position is 110000_{bin} which is two's complement of the corresponding compressed data length 010000_{bin}.

5.3.2 Free ECC Design

Based on the proposed convergent allocation scheme, we first examine a straightforward implementation of embedding ECC into fragments. In the straightforward design, an ECC code word is maintained for all the data blocks resided in the cache block entry. In the case that a cache block is short of space for ECC code word, the overflowed ECC is maintained in any other fragment in that cache set.

Such a design introduces several issues. Taking Figure 5.2b as an example, when a write request updates data 0 in set 1 in the figure, it incurs an extra read operation to retrieve data 1 for updating ECC. Second, ECC may cause overflow in compressed caches. For example, if there is no sufficient space to maintain ECC in way 2 set 1 in Figure 5.2b, its ECC needs to store in another place. This introduces overhead of pointers for locating ECC and requires extra read operation for retrieving ECC. Moreover, write overflow induced ECC movement issue. Assume the ECC code word ECC 2 for way 2 is placed in way 0 as way 2 lacks space in

Group	Methods	Encoding	l_{cps}	l_{opt}	Form A $(x+z)$	Form B $(x+y+z)$
small	Zero-run	0000	0	0	$0+0$	$0+0+0$
	Repeat	0001	8	8	$8+1$	$8+0+1$
	$B_8\Delta_1$	0010	16	15	$15+2$	$15+0+2$
	$B_8\Delta_1 I$	0011	17	16	$16+2$	$16+0+2$
	$B_4\Delta_1$	0100	20	19	$19+3$	$19+0+3$
medium	$B_4\Delta_1 I$	0101	22	21	$21+3$	$21+1+0$
	$B_8\Delta_2$	0110	24	22	$22+3$	$22+1+0$
	$B_8\Delta_2 I$	0111	25	23	$23+3$	$23+1+0$
large	$B_2\Delta_1$	1000	34	33	$33+5$	$33+1+1$
	$B_4\Delta_2$	1001	36	34	$34+5$	$34+1+1$
	$B_4\Delta_2 I$	1010	38	36	$36+5$	$36+1+1$
	$B_2\Delta_1 I$	1011	38	37	$37+5$	$37+1+1$
	$B_8\Delta_4$	1100	40	36	$36+5$	$36+1+1$
	$B_8\Delta_4 I$	1101	41	37	$37+5$	$37+1+1$
Uncompressed		1111	64	64	$64+0$	$64+1+0$

Table 5.1: Tailored $B\Delta I$ algorithm with ECC/EDC integrated. l_{cps} and l_{opt} are original and optimized compressed data lengths, respectively. Columns six and seven show the two possible store forms of the compressed pattern. x: compressed data length; y: EDC bytes stored with the data, z: ECC bytes stored with the data.

Figure 5.2b. In the case that data 0 overflows, ECC 2 would move to another space as fragment in way 0 runs out. This causes extra read and write operations.

Objectives in Designing *Free ECC*. The design of *Free ECC* is to resolve those challenges and make it simple and effective. We thus have the following design objectives:

- Avoiding the extra read operation when updating ECC.
- Avoiding the second read operation to fetch ECC.
- Eliminating the ECC movement issue induced by fat write.
- Minimizing the impact to compression ratio when ECC is integrated.

In order to meet our design goals, we first optimize the $B\Delta I$ compression algorithm and then propose three strategies to simplify *Free ECC* design.

Optimization of $B\Delta I$. In $B\Delta I$ compression algorithm, the first non-zero based Δ is always zero because the first non-zero data segment is selected as the base. Therefore, the first Δ can

be removed to shorten the compressed data length. The detailed optimization is described in Column five (l_{opt}) in Table 5.1. The optimization guarantees that there is at least a four-byte fragment left unused for any two compressed data blocks maintained in one cache entry. In the worst case, the two compressed data block lengths are 23 and 37.

Strategy to separate ECC for each data block. In order to resolve the first challenge, we propose to separate ECC for each data block instead of applying ECC for the entire 64-byte cache block. Using conventional SECDED (72,64) coding, we may need to pad zeros for a data block to make it eight-byte aligned. For example, 5-byte zeros are appended following a 19-byte data block for calculating its 3-byte SECDED parity word. Therefore, no extra ECC generating or checking logic is required for varying lengths of compressed data.

Strategy for ECC placement. In order to avoid introducing high cost of ECC pointers and resolve ECC movement issue, we propose the following policies. First, at least one data block's ECC is always maintained in the same cache block if two compressed data reside in one cache line. Therefore, only one ECC pointer is required for each cache block at most. In the particular case that a 64-byte uncompressed data and an all-zero data block are placed together in one entry, it still solely requires one ECC pointer. In *Free ECC* design, we thus add one ECC pointer E for each cache block and an extra bit D to indicate which block's ECC is maintained in another cache block. This strategy introduces minimal impact to compression ratio as we observe that for any two compressed data that can reside in one block, there is at least four-byte fragment left in $B\Delta I$ algorithm.

Second, the overflowed ECC is stored in the last cache block in that set instead of any arbitrary fragment in cache to avoid ECC movement issue and simplify ECC locating. The reserved ECC space is regarded as 16 four-byte segments, which means E is four bits. In the worst case that all data are uncompressed, 56 bytes of storage is reserved in the last entry for ECC for an eight-way set associative cache. We therefore leave the first eight bytes in the last cache block for data and $E = 0000$ is an invalid ECC pointer, which means all the ECCs are maintained in the same block for that cache line.

Strategy for data protection. In the case that there is no sufficient space to maintain ECC in the same block, an extra cache access is required to fetch its ECC. In order to avoid the

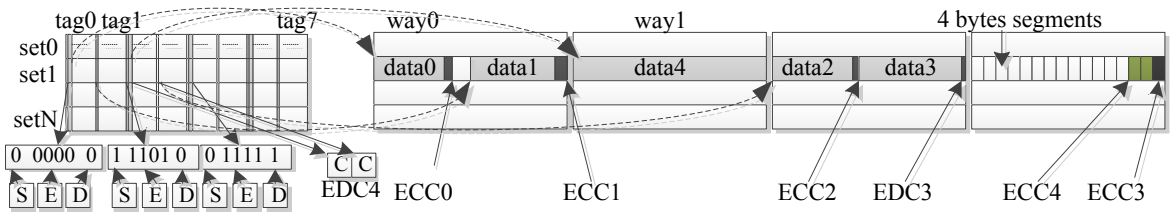


Figure 5.4: *Free ECC* cache organization.

second access, we bind one byte EDC for such data block. The computing of EDC is similar to that of ECC such that the data is aligned by padding zeros. With the adoption of this strategy, the second cache access is only required if an error is detected, which is a rare event. The integrating of EDC introduces negligible impact on compression ratio as we observe that there is one byte space left in almost all cases if the data is compressible.

For the particular case that data is uncompressible, we place its one-byte EDC into the two 4-bit encoding C fields. The design is reasonable as in this case the encoding bits are unused. An extra special bit S is thus required to identify this case. If S is set, the two C fields store the one-byte EDC for an uncompressed data. Tag of the data is selected by checking D bit as no ECC is needed for the other possible data zero.

Optimization for compression ratio. In the above design, all data blocks in cache are protected by ECC. Such a scheme can introduce penalty to cache compression ratio. We thus propose an optimized protection scheme to reduce the cost. In the optimization, clean data blocks are not protected by ECC based on the observation that their duplications exist in DRAM system. In case that EDC detects an error, the dependable duplication can be retrieved from lower level main memory. Therefore, a clean data block merely requires one-byte EDC code and thus mitigates the costs introduced by *Free ECC* design. Such an optimization causes no extra complexity to hardware implementation. Once cache write logic recognizes a clean block, it simply calculates an EDC byte and writes it together with data. For uncompressed block, its EDC is still maintained in the two tag fields.

***Free ECC* Cache Organization.** Figure 5.4 presents *Free ECC* cache organization by applying all those optimization and strategies. It is similar to conventional compressed cache illustrated in Figure 5.2b with a slight extension of tag field. A four-bit ECC pointer E and

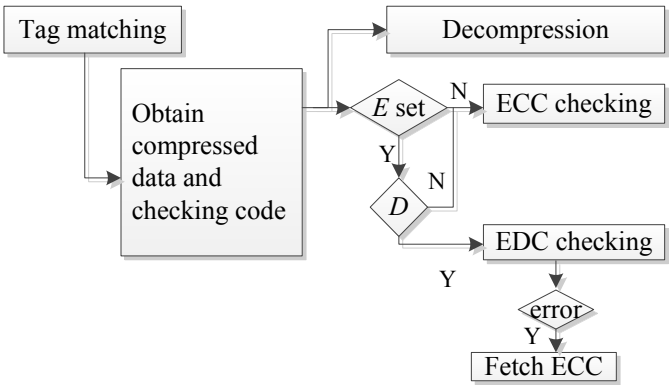


Figure 5.5: *Free ECC* cache read operation.

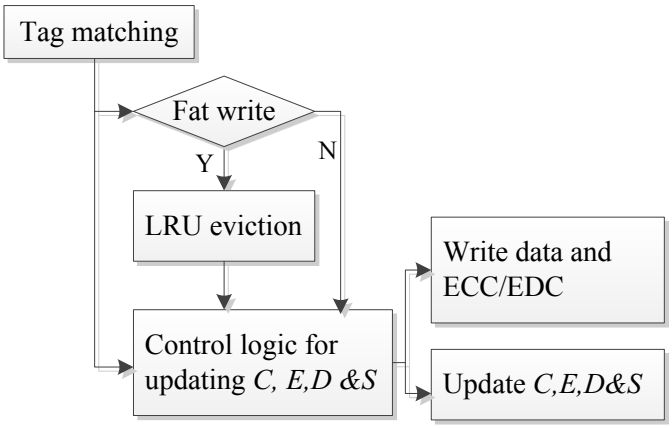


Figure 5.6: *Free ECC* cache write operation.

a one-bit D are appended to locate the overflowed ECC; and a one-bit S is added to tell if EDC is stored in the two encoding fields. As described in Figure 5.4, E and D fields for way 0 set 1 are 0000 and 0 in binary, respectively, which means ECC 0 and ECC 1 are stored in the same block with data 0 and data 1. As for way 1, its S bit is set to 1 and D is 0. This means an uncompressed data block is stored in way 1 and its tag is tag 2. If tag 3 is valid, its corresponding data is zero. For the uncompressed data, its EDC is maintained in the two encoding C fields and its ECC is stored at the 13th (1101 in binary) segment in the last cache block in that set. The S, E and D for way 2 are 0, 15 and 1, which indicates that EDC of data 3 is embedded together with the data and its ECC is stored in the 15th segment in the last cache block. Since D points to data 3, it implicates that ECC 2 for data 2 is maintained following data 2.

***Free ECC* Read Operation.** *Free ECC* cache read operation is similar to conventional compressed cache access as illustrated in Figure 5.5. After tag matching, the corresponding S and E bits are examined and D bit is checked. If E is unset, all the ECCs are stored following the data. Otherwise, the EDC of D pointed data resides in the block or in C field if S is set. Then one cache access is executed and either ECC or EDC and the compressed data are fetched. The data decompression and error detection and correction are operated in parallel. In a rare case that EDC detects an error but cannot correct it, a second cache access is required to fetch its ECC located by ECC pointer E. Given the fact that cache error is a rare event, (i.e. 10^{-3} FIT/bit), the case that two cache accesses are required for one cache read request is negligible.

***Free ECC* Write Operation.** Figure 5.6 illustrates the *Free ECC* cache write operation. When a tag matches, the compressed data is placed to the cache block. In the case of a fat write, LRU replacement is activated to select a proper position for the coming data. During the operation, an eviction may be necessary since there is no sufficient space to hold that data. The detailed replacement policy adopted is presented in Algorithm 1 in Section 5.5. If a position is selected for the coming data, its ECC or EDC is selectively written into the same entry based on the compressed data length and ECC pointer E.

We divide $B\Delta I$ encodings into three groups to simplify the design. Group G_{small} are data whose compressed lengths are less than 20 bytes and G_{large} are data with compressed lengths greater than 23 bytes. Group G_{medium} are those with lengths in between the two.

$$G_{small} : \{l_{opt} \mid l_{opt} \leq 19\}$$

$$G_{medium} : \{l_{opt} \mid 19 < l_{opt} \leq 23\}$$

$$G_{large} : \{l_{opt} \mid 23 < l_{opt} \leq 37\}$$

The partition of the compressed lengths into three group simplifies cache hardware implementation. In order to write the coming data block, the extended tag fields are fetched to check the current cache status. Table 5.2 presents the detailed combinations for writing the coming data blocks in a cache. If the two compressed data reside in one cache line and any of them is from G_{small} or no one is from G_{large}, both ECCs are placed in the same entry. If the two

Existing data form		Coming data	Write form
small	ECC	X	data + ECC
X	X	small	data + ECC
X	EDC	X	data + ECC
medium	ECC	medium	data + ECC
		large	data + EDC + 1-byte ECC
large	ECC	medium	data + EDC

Table 5.2: Hardware implementation truth table. X means any case. The existing and coming data are the two possible data in one cache block in a 2x compressed cache.

compressed data are from G_{medium} and G_{large}, one data's ECC is placed in the reserved last cache block depending on E bits. In the case that five-byte ECC is required to store in the reserved block, we keep one-byte of the ECC in the data block and only maintain the other four bytes in reserved entry to align the four-byte segment.

Table 5.1 also contains the details of different combinations. In the table, form A and form B are two possible writing forms for a particular compressed data pattern in the worst case. For example, data from group G_{small} will always writes its ECC together with itself. In the most common cases, data from groups G_{medium} and G_{large} will be written together with their ECCs. However, in the worst case where there is no sufficient fragment left, a one-byte EDC is maintained following the data from G_{medium}. For data from G_{large}, the first byte of its ECC and its one-byte EDC will be maintained together with data block. This guarantees that the remaining ECC code word is exactly four bytes, which uses one segment in the last cache entry in that set.

We implement the control logic with verilog HDL to calculate write location and update status bits based on current cache status. The circuit is designed following the truth Table 5.2 and synthesized by Cadence RTL compiler with 65nm TSMC library. In total, the circuit solely costs tens of standard cells and the power consumption is negligible. In the case that the coming data is uncompressed, the special bit S is set and its EDC byte stores in the two C fields. The ECC pointer E is updated to point to its ECC location. The status bits E, D and S are updated correspondingly during ECC write to facilitate cache read request in future.

Parameter	Value
Processor	1 or 4 ooo cores, 4GHz, 14-stage pipeline
Functional units	2 IntALU, 4 LSU, 2 FPALU
IQ, ROB and LSQ	IQ 32, ROB 128, LQ 48, SQ 44
Physical registers	128 Int, 128 FP, 128 BR, 128 ST
L1 caches (per core)	64KB Inst/64KB Data, 2-way, 16B line, hit latnecy: 3-cycle Inst, 3-cycle Data
L2 cache (shared)	0.5∼8MB, 8-way, 64B line, 14-cycle latency for 2MB cache
Memory	200 cycles latency

Table 5.3: Major simulation parameters.

5.4 Experimental Methodologies

We use a cycle accurate full system simulator Marss-x86 [78], which is for x86-64 architecture. Both single- and four-core out-of-order configurations are simulated with a two-level cache hierarchy. To evaluate the compression ratio of the proposed convergent allocation scheme, we configure L2 cache with 64-byte block size with capacity varying from 512KB to 8MB. All the 27 compilable benchmarks from 29 SPEC CPU2006 suite [98] and 11 out of 13 PARSEC [8] benchmarks are simulated. We create checkpoints for Marss simulator after initialization. After a warm-up period, we sample the compression ratio every 100 million cycles for the following 500 million instructions. Table 5.3 shows the major parameters for the simulation platform.

For quad-core system simulation, we fix the L2 cache to 8-way 2MB capacity. The cache LRU replacement policy is optimized to improve the compression ratio and system performance. The detailed optimization is illustrated in Algorithm 1 in Section 5.5.

We construct four-core workloads based on their cache capacity sensitivity and compression ratio following study [80]. The detailed workloads are presented in Table 5.4. The system performance is characterized using SMT weighted speedup [96] $\sum_i^n \frac{IPC_{multi}[i]}{IPC_{single}[i]}$, where n is the total number of applications running, $IPC_{multi}[i]$ and $IPC_{single}[i]$ are IPC values of application i running under multi-core and single-core environment, respectively.

Workload	Applications
C4S4	soplex, astar, hmmer, bzip2
C4S3	soplex, hmmer, xalancbmk, h264ref
C4S0	libquantum, gobmk, tonto, namd
C4S2	calculix, gamess, gromacs, leslie3d
C0S1	mcf, lbm, milc, sjeng
C4S3	soplex, astar, hmmer, libquantum
C3S4	bzip2, xalancbmk, h264ref, calculix
C4S1	soplex, libquantum, gobmk, tonto
C4S1	hmmer, namd, calculix, gamess
C1S1	soplex, mcf, lbm, milc
C1S1	hmmer, lbm, milc, sjeng
C0S4	sphinx3, perlbench, omnetpp, gcc
C2S4	sphinx3, omnetpp, soplex, astar
C2S4	perlbench, gcc, hmmer, h264ref

Table 5.4: Workload construction. C: compression ratio, S: sensitivity. Compression ratio is evaluated using 2MB 8-way L2 cache and sensitivity is examined with L2 cache size varying from 512KB to 8MB. CmSn consists of m benchmarks with high compression ratio and n benchmarks with high cache capacity sensitivity.

5.5 Experimental Results

5.5.1 Comparison of Cache Allocation Schemes

In the simulation of the three cache allocation schemes, we first optimize cache replacement policy based on LRU, as we observed that cache compression introduces extra replacement due to fat write. For example, in a straightforward replacement policy in fixed-space and convergent allocations, a fat write to Data$_A$ will introduce replacement of its neighbor, marked as Data$_B$. However, this will introduce thrashing if Data$_B$ is required next. Therefore, we make the following optimization for a fat write. The policy is shown in Algorithm 1.

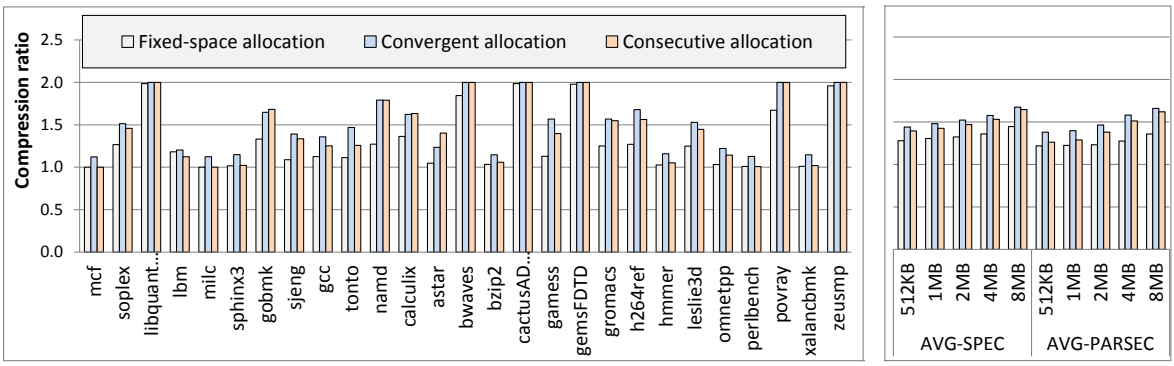
In the algorithm, it first searches if coming data block is suitable to any invalid cache block before applying LRU algorithm directly. If so, the coming data block uses the invalid entry. This optimization can improve compression ratio and avoid thrashing. In previous example, Data$_A$ is a fat write to Data$_B$ and will introduce eviction to Data$_B$. After optimization, Data$_A$ is placed in another invalid entry and Data$_B$ is thus saved in cache.

Compression Ratio. Figure 5.7 presents cache compression ratio applying the three dif-

Algorithm 1 Cache replacement policy based on LRU

```
while (Cache set scan not finished) do
  if (find an invalidated way in set) then
    if (data fit into way) then
      write to the way and return
    end if
  end if
end while
while (Cache set scan not finished) do
  if (find a way using LRU) then
    if (data fit into way) then
      write to the way and return
    end if
  end if
end while
if (fixed-space or convergent allocation) then
  evict its neighbor
  write data and return
end if
if (consecutive allocation) then
  evict multiple blocks till data fit
  write data and return
end if
```

ferent allocation schemes based on the optimized replacement policy. Compression ratio is defined as the compressed cache capacity over uncompressed cache size and is normalized to fixed-space allocation. Figure 5.7a shows the compression ratio for SPEC benchmarks of a 2MB cache in detail. On average, the proposed convergent allocation improves the compression ratio by 15.0% with 40.9% in maximum. The reason for the improvement is illustrated in Figure 5.3. By applying $B\Delta I$ compression algorithm in our simulation, the compressed data length can be greater than half of cache block size, which is 32 bytes. Therefore, two data blocks may not maintain in one cache entry by fixed-space allocation but by convergent allocation scheme. Figure 5.7b presents the averages of SPEC and PARSEC benchmarks for caches varying from 512KB to 8MB. On average, convergent allocation improves the compression ratio by 12.6%, 13.3%, 15.0%, 16.0%, 15.8% for SPEC benchmarks of those cache sizes, respectively. The improvements for PARSEC benchmarks are 13.2%, 14.2%, 18.9%, 24.0% and 22.1%, respectively. For same compressed data length, an efficient placement policy can improve cache capacity



(a) Detailed compression ratio comparison for a 2MB cache for SPEC benchmarks.

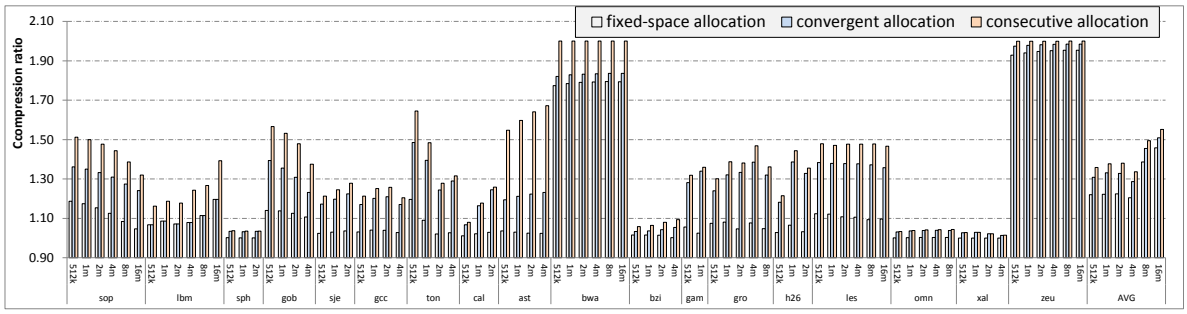
(b) Average compression ratio comparison for varied cache capacities.

Figure 5.7: Comparison of cache compression ratio for three cache allocation schemes for both SPEC and PARSEC benchmarks. The legend and y-axis for right figure is not presented to save space and it is the same to left figure.

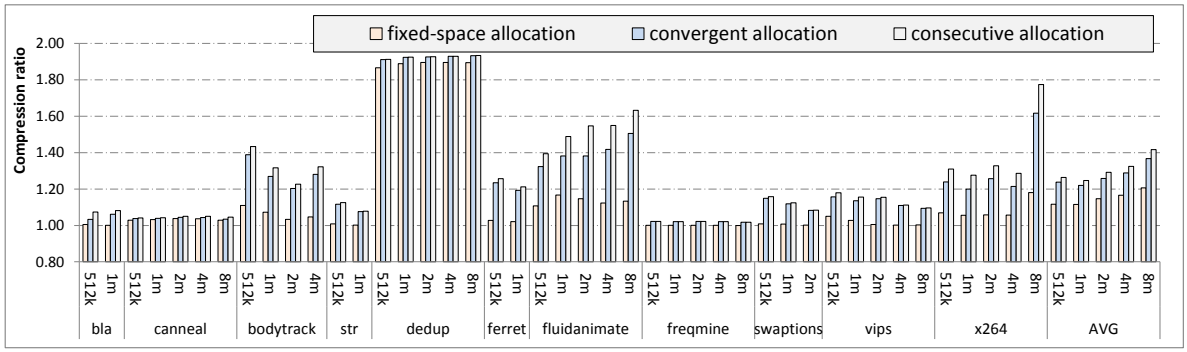
utilization.

When compared with consecutive allocation, the results are interesting. For some benchmarks, the proposed convergent allocation performs higher compression ratio than consecutive allocation. The possible reason is that a fat write may evict multiple data blocks in our optimized cache LRU replacement policy as illustrated in Algorithm 1. This reduces the number of effective data blocks in the cache, but may benefit the system performance. We thus did a profiling simulation that compares cache compression ratio for the same data placed in cache using the three allocation schemes. In profiling simulation, cache is physically configured to 2x capacity. We sample the 2x data and virtually place them in a 1x sized compressed cache using those allocation schemes. Note that 1x sized compressed cache targets 2x capacity depends on compression ratio. It therefore shows the layout impact to compression ratio.

Figure 5.8 presents the results. Figure 5.8a compares the profiled compression ratio for SPEC benchmarks. The benchmark names are represented by the first three characters to make the figure readable. Part of the combinations of benchmarks and cache capacity configurations are not presented as that large cache is not fully utilized. It is obvious that our proposed convergent allocation scheme improves cache compression ratio compared to fixed-space allocation. The maximum improvement is 30.8% for gamess benchmark for 1M cache



(a) Profiled cache compression ratio for SPEC benchmarks

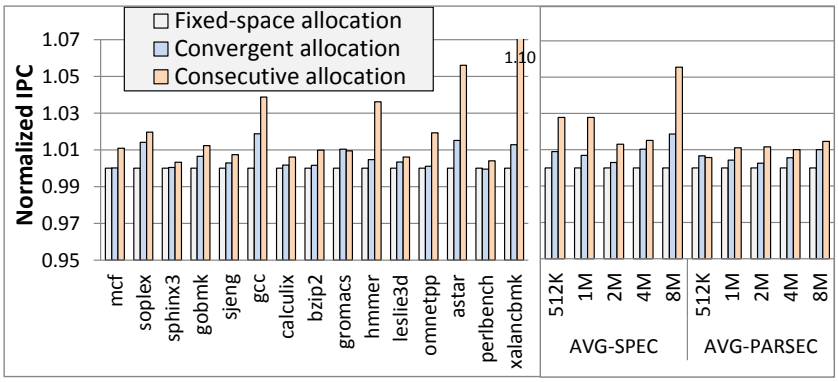


(b) Profiled cache compression ratio for PARSEC benchmarks

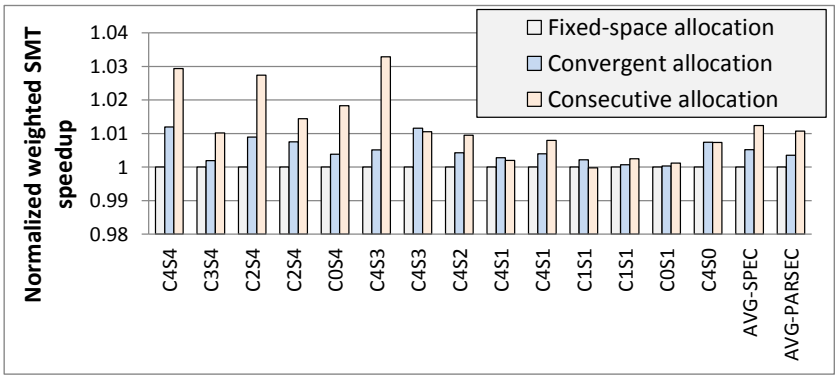
Figure 5.8: Comparison of cache compression ratio for three cache allocation schemes for varied cache capacities. Due to space limit, part of the benchmarks are represented by its first three characters. Large cache sizes for which benchmarks cannot fully utilize are not presented.

capacity. On average, convergent allocation scheme improves compression ratio by 7.3%, 8.9%, 8.5%, 6.8%, 4.9% and 3.5% for 512KB, 1MB, 2MB, 4MB, 8MB and 16MB cache capacities respectively, compared to fixed-space allocation. When being compared to the ideal compression ratio of consecutive allocation, it is 3.6%, 3.4%, 3.8%, 3.7%, 2.7% and 2.7% lower for those cache capacities, respectively. Figure 5.8b presents the comparison for PARSEC benchmarks and it shows the same trend. On average, convergent allocation improves compression ratio by 10.8%, 9.3%, 9.7%, 10.5% and 13.3% for the varied cache capacities respectively, compared to fix-space allocation. The differences are merely 2.1%, 2.2%, 2.6%, 2.7% and 3.5% compared to consecutive allocation scheme for these cache capacities, respectively.

System Performance. Figure 5.9 illustrates the performance comparison of the three allocation schemes for single- and four- core systems. Benchmarks not presented here are insensitive to cache capacity. The left part of Figure 5.9a shows system performance comparison



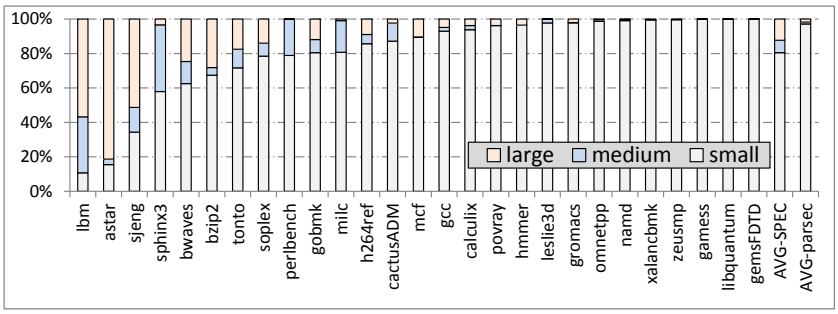
(a) Single core system performance comparison.



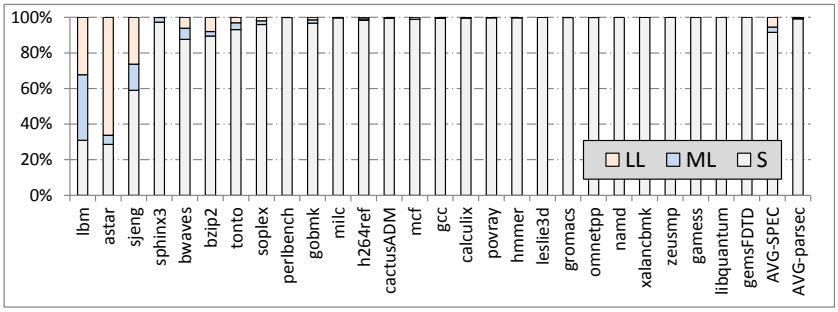
(b) Four-core system performance comparison.

Figure 5.9: Single- and four- core system performance comparison of three cache allocation schemes.

for a 2M L2 cache in detail and the right part presents the average results for varied cache capacities. All the IPCs are normalized to fixed-space allocation scheme. On average, there is 1∼2% improvement for convergent allocation compared to fixed-space allocation. The improvement comes from the increased cache capacity from compressing. Note that consecutive allocation performs better than convergent allocation in almost all cases in the simulation although its compression ratio can be lower than convergent allocation as shown in Figure 5.7. This presents the fact that both cache capacity and the importance of data blocks maintained in cache affect system performance. Figure 5.9b presents performance results for four-core system with 2M LLC. All data are normalized to fixed-space allocation. On average, convergent allocation improves system performance by 1∼2% and higher improvement comes from capacity-sensitive workloads.



(a) Compressed data patterns in a 16-way set associative, 2M L2 cache. Groups are defined following Table 5.1.



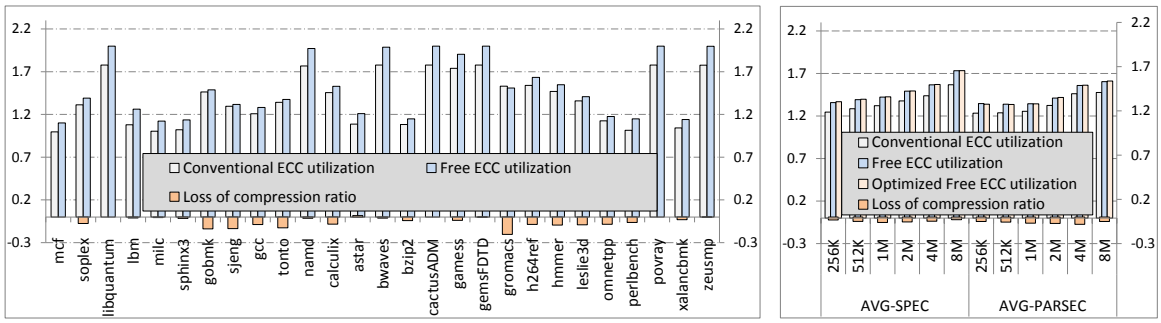
(b) Mathematical probability for combinations of two compressed patterns.

Figure 5.10: Compressed data patterns rate and probability of various combinations. "LL": both data blocks are from group large; "ML": two data blocks are from group medium and group large; "S": at least one data is from group small or both data blocks are from group medium. In combination "S", there is sufficient fragment in the same cache entry to place ECC.

5.5.2 $B\Delta I$ Data Compressed Pattern Analysis

We analyze $B\Delta I$ data compression patterns before presenting the details of *Free ECC* results. Figure 5.10a presents the ratio of different compression patterns in $B\Delta I$ compression algorithm for real data in L2 cache. We group those patterns into three groups following the definition in Section 5.3.2. It is obvious that group G_{small} occupies a large portion across the three groups for most benchmarks. It is as high as 100% for a few benchmarks, i.e. libquantum and gemsFDTD. On average, the ratios are 80.5%, 7.3% and 12.3% for small, medium and large groups respectively for SPEC benchmarks. Those results are 97.2%, 1.0% and 1.8% for PARSEC benchmarks. The higher rate for small and medium group means higher probability that fragments left in cache is sufficient for ECC.

In Table 5.2, we analyze the writing forms for coming data blocks. As long as one of



(a) Effective cache utilization comparison of conventional ECC and unoptimized *Free ECC* design for SPEC CPU2006 benchmarks on 2MB L2 cache.

(b) Average effective cache utilization comparison for SPEC and PARSEC benchmarks on varied cache capacities.

Figure 5.11: Comparison of effective cache capacity utilization of conventional ECC, optimized *Free ECC* and *Free ECC* cache design.

the two data blocks is from group small or both data are from group medium, there is sufficient fragment left in the same data entry to maintain ECC. We thus mathematically analyze the probability that ECC can be maintained in fragment based on the patterns ratio in Figure 5.10a. Figure 5.10b shows the mathematical deduction results. In the figure, "S" represents a combination that at least one data is from group small or both data are from group medium. "ML" means the two data are from group medium and large. "LL" means both data are from group large. Note that case "LL" is impractical in reality. It is obvious that for most of the compressed data, there is sufficient fragment left in the same data entry to maintain ECC. In such cases, the proposed *Free ECC* design thus introduces no penalty to compression ratio as ECCs are placed in otherwise unused fragments following the data and no reservation space in last cache entry is required. The average probability of such cases is as high as 91.6% and 99.2% for SPEC and PARSEC benchmarks, respectively by mathematical deduction. This firmly consolidates our proposal to embed ECC into the otherwise unused fragments. The *Free ECC* design can thus save extra space required for ECC and power consumption while introduces minimal cost to compression ratio.

5.5.3 Effective Utilization of Cache Capacity

Figure 5.11 shows the comparison of effective cache capacity utilization for compressed cache with conventional ECC and *Free ECC* cache design. As illustrated in the figure, the compression ratio is reduced slightly by 3.0%, 3.6%, 4.4%, 3.6%, 3.0% and 1.7% for 256KB, 512KB, 1MB, 2MB, 4MB and 8MB caches, respectively for SPEC benchmarks. Those values are 3.2%, 4.1% 4.5%, 5.5%, 4.8% and 3.3% for PARSEC benchmarks. Although *Free ECC* cache reduces compression ratio, it still gains in effective capacity utilization as conventional ECC has a 12.5% overhead for ECC storage. On average, the gains are 9.0%, 8.2%, 7.6%, 8.3%, 8.9%, and 10.5% for those cache capacities for SPEC benchmarks, respectively. Those improvements are 8.9%, 7.9%, 7.4%, 6.4%, 7.1% and 8.8% for PARSEC benchmarks. With higher compression ratio, the improvement of effective utilization is potentially higher and the highest improvement is 0.22 in theory, which is gained by several benchmarks, e.g. zeusmp, povray, and gemsFDTD. For a few benchmarks that can not hold ECC in fragments efficiently, there is slight loss in utilization. For example, the loss is 1.7% for gromacs with 2MB physical cache capacity.

With optimization that no ECC storage is required for clean data blocks, loss of compression ratio is reduced slightly and capacity utilization is improved. The details of each benchmark are not presented in figure to make it readable and the average results are shown in Figure 5.11b. For some benchmarks, the improvement is as high as 0.029 (i.e. leslie3d), which means for the 2MB cache simulated, optimization can improve effective cache capacity by 60KB compared to unoptimized *Free ECC* design. Although the improvement is not significant on average, it gains without any extra cost.

We analyze the details for optimized *Free ECC* design. Figure 5.12 presents clean data blocks ratio in cache. As it shows, 12 out of 27 simulated SPEC CPU2006 benchmarks have more than 50% of clean blocks in entire cache. On average, 47.5% of data blocks are clean for the simulated 2MB L2 cache. This provides opportunity for our proposed optimization that there is no need to maintain ECC for clean data blocks. It therefore reduces requirement of fragments in cache to maintain ECC and reduces the negative impact of *Free ECC* design. The clean

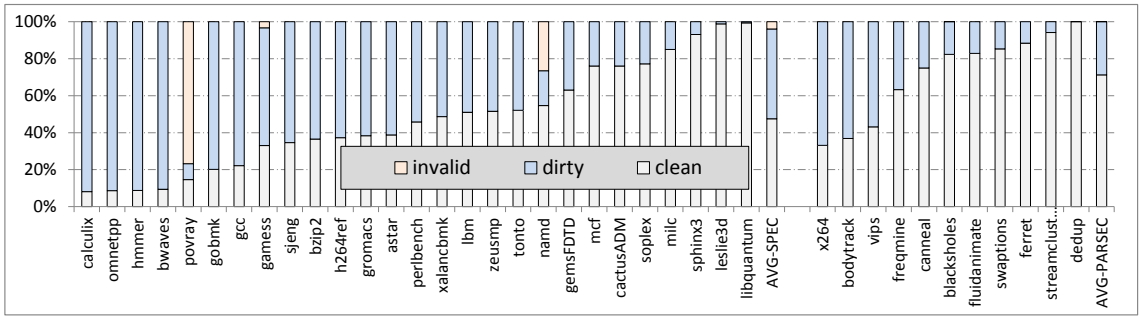
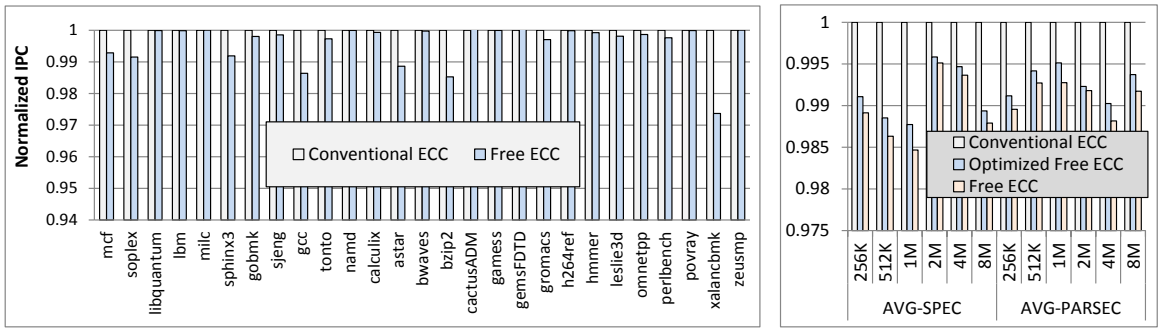


Figure 5.12: Clean data blocks rate for SPEC and PARSEC benchmarks in a 2MB L2 cache.



(a) Single-core system performance comparison of conventional ECC with *Free ECC* cache.

(b) Average single-core system performance comparison for varied cache capacities.

Figure 5.13: Comparison of cache performance for conventional ECC with *Free ECC* cache.

data block ratio for PARSEC benchmarks is higher, for which, 8 out of 11 simulated PARSEC benchmarks have more than 60% clean data blocks in cache. On average, the clean blocks rate is 71.3%, which explains that the proposed optimization can improve cache compression ratio and cache capacity utilization.

5.5.4 Performance of Free ECC

Figure 5.13 illustrates the system performance comparison of *Free ECC* with conventional compressed cache for single-core configuration. Figure 5.13a presents details for a 2MB physical cache for SPEC benchmarks and Figure 5.13b shows the average performance comparison for varied cache capacities. The optimized *Free ECC* cache performance is included. Both the detailed and average performance show that the performance loss is around 1% for *Free ECC* design for all cache configurations for both SPEC and PARSEC benchmarks. As most ECC

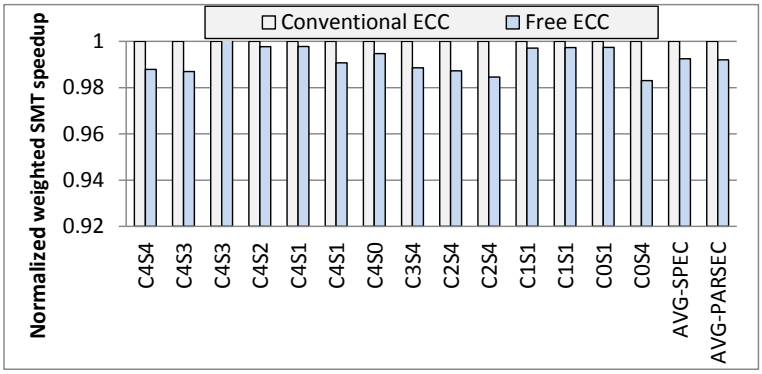


Figure 5.14: Four-core system performance comparison of conventional ECC with *Free ECC*.

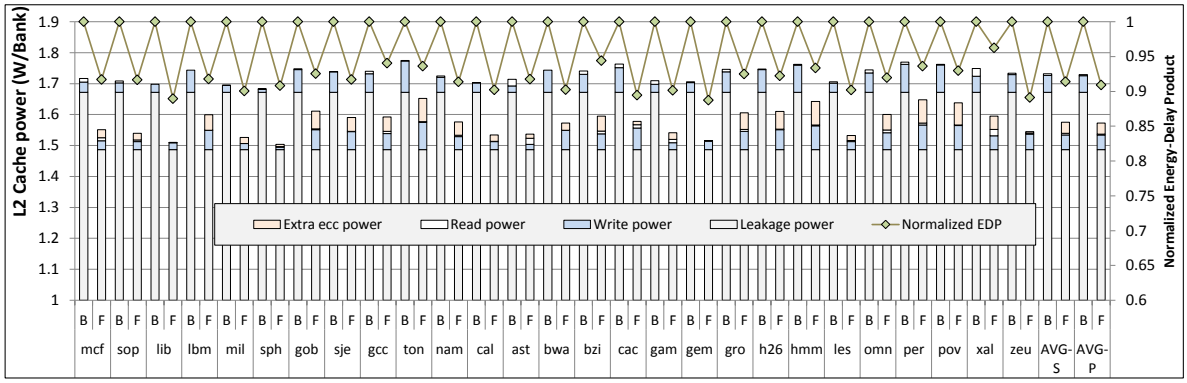


Figure 5.15: 2MB L2 cache power consumption comparison of conventional ECC cache and *Free ECC* cache design in a single-core machine. AVG-S and AVG-P mean the average of SPEC and PARSEC benchmarks. B: baseline, F: *Free ECC*.

can fit into unused fragments in compressed cache, there is only small capacity loss for ECC reservation, which affects system performance slightly. The proposed optimization improves system performance slightly as show in Figure 5.13b. Although the improvements are minimal, they are obtained without any cost.

Figure 5.14 presents performance of four-core configuration. The cache capacity is fixed to 2MB in physical. Consistent with single core results, the loss is within 1% for both SPEC and PARSEC workloads. The results show that *Free ECC* design efficiently uses the fragments left in compressed cache schemes and introduces negligible impact to cache performance. The performance for optimized *Free ECC* design is not presented here and it is consistent with single core results. The improvement is slight but the gain is obtained without extra cost.

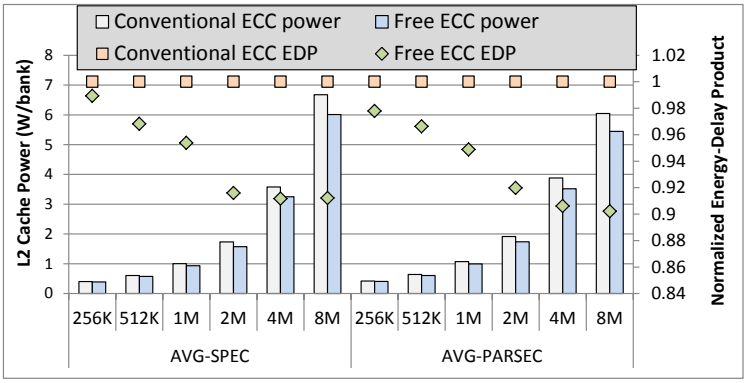


Figure 5.16: Average L2 cache power consumption for conventional ECC and *Free ECC* cache design for SPEC and PARSEC benchmarks. Cache capacity varies from 512KB to 8MB.

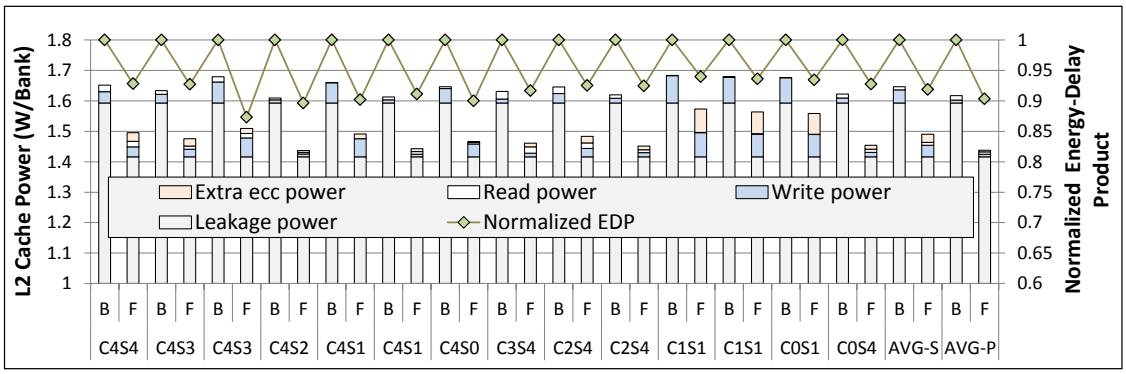


Figure 5.17: 2MB L2 cache power consumption comparison of conventional ECC cache and *Free ECC* cache design in a four-core system. B: baseline, F: free ECC design.

5.5.5 Cache Power Consumption

We further evaluate cache power consumption of *Free ECC* when compared with conventional ECC design of compressed cache using CACTI 5.3 [102]. The cache is configured with two banks and 45nm technology is applied. Figure 5.15 presents the power consumption for 2MB cache with single core configuration for SPEC benchmarks in detail. The baseline is the conventional compressed cache with 12.5% storage overhead for ECC. For 2MB cache capacity, the *Free ECC* design reduces the power consumption by 9.0% and 9.1% on average for SPEC and PARSEC benchmarks, respectively. The maximum improvement is 11.1% for gemsFDTD because the ratio of extra ECC write is small. As the figure shows, *Free ECC* cache introduces extra ECC write which increases the write power. However, it significantly reduces the leakage

power, which dominates the power consumption.

Figure 5.16 illustrates the average power consumption for caches varying from 512KB to 8MB. *Free ECC* cache reduces the consumed power by 2.7%, 5.1%, 6.9%, 9.0%, 9.3% and 10.0% for SPEC benchmarks for those caches, respectively. Those improvements are 3.1%, 5.4%, 7.3%, 9.1%, 9,4% and 9.9% for PARSEC benchmarks. As capacity grows, the leakage power increases and *Free ECC* design can significantly reduce the overhead in conventional ECC cache, thus power efficiency improves with the increase of cache size.

The detailed power consumption of optimized *Free ECC* design is not presented in figure as it is consistent with performance improvement. The optimization can reduce cache write power and leakage power slightly. As it avoids partial writing of ECC to the reserved cache entry and improves system performance compared to *Free ECC*, the power consumption is thus reduced slightly.

Figure 5.17 presents the 2MB L2 cache power consumption for four-core system. On average, *Free ECC* design saves power by 9.3% as it significantly reduces the leakage power. As shown in the figure, *Free ECC* saves slightly more power for workloads that with higher compression ratios. This is because the extra ECC write is less than that of workloads with lower compression ratios. Overall, *Free ECC* design removes the dedicated storage of ECC in conventional ECC cache and thus saves the power consumption of cache memories.

5.5.6 Energy-Delay Product Improvement

Figures 5.15, 5.16 and 5.17 also present the Energy-Delay Product (EDP) comparison for conventional compressed ECC cache and *Free ECC* cache. Across all configurations with all workloads, *Free ECC* design improves system EDP. On average, *Free ECC* cache improves EDP by 1.1%, 3.2%, 4.6%, 8.4%, 8.8% and 8.8% for SPEC benchmarks for varied cache capacities from 256KB to 8MB, respectively. Higher cache capacity presents higher EDP saving which is consistent with the trend of power saving. Those improvements for PARSEC benchmarks are 2.2%, 3.4%, 5.1%, 8.0%, 9.4% and 9.4%, respectively. These results show that the performance loss caused by *Free ECC* design is slight and its saving of power consumption offsets the performance loss.

5.6 Summary

We have presented a low-cost and efficient reliability scheme for compressed last-level caches. An efficient cache allocation scheme is designed for compressed data layout and *Free ECC* cache organization is proposed based on this scheme to explore the hidden error protection capability of compressed cache. The design fits ECC/EDC into small fragments left in compressed cache to remove the dedicated ECC storage required in conventional ECC design. It significantly improves the effective cache capacity utilization and power efficiency, while introducing minimal impact to system performance. The *Free ECC* is specific for cache compression and the design is simple, effective and power efficient.

CHAPTER 6. MEMGUARD: A LOW COST AND ENERGY EFFICIENT DESIGN TO SUPPORT AND ENHANCE MEMORY SYSTEM RELIABILITY

The conventional reliability design is to use redundant memory bits for error detection and correction, with significant storage, cost and power overheads. In this chapter, we propose a novel, system-level scheme called MemGuard for memory error detection. With OS-based checkpointing, it is also able to recover program execution from memory errors. The proposed MemGuard design is much stronger than SECDED in error detection capability and it incurs negligible hardware cost and energy overhead, no storage overhead, and is compatible with various memory organizations. We have comprehensively investigated and evaluated the feasibility and reliability of the scheme. Our mathematical deduction and synthetic simulation prove that MemGuard is robust and reliable.

6.1 Introduction

Main memory error protection is generally implemented with redundant information stored in memory devices. A conventional ECC memory module uses SECDED code with a 12.5% storage and power overhead. Chipkill Correct [15, 64] has additional capability of Single Device Data Correction (SDDC) to tolerate multiple errors from a single chip with high power consumption. Both of these designs are incompatible with non-ECC memory modules. They also put a restriction on memory organization such that recently proposed memory sub-ranking [105, 121, 2, 1], which improves memory energy efficiency, may not be used. The majority of consumer-level computers, including desktop and laptop computers and mobile devices, have not yet adopted any memory error protection scheme.

In this chapter, we propose *MemGuard*, a system-level scheme with lightweight hardware extension to support or enhance memory reliability for a wide spectrum of computer systems. The core part of MemGuard is a low-cost and highly-effective mechanism of *memory error detection*, which is revised from more complex designs of memory integrity verification [4, 99] proposed for secure processors. By maintaining a *read log hash* (ReadHash) and a *write log hash* (WriteHash), 128-bit each, MemGuard can detect multi-bit errors of the main memory in very strong confidence. Conceptually, ReadHash is a hashed value of the log of all read accesses from the main memory, and WriteHash is one for all write accesses to the main memory. For each access, the logged data is a tuple (address, data). Periodically or at the end of program execution, MemGuard synchronizes ReadHash and WriteHash to the same point of execution and then matches them. A mismatch means that the result of at least one read does not match that of the previous write to the same memory address, and therefore a memory error must have happened.

MemGuard does not have *hardware* memory error correction as ECC and Chipkill Correct do, but is much stronger in error detection. It utilizes (and relies on) an OS-based checkpointing system for error recovery. If a memory error is detected during the execution of a given program, the program will be rolled back to the latest checkpointed state. Since memory error is relatively infrequent (hours for memory of gigabytes), checkpointing can be done at a slow pace and the performance degradation from checkpointing is insignificant. MemGuard is very useful for computers that have no other memory error protection. For consumer-level computers in particular, MemGuard does not require any change to computer motherboards or memory modules, the hardware cost of MemGuard itself is negligible, and its protection can be selectively enabled for programs for which computation reliability is desired. Additionally, MemGuard is compatible with sub-ranked memories as well as narrow-ranked memories used in mobile devices.

MemGuard is also useful for large-scale, high-performance computing applications, in which a single-node crash may lead to checkpoint rollback of many computing nodes. Many of those applications already use checkpointing or redundant nodes or both, otherwise they may not run to completion [38, 22]. With MemGuard, they do not have to run computers with ECC

memory, which leads to reduced cost and improved energy efficiency. For computers of ECC or Chipkill Correct memories, MemGuard can further enhance memory reliability by reducing the probability of silent data corruption (false negative), particularly when single-bit error correction occurs. MemGuard alone, however, may not be suitable for commercial workloads that require both high availability and instant response, as checkpoint recovery may cause a delay visible to end users.

MemGuard is motivated by memory integrity verification using read and write log hashes [99] in secure processor, but the design objective and complexity are very different. In secure processor design, the hash function must be very strong to repel carefully crafted attacks from adversaries, and particularly it has to include a timestamp per memory block to detect replay attacks. Significant design complexity, storage and energy overheads, and high cost can be justified. In memory reliability design, the error pattern is random and unintentional, and for consumer-level computers the cost has to be contained. In the design of MemGuard, we carefully choose a simple, non-cryptographic hash function to minimize the impact on energy efficiency and cost, and has dropped the use of timestamp. Had timestamp been used, there would be significant storage, performance and energy efficiency overhead for modern DDRx memories. To our best knowledge, we are the first to adopt such a scheme solely for memory reliability in conventional computers and we comprehensively evaluate and prove its feasibility and reliability for error protection.

The rest of the chapter is organized as follows. Section 6.2 introduces background of main memory organization variations and related work. Section 6.3 presents the details of MemGuard design scheme. Section 6.4 describes the evaluation methodologies and the results are presented and analyzed in Section 6.5. Finally, Section 6.6 concludes the chapter.

6.2 Background and Related Work

6.2.1 Memory Organization Variants

Various memory structures and organizations have been proposed to improve memory performance or energy efficiency. In sub-ranked DRAM memories [105, 121, 2, 1], the number of

DRAM devices in a rank is reduced so as to reduce DRAM operation power spent on precharge and activation. However, the reduced number of devices in a rank presents a new challenge to memory error protection as it breaks the 8:1 ratio of conventional SECDED design. The details are presented in Section 2.4. Another type of memory structure stacks DRAM dies by Through-Silicon Via (TSV) technology to reduce its power consumption and improve bandwidth and capacity. One promising product is Hybrid Memory Cube (HMC) [34] with high bandwidth and low power consumption and it is projected to appear in market in 2014. Such a product drastically changes conventional DRAM organization and also presents challenges for memory error protection. MemGuard does not put any constraints on memory organization and thus can work with those new organizations.

6.2.2 Related Work

There have been many studies on memory system reliability [113, 114, 103, 73]. Most of them focus on error correction for phase change memory and DRAM at memory module level. One of the most recent studies closely related to our work is ArchShield [73]. It proposes an architectural framework to tolerate fabrication faulty cells (hard error) induced by the extreme scaling of DRAM. In their design, a fault map is used to record all the faulty cell locations obtained by built-in self test. By consulting the fault map, it maintains replications of those word in memory space for error correction. MemGuard is very different from ArchShield, and they serve for different purposes. MemGuard targets soft and intermittent errors rather than hard errors, and it does not have to maintain faulty cell locations.

6.3 MemGuard Design

6.3.1 Incremental Hash Functions

Incremental hash function is first proposed by Mihir Bellare et al [7] in 1990s. Such a hash function has the property that the cost to update the result hash upon a modification to the original message is proportional to the modification. Consider a message M and its hash value $H(M)$. With modifications δ to the message, the result message is denoted as $M' = M + \delta$, in

which + denotes a modification such as to replace, insert or delete a data block of the message. Incremental hash function can update the result hash of modified message using the following equation:

$$H(M') = H(M) + H(\delta)$$

where = and + are equality and modification operations, respectively, for the defined hash function. The equation means that as long as we have $H(M)$ and $H(\delta)$, we could calculate the result hash. There is no need to retrieve the original message information of M.

Multiset hash functions are a particular type of incremental hash function, operating on multiset (set) [4]. A multiset is defined as a finite unordered collection of elements where the occurrence of each element can be greater than one. If the occurrence of each element is exactly one time, the multiset is reduced to a set. Multiset hash functions are incremental and the result hash is independent of the ordering of the input elements. In detail, if we use $\cup$ to denote the union operation of multiset, the properties of multiset can be denoted using the following two equations:

$$H(M \cup \{b\}) = H(M) +_H H(\{b\}) \tag{6.1}$$

$$H(\{b_1\}) +_H H(\{b_2\}) = H(\{b_2\}) +_H H(\{b_1\}) \tag{6.2}$$

where $+_H$ denotes defined hash addition operation. Equations (6.1) and (6.2) show the properties of additivity and commutativity, respectively, for multiset hash functions.

These two properties can be explored in main memory system to create a fingerprint of memory accesses. In that scenario, each memory access is regarded as an item and a sequence of memory accesses is considered as a multiset as there exists duplicated memory accesses to same memory address. Define $H(\{q\}) = H_s(s_q)$, where H_s is a hash function that takes input of a string s_q formed of (address, data) pair of the memory request q. The details of selecting H_s hash function are presented in Section 6.3.4. Based on the two properties of multiset hash function, we thus have the hash for a sequence Q of memory requests q_i ($i \in [1, N]$) below

$$H(Q) = H_s(s_{q_1}) +_H H_s(s_{q_2}) +_H \cdots +_H H_s(s_{q_N}) \tag{6.3}$$

, where Q is a multiset that $Q = \{q_1\} \cup \{q_2\} \cdots \cup \{q_N\}$. Following the two properties of multiset hash functions, the result hash $H(Q)$ is irrelevant to the ordering of the requests and it can be

calculated incrementally by adding the hash value of coming memory request. In other words, $H(Q)$ represents a fingerprint of a sequence of memory requests.

The previous study [4] proposes four types of multiset hash functions: *MSet-XOR-Hash*, *MSet-Add-Hash*, *MSet-Mu-Hash* and *MSet-VAdd-Hash* based on four different operations: binary XOR, conventional addition, multiplication and vector addition, respectively. That means the $+_H$ can be any of these four operations to form a multiset hash function based on a strong hash H_s. *MSet-XOR-Hash*, *MSet-Mu-Hash* and *MSet-VAdd-Hash* are not proper in MemGuard design as they either merely support set collision resistance or introduce high overhead because of operational complexity. We choose *MSet-Add-Hash* for being multiset collision resistant and simple in operation. The function uses simple addition operation and outputs the lower m bits of the sum, where m is the output length of hash function H_s.

6.3.2 Log Hash Based Error Detection

In MemGuard design, error detection is implemented by maintaining and cross-checking a read log hash and a write log hash maintained in the memory controller. The two hashes are denoted as ReadHash and WriteHash, respectively, which *conceptually* log the (address, data) pairs of memory reads and memory writes. *Here memory reads and memory writes are different from actual reads and writes to the main memory, which will be discussed soon.* At runtime, the two hashes are updated upon memory events. Periodically and at the end of a program execution, MemGuard synchronizes the ReadHash and the WriteHash to ensure every memory read is logged in ReadHash and every write is logged in WriteHash. If no error occurs, the two hash values must match to each other. The details for selecting a hash function are discussed in Section 6.3.4.

Figure 6.1 and Algorithm 2 show the steps of updating the ReadHash and the WriteHash. We assume that the last-level cache is write-back and write-allocate. In the beginning, when the OS loads the program to be executed into memory, the memory controller will *log* each write of a memory block into WriteHash, i.e. to hash the pair (address, data) of the memory block into WriteHash. A memory block is a block of memory of the cache block size. Note that we do not assume the OS will load all memory pages of the program into memory

Algorithm 2 MemGuard error detection algorithm

Initialization Operation
function `init_block` ($\mathcal{T}$, Address, predefined Data) {
  update $\mathcal{T}$.WriteHash with the hash of (Address.predefined Data)
  write (predefined Data) to address (Address) in memory storage
}

Run-Time Operation
when there is a cache miss
function `read_block` ($\mathcal{T}$, Address) {
  read the Data from Address in memory
  update $\mathcal{T}$.ReadHash with the hash of (Address.Data)
}
and store the block in Cache

when there is cache eviction
function `write_block` ($\mathcal{T}$, Address, Data) {
  update $\mathcal{T}$.WriteHash with the hash of (Address.Data)
  if block is dirty **then**
    write (Data) to address (Address) in memory storage
  end if
}

Integrity Check Operation
function `integrity_check` ($\mathcal{T}$) {
  $New\mathcal{T} = (0, 0)$
  while block covered by $\mathcal{T}$
    if block is not in Cache **then**
      call `read_block` ($\mathcal{T}$, Address)
      update $New\mathcal{T}$.WriteHash with the hash of (Address, Data)
    end if
  end while
  if ReadHash == WriteHash **then**
    there is no error
    $\mathcal{T} = New\mathcal{T}$
  else
    there is error detected
  end if
}

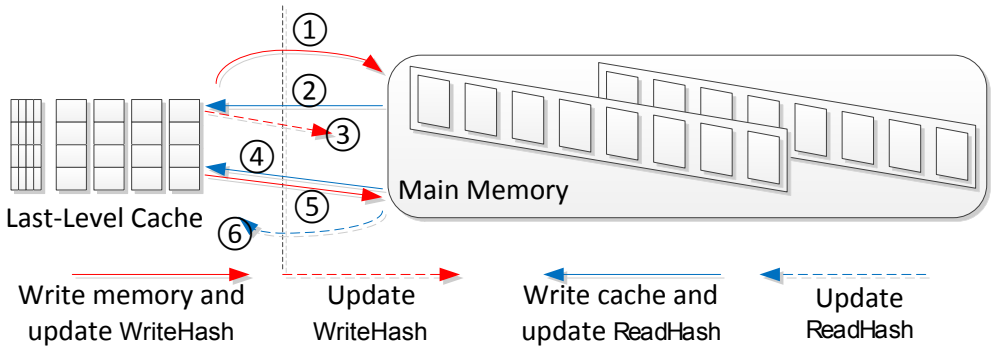


Figure 6.1: Memory operations for memory error detection.

at this time. If a memory page is loaded during program execution because of page fault, all blocks of the faulted page will be logged into WriteHash. We assume that there is a special DMA mode for program loading, which triggers the update of WriteHash at the memory controller. Normal memory accesses during OS kernel execution are not logged. This step is shown as ① in Figure 6.1.

When the program is running, the memory controller enters a logging mode (set by the OS). The `read_block` operation is executed at the memory controller when a last-level cache miss triggers a memory read. The (address, data) pair of the memory block is logged into ReadHash, shown as ② and ④ in Figure 6.1. If a memory block is loaded into cache for the first time after program loading, the read will match the write to the same memory block address at the time of program loading.

For each cache replacement (eviction), the `write_block` operation is executed to log a memory write in WriteHash. Note that an actual memory write happens only when the replaced block is dirty; however, even if the replaced block is clean, the eviction is treated as a (artificial) write and is logged in WriteHash. The purpose is to ensure the same set of (address, data) pairs will be logged in ReadHash and WriteHash: if the evicted block is loaded to cache again, the read will match the logged write at the time of eviction (assuming no error happens between the eviction and the load), no matter whether an actual memory write happened or not then. Figure 6.1 ③ and ⑤ shows the `write_block` operation for clean and dirty block, respectively.

At the end of program execution or during periodical memory error checking, the `integrity_`

`check` operation is executed as shown in Figure 6.1 as ⑥. For each physical memory block used by the program, the OS checks if the block is cached or not. If it is not cached, the OS requests the memory controller to load the memory block and log the (address, data) pair in ReadHash. Again, this is to ensure that the same set of (address, data) pairs will be logged in ReadHash and WriteHash: for every memory block not cached at this time, there is an (address, data) pair that has been logged in WriteHash but not in ReadHash. This step effectively synchronizes ReadHash and WriteHash. Then, if ReadHash and WriteHash match, the OS determines that no memory error has happened. Otherwise, it determines an error has happened and will trigger checkpoint recovery. Note that in this step, `integrity_check` creates a new WriteHash and adds all the uncached blocks into the new WriteHash when they are added to ReadHash. If no error occurred, the new WriteHash will replace the old WriteHash and ReadHash is cleared so that a new checking period starts. In addition, we assume that the memory controller is in a special mode, in which it does not log the normal memory accesses from the OS. We also assume that in this mode the OS may write a memory block address to a memory-mapped register of the memory controller to trigger an update of ReadHash.

MemGuard does not use per-block timestamp as in the study on secure processor design [99]. The purpose of using timestamp is to prevent replay attack from an adversary, which is very unlikely in a normal system. Assume that an error occurs in memory and changes the value of a memory block of block address A and value X. During program loading, (A, X) will be logged in WriteHash. When the block is loaded into cache, (A, X') is logged into ReadHash, which does not match the logged write. To form an error like one from a replay attack, the program execution may have to generate a write of (A, X') at a later time, and then another memory error shall happen to change the block's value back to X.

6.3.3 Reliability Analysis

MemGuard compares ReadHash and WriteHash to decide if a memory error occurs in a sequence of N accesses. A hash collision happens if a memory error has happened but ReadHash matches WriteHash. The probability of hash collision is the probability of false

negative of memory error detection. The study of *MSet-Add-Hash* [4] defines that two result hashes are equivalent if the modulus sums of the hashed values by hash function H_s are the same. In other words, if the modulus sum of hashes for each read request is equivalent to that of write request, there is a collision if error occurred in one or more of those accesses. Given that the collision rate is $\frac{1}{2^m}$ for a strong m-bit hash function and assume i out of N requests have errors, the probability of collision is given by Formula (6.4) given that the hash result for each access is in the range of 2^m:

$$P(i) = (\frac{1}{2^m})^i + (\frac{1}{2^m})^{i-1}(1 - \frac{1}{2^m}) + \cdots + \frac{1}{2^m}(1 - \frac{1}{2^m})^{i-1} \tag{6.4}$$

Let p_0, p_d and p_w denote the probabilities of no error, detectable error and undetectable error in one data block for conventional error protection scheme, respectively. The probability that i out of N accesses have error is $C_N^i(p_d+p_w)^i(1-p_d-p_w)^{N-i}$. Combining Equation (6.4), we thus have the collision rate for MemGuard design, represented by

$$F_{MemGuard} = \sum_{i=1}^{N} P(i) \cdot C_N^i(p_d + p_w)^i(1 - p_d - p_w)^{N-i} \tag{6.5}$$

For conventional error protection scheme, it fails as long as an undetectable error occurred to one of the N requests. Therefore, the failure rate for conventional error protection is given by

$$F_{conventional} = 1 - (1 - p_w)^N \tag{6.6}$$

Based on Formulas (6.5) and (6.6), we build a simple model to compare the error detection capability of MemGuard design with the conventional SECDED protection. Assume the probability that any one bit error occurs to a data block with n_b bits is p in a time period t. Then, $p_0 = (1-p)^{n_b}$ denotes the probability of no error in the data block. The probability of having an exact one-bit error is denoted as $p_1 = n_b p(1-p)^{n_b-1}$. The probability for two-bit error is denoted as $p_2 = C_{n_b}^2 p^2(1-p)^{n_b-2}$. As SECDED detects up to two-bit error, the detectable error rate for SECDED is thus approximately[1] $p_d = p_1 + p_2$; and the undetectable error rate is $p_w = 1 - p_0 - p_1 - p_2$.

[1]SECDED can detect three and more bits of errors with certain probabilities. The exact probability is evaluated by Monte Carlo simulation in Section 6.5.

With error rate of 25,000∼70,000 FIT/Mbit from study [92], we compare failure rate $F_{MemGuard}$ with $F_{conventional}$. The results are presented in Section 6.5.1.1. In general, MemGuard has much higher error detection rate than SECDED in all the cases we studied. Additionally, as the time period t grows, the undetectable error rate for SECDED grows but that for MemGuard does not increase. Furthermore, because the collision rate for a strong hash function is irrelevant with number of bits flipped in data, MemGuard design has better tolerance for multi-bit errors.

6.3.4 Selection of Hash Function

In MemGuard, a hash function H_s is used to convert the (address, data) pair of a memory request to a hash value before the multiset hash function is used. Hash functions are generally classified as cryptographic ones and non-cryptographic ones. Cryptographic hash functions are applied in secure applications and systems, i.e. MD4 [85], MD5 [86], SHA1 [46], SHA2 [30], etc., to protect the system from resourceful and malicious adversaries. They are typically complex in computation and have a low throughput. For example, MD5 generates a 128-bit hash code by four rounds of computation with 16 operations in each round. To the best of our knowledge, the best FPGA and ASIC implementation report 0.73GB/s and 0.26GB/s throughput with cost of 11,498 logic slices and 17,764 logic gates [41, 89], respectively. SHA1 is more complicated, requiring four rounds of 20 operations each to generate a 160-bit hash code.

Memory error is completely disparate from the intentional malicious attacks. It is caused by stochastic and unskilled cosmic rays, alpha particles and others. Therefore, simple and cost effective non-cryptographic hash functions can be adopted in MemGuard. There exist multiple non-cryptographic hash functions, for example Pearson Hashing [79], Fowler-Noll-Vo(FNV) [23], CRC [81], MurmurHash [5], lookup3 [42], SpookyHash [43], CityHash [26] and others. We still have to make a careful selection for performance and power efficiency.

Above all, the hash function should be collision resistant to provide a strong capability of error detection. A strong hash function with m-bit output has a collision rate of $1/2^m$ if we assume the function produces each hash value with exactly the same probability. With birthday attack, the rate increases to around $1/2^{\frac{m}{2}}$. Therefore, the selected hash function

should be able to create output with a decent length. The original Pearson hashing algorithm generates only 8-bit output, which cannot be adopted in this design. Second, the output hash values need to follow uniform distribution independent of the distribution of inputs. Otherwise, it will introduce clustering problem which can result in high collision rate. Third, a good hash function is required to have a good level of avalanche effect. Avalanche effect presents the ability for a hash function to produce a large change in output bits upon a minor modification to input bits. The avalanche effect thus can dissipate minor modification in input data to a large structure of output bits, which enhances error detection capability. Previous studies [20, 43] present that lookup3 Hash, SpookyHash, MurmurHash and CityHash all have good properties in avalanche effect.

In addition, the required hash function should be non-linear. Linear in this context means $H_s(A + B) = H_s(A) + H_s(B)$ mathematically, where A and B are two inputs, and $+$ can be general addition operation or binary xor operation. The reason is that it can introduce high collision when apply to *MSet-Add-Hash*. Following Formula (6.1), the hash values of two tampered data can cancel the modification out with a high possibility if the hash function is linear. For example, given two data A and B, assume there is single bit error to both of these data at the same bit position. Using A' and B' to denote the tampered data, we have $A + B = A' + B'$. Therefore, $H_s(A') + H_s(B') = H_s(A) + H_s(B)$ even if a single-bit error in data A and B creates great changes in their hash values, given a linear hash function. Thus, CRC hash function can not be applied as $CRC(A \oplus B) = CRC(A) \oplus CRC(B)$.

Based on the criteria required, we opt for SpookyHash designed by Jenkins in 2011 [43]. SpookyHash produces well distributed 128-bit hash values for variable length of input. It has been tested by the author for collision up to 2^{72} key pairs, which presents a good collision resistance. SpookyHash is said to achieve avalanches for 1-bit and 2-bit inputs which means that any 1-bit or 2-bit change in inputs results in a flip in each output bit with 1/2 possibility. In addition, SpookyHash is simple and fast and it costs merely 64-bit addition, xor and rotation operations. SpookyHash classifies keys as short if the length of input is less than 192 bytes and thus computes hash code with a simpler function. In MemGuard design, we combine each 64-byte data block with its address as an input key. The address is assumed to be 8 bytes

with paddings to make a sufficiently large space. The input key size is thus 72 bytes, which is regarded as short by SpookyHash.

The cost of SpookyHash compared to DDR3 main memory is minimal. We first carefully scrutinize SpookyHash function and observe that it takes 45 64-bit addition operations, 35 xor and 35 rotation operations to do the hashing. A previous study [66] presents that the energy consumption of a 64-bit adder with 65nm process is 8.2 pJ. Using this number, we calculate that SpookyHash consumes less than 1.0 nJ for each hashing operation. On the other hand, DDR3 memory power calculation is well established by Micron [70]. A complete memory access cycle includes precharge, activation, I/O drive and termination, and data transfer operations. In addition, there is consecutive background power consumption and it is increasing as the number of DRAM devices in a system grows. Taking Micron MT41J256M8 [69] device as an example, an eight x8 DIMM can consume 62.0 nJ energy for a complete memory access cycle. The energy consumption of a real system can be higher as it conventionally comprises multiple channels with multiple DIMMs per channel. Therefore, the energy consumption for SpookyHash is almost negligible compared to that of the DRAM access.

6.3.5 Checkpointing Mechanism for Error Recovery

In MemGuard design, errors in the program can be efficiently detected. We turn to OS-based checkpointing mechanism for error recovery. Checkpointing method has been studied for decades [57, 82, 88, 53, 74] and most recent studies [49, 120, 21] discuss checkpointing recovery scheme for failures in high performance computings (HPC). In general, checkpointing takes the snapshot of entire state of a program at the moment it was taken. It thus maintains all the necessary information for a process to restart from the checkpoint.

Upon an error, checkpoint recovery is initiated and the program is rolled back to the most recent checkpoint. The program state is overwritten with the stored checkpoint state. In this case, the computation back to the checkpoint is discarded and the system pays the performance overhead for error recovery. The more frequent the checkpoints are taken, the less the rollback overhead. However, checkpointing itself introduces penalty as it requires time and storage to generate the checkpoints. Book [53] presents an analytical model of checkpointing and

discusses in detail of checkpointing placement issue and its optimization. As a memory error is an uncommon event, failure recovery is rarely called.

In MemGuard, the checkpointing frequency is lower than that of error checking frequency as there is at least one error checking before checkpointing. There can be multiple memory error checking in between two neighbor checkpoints to detect errors timely. Otherwise, the system rollback overhead is high as error detection is delayed. In case of an error, the system is rolled back to a most recent checkpoint. If error still exists, the OS can improve checkpointing and integrity checking frequency to exactly capture errors in a shortened period. For repeating failures, a system reconfiguration is required as it is highly possible to be a hard error.

6.3.6 Integrity-Check Optimization and Other Discussions

The `integrity_check` step in Algorithm 2 requires to scan the entire memory space ever allocated to the process if the data block is not presented in cache. Although `integrity_check` period can be prolonged and it will not affect reliability much, the checking frequency may be limited by practical requirement, i.e. error detection is required before each checkpointing. In such cases, `integrity_check` can introduce visible overhead if the allocated memory space is significantly large. We further propose *lazy-scan* or also called *touched-only* scan scheme to reduce the scan overhead. The *lazy-scan* scheme only fetches pages that have been touched during an integrity checking period instead of all the pages allocated to the process. Typically, the required memory space is allocated at very beginning while merely part of them is touched during a period. Therefore, a *lazy-scan* will significantly reduce the memory traffic. Current processors can already provide the information to the OS.

In order to guarantee that all the allocated memory pages will be added into ReadHash for hash comparison, the hashes of all the untouched pages are still required. We thus propose to maintain a 128-bit (16-byte) sum hash for each memory page. In `integrity_check` step, hashes for untouched pages are added into ReadHash directly and hashes for touched pages are calculated based on the fetched data blocks from main memory. A 128-bit hash can be applied for each page or a group of pages to reduce the cost with penalty of possibly increased touched page size. The *lazy-scan* scheme has an additional advantage that it only detects

Real Machine Configuration	
Processor	Intel Xeon E5520 Quadcore 2.26GHz
OS kernel	Linux 2.6.27.6-117.fc10.x86_64
Compiler	GCC 4.6.2
L1 caches (per core)	32KB Inst/32KB Data, 8-way, 64B line
L2 caches (per core)	256KB unified, 8-way, 64B line
L3 cache (shared)	8MB, 8-way, 64B line
Memory	DDR3-1066 2DIMMs with 2GB/DIMM
Marss Simulator Configuration	
Processor	1 ooo core, 4GHz,14-stage pipeline
Functional units	2 IntALU, 4 LSU, 2 FPALU
IQ, ROB and LSQ	IQ 32, ROB 128, LQ 48, SQ 44
Physical registers	128 Int, 128 FP, 128 BR, 128 ST
L1 caches (per core)	64KB Inst/64KB Data, 2-way, 16B line
L2 cache (shared)	8MB, 8-way, 64B line
Memory	4GB, 200 cycles latency

Table 6.1: Major configuration parameters.

errors in program correction related pages and the untouched pages are assumed to be correct as they are not read from main memory again. This can reduce program recovery rate as OS may allocate a large memory space for some applications while the actually used is small. For example, 434.zeusmp from SPEC CPU2006 allocates 1131MB of virtual memory but only uses 502MB in its lifetime following our experiment in Section 6.5.

In practice, MemGuard can be applied in combination with an SECDED scheme as single-bit errors are most common. In this case, all single-bit errors can be corrected by SECDED and it reduces the overhead for checkpointing rollback recovery caused by single-bit errors. For example, high-performance computing servers can both adopt SECDED and MemGuard design to avoid most common single-bit errors and detect multiple-bit errors. With MemGuard design, it saves the computation overhead as errors can be efficiently detected. For consumer level computers and mobile systems without ECC, MemGuard design can be employed to efficiently provide error protection.

6.4 Experimental Methodologies

We build a memory request generator to generate synthetic traces of memory write and read for reliability evaluation. A configurable error injector is built to inject specific errors into the memory traces to evaluate the error detection capability of proposed MemGuard scheme and conventional SECDED design. SECDED is implemented following the reference design RD1025 [93] from Lattice Semiconductor and SpookyHash function is implemented using open source code [43]. To evaluate SECDED, we inject specific error types into each memory request and use SECDED to detect the error. The experiment is repeated by 1 billion times and then the error detection ratio is reported. For MemGuard, we generate 1 and 10 billion memory requests and inject specific errors to evaluate MemGuard error detection probability. The experiment is repeated for 100 times and the average error detection ratio is reported.

In order to evaluate system performance overhead introduced by MemGuard design, particularly from the `integrity_check` step, we run all 26 compilable benchmarks from SPEC CPU2006 suite [98] with different input-sets till the end on a real machine. The machine uses an Intel Xeon E5520 2.26GHz processor of 8MB last level cache and 4GB main memory. The detailed configuration is described in Table 6.1. We follow the study [29] to collect virtual and physical memory sizes for the total 51 benchmark-input sets to estimate memory scan overhead.

In addition, we use a full system simulator Marss-x86 [78] to further study the introduced memory traffic by MemGuard. Marss is an x86-64 architecture based cycle accurate simulator and its detailed configuration is also listed in Table 6.1. We run 26 benchmark-input sets on Marss for 10 billion instructions to collect memory traffic for baseline machine and estimate extra memory traffic caused by MemGuard.

6.5 Experimental Results

6.5.1 Reliability Study

We study error detection capability of MemGuard design and SECDED from two aspects. The analytical results are based on our error model and mathematical deduction presented in Section 6.3.3 while simulated results rely on synthetic memory traces and error injection.

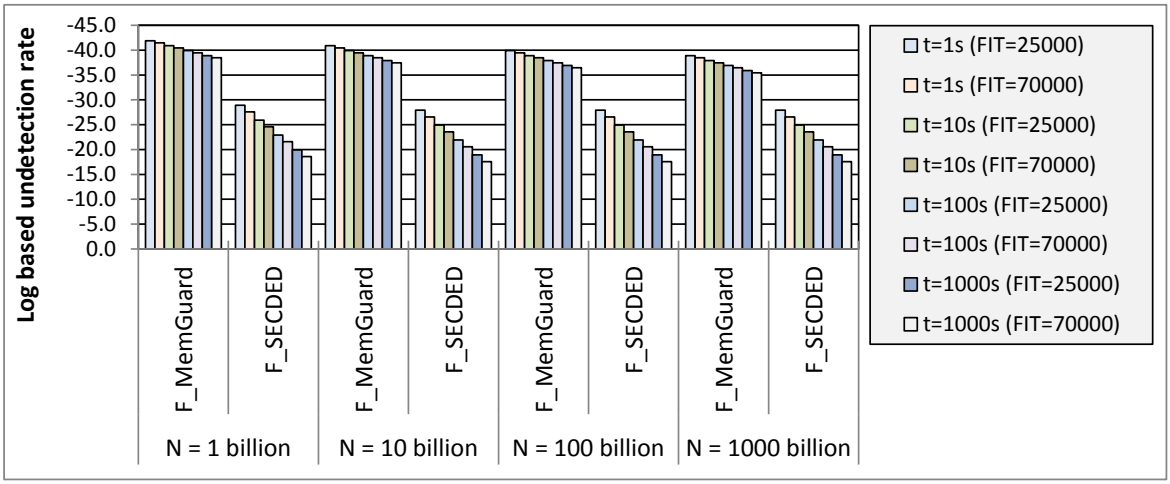


Figure 6.2: Error detection failure rate comparison of MemGuard and SECDED. The higher the bar the lower the error un-detectable rate.

6.5.1.1 Analytical Error Detection Rate

We calculate the error detection capability of MemGuard and SECDED design following the analytical model discussed in Section 6.3.3 for 25,000 and 70,000 FIT, respectively. In the calculation, we vary the time period t from 1 to 1,000 seconds and number of requests N from 1 to 1,000 billion. Figure 6.2 presents the results. The y-axis is reversed log-based to make the figure readable and thus the higher the bar the lower the error un-detectable rate.

It is obvious that for all these cases, our proposed MemGuard design is orders of magnitude stronger than conventional SECDED in error detection capability. The reason is that the strong hash function presents a low collision rate of $\frac{1}{2^{128}}$. With the number of requests in a checking period grows, the error detection capability decreases slightly due to possibly increased collision by addition operation. However, the error detection rate is mainly decided by collision rate of the selected hash and the decrease is gradually attenuated. Therefore, the error checking frequency of MemGuard design can be prolonged almost arbitrarily without much loss of reliability. As SECDED protection detects error in each request, its error detection rate almost keeps constant in this case.

As the time period t grows from 1 to 1,000 seconds, error rate for each bit in a data block grows, which increases multi-bit (> 2) error rate. Therefore, SECDED error detection

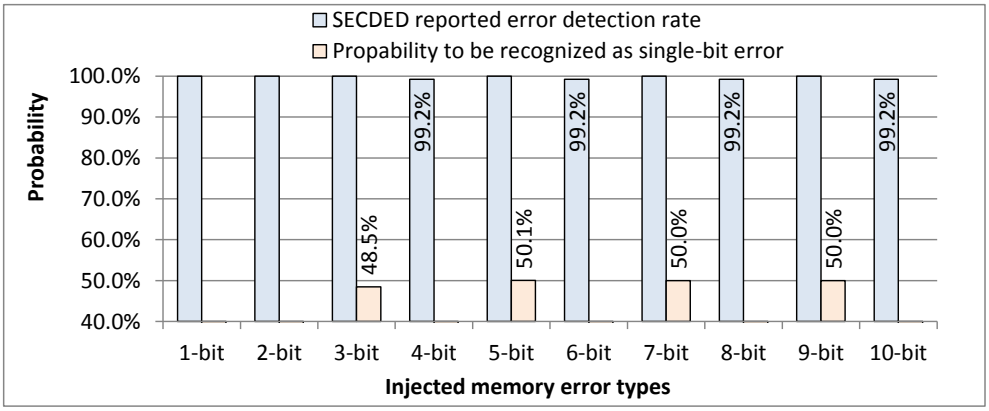


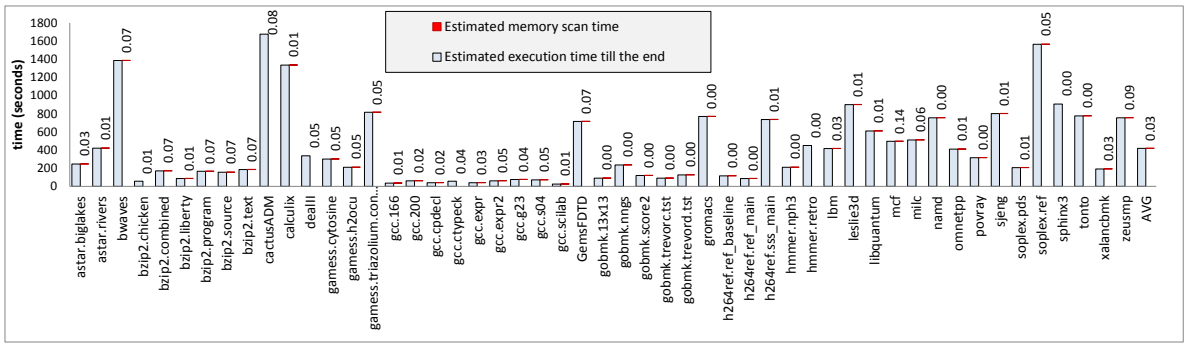
Figure 6.3: SECDED error protection capability. Note that although SECDED can fully report that error occurs for odd-bit (>1) errors, it might mistakenly recognize them as single-bit errors and correct them incorrectly.

capability is reduced significantly. However, in MemGuard, it employs strong hash function of which the collision rate is irrelevant with number of flips in a data block. As the error rate for each data block grows, error detection capability reduces slightly for MemGuard design since the number of tampered data blocks increases and thus the collision possibility. However, the decreasing rate is lower than that of SECDED.

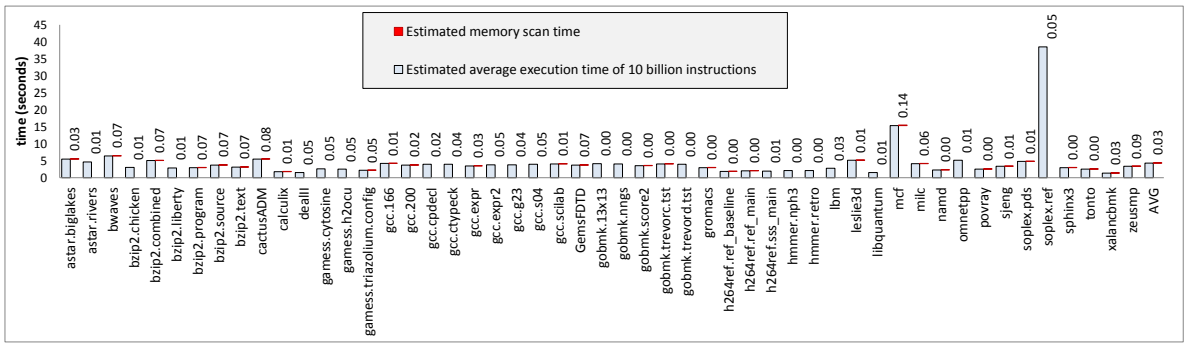
We do not present results data for Chipkill Correct as the computation exceeds the precision. However, we expect that MemGuard design can be stronger than Chipkill Correct. The reason is that the error detection capability of Chipkill Correct is closely related to multi-bit error rate while MemGuard design is not. As errors are usually correlated in real world [33], undetectable error rate can be high and Chipkill Correct may fail. As MemGuard design is highly correlated to the reliability of hash function itself, it can be more reliable than Chipkill Correct in error detection given that a strong hash function is carefully selected.

6.5.1.2 Simulated Error Detection Rate

For SECDED protection, we inject 1 to 10 bits of errors to a 64-bit data block at random positions and apply SECDED to detect the error. The experiment is repeated for a billion times to report error detection rate. Figure 6.3 presents the results. SECDED can completely detect all single- and double- bit errors as it has a Hamming distance of 4. In addition, the



(a) SPEC CPU2006 total execution time with memory scan overhead.



(b) SPEC CPU2006 execution time for 10 billion instructions with memory scan overhead.

Figure 6.4: Memory integrity-checking overhead of SPEC CPU2006 benchmark-input sets.

implemented SECDED code can detect all odd-bit errors similar to parity checking, which counts the total number of '0's or '1's in the data. Any odd-bit flip will modify the even/odd parity bit which can be detected. In theory, SECDED code cannot correct any errors with more than one bits flipped. The particular SECDED implementation reports if error occurred. If so, it simply identifies it is single-bit error or double-bit error no matter how many bits flipped in the data block. For odd-bit (except 1-bit) errors, they will be mistakenly recognized as double-bit errors with a 50% chance and the other 50% chance to be mistakenly recognized as single-bit errors and corrected by SECDED incorrectly. In addition, SECDED can not fully detect other even number of bits errors. The error detection rate is around 99.2% and all of them are mistakenly recognized as double-bit errors by implemented SECDED.

For MemGuard design, we inject one, two, three, four, five, ten, 100 and 1000 errors into 1 billion and 10 billion memory requests, separately. We group errors into six categories: single-, double-, triple-, quad-, multi- (randomly generated > 4) bit and mixed (of these five)

errors. The experiment is repeated for 100 times. We did not present the result figure as MemGuard design yields 100% error detection rate across all of our experiments. In reality, the probability of error detection is not 100% but is too high for a false negative to be observed in the experiments. Although the repeat times are limited to 100, we believe the results are representative as SpookyHash itself performs great in collision resistance (passing through $2^{72} \approx 10^{21}$ key pair test by the author).

The high reliability of MemGuard in memory error detection comes from the applied hash function. As illustrated, MemGuard design with SpookyHash is stronger than conventional SECDED error protection. A hash function with higher collision resistance can further improve its reliability. Given that Chipkill Correct may fail when multi-bit error rate is high, MemGuard design can be stronger than Chipkill Correct in error detection, and it is designed to be irrelevant to number of flips in a data block.

6.5.2 System Performance Study

In MemGuard design, the major overhead to system performance is to scan allocated memory space during integrity check. We thus estimate memory scan overhead and compare it to the program execution time. We run all the executable benchmark-input sets on a real machine to obtain the program execution time and collect the allocated virtual memory space following a previous study [29] to estimate memory scan latency. As memory scanning has great page locality, we assume that each 64-byte memory block takes 5ns for DDR3-1600 DRAM memory.

Figure 6.4 describes the results. Assume memory integrity checking is executed solely at the end of a program, the scan overhead is minimal. As illustrated in Figure 6.4a, the program execution time is orders of magnitude higher than memory scan time. On average, the program execution time is 418 seconds while memory scan takes merely 0.03 seconds given that the virtual memory space allocated is 372MB on average. In practice, the low integrity checking frequency can increase checkpointing recovery overhead as errors are delayed to detect. We thus assume the checking is activated every 10 billion instructions and Figure 6.4b presents the results. For SPEC 2006 benchmarks, the average execution time is 4.35 seconds for 10 billion instructions and memory scan overhead keeps unchanged, which is approximately 0.7%

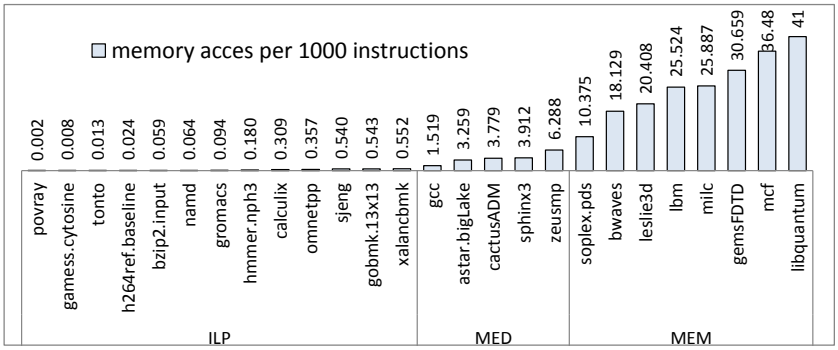
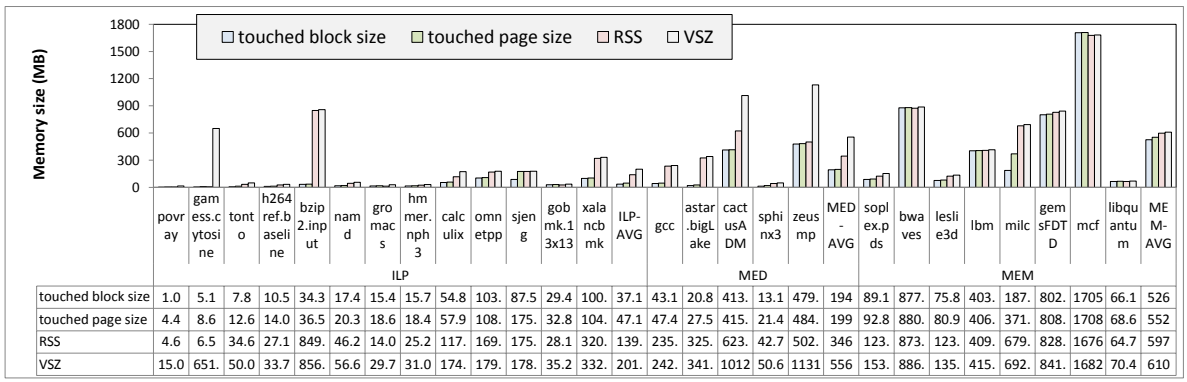


Figure 6.5: SPEC 2006 memory traffic characterizations.



touched block size	1.0	5.1	7.8	10.5	34.3	17.4	15.4	15.7	54.8	103.	87.5	29.4	100.	37.1	43.1	20.8	413.	13.1	479.	194	89.1	877.	75.8	403.	187.	802.	1705	66.1	526
touched page size	4.4	8.6	12.6	14.0	36.5	20.3	18.6	18.4	57.9	108.	175.	32.8	104.	47.1	47.4	27.5	415.	21.4	484.	199	92.8	880.	80.9	406.	371.	808.	1708	68.6	552
RSS	4.6	6.5	34.6	27.1	849.	46.2	14.0	25.2	117.	169.	175.	28.1	320.	139.	235.	325.	623.	42.7	502.	346	123.	873.	123.	409.	679.	828.	1676	64.7	597
VSZ	15.0	651.	50.0	33.7	856.	56.6	29.7	31.0	174.	179.	178.	35.2	332.	201.	242.	341.	1012	50.6	1131	556	153.	886.	135.	415.	692.	841.	1682	70.4	610

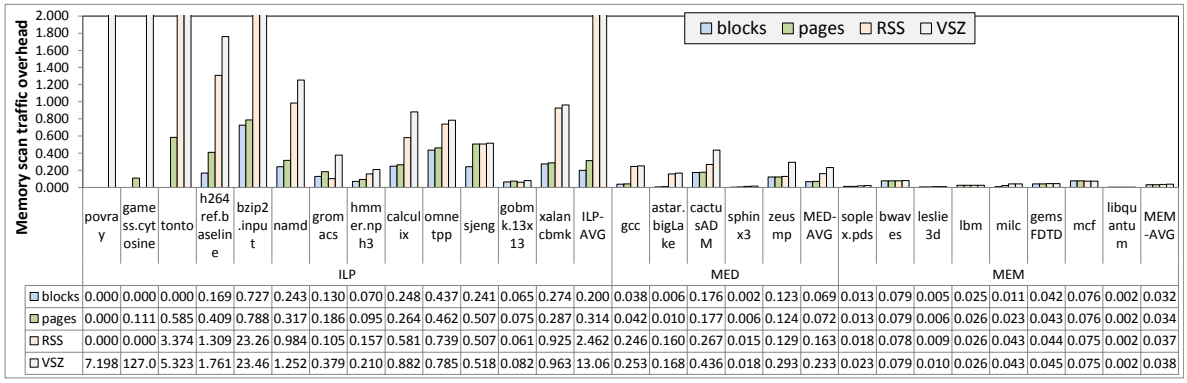
Figure 6.6: Comparison of touched memory blocks, touched memory pages in 10 billion instructions and used physical memory space and allocated virtual memory space during its lifetime.

of the execution time. Therefore, as long as the integrity checking rate is greater than 10 billion instructions, the system performance overhead is negligible.

In the case that the application has a large memory footprint and integrity checking frequency is required to be high by practice, *lazy-scan* scheme can help reduce scan overhead as only the touched pages are required to read from main memory. As the checking period is short, the touched page size is comparatively small. *lazy-scan* will significantly save the cost and we can see the results from next section.

6.5.3 Memory Traffic Overhead

We assume that the memory error checking frequency is every 10 billion instructions to study memory traffic overhead and proposed *lazy-scan* scheme. We first use the Marss-x86



	povray	gamess.cytosine	tonto	h264ref.baseline	bzip2.input	namd	gromacs	hmmer.nph3	calculix	omnetpp	sjeng	gobmk.13x13	xalancbmk	ILP-AVG	gcc	astar.bigLake	cactusADM	sphinx3	zeusmp	MED-AVG	soplex.pds	bwaves	leslie3d	lbm	milc	gemsFDTD	mcf	libquantum	MEM-AVG
blocks	0.000	0.000	0.000	0.169	0.727	0.243	0.130	0.070	0.248	0.437	0.241	0.065	0.274	0.200	0.038	0.006	0.176	0.002	0.123	0.069	0.013	0.079	0.005	0.025	0.011	0.042	0.076	0.002	0.032
pages	0.000	0.111	0.585	0.409	0.788	0.317	0.186	0.095	0.264	0.462	0.507	0.075	0.287	0.314	0.042	0.010	0.177	0.006	0.124	0.072	0.013	0.079	0.006	0.026	0.023	0.043	0.076	0.002	0.034
RSS	0.000	0.000	3.374	1.309	23.26	0.984	0.105	0.157	0.581	0.739	0.507	0.061	0.925	2.462	0.246	0.160	0.267	0.015	0.129	0.163	0.018	0.078	0.009	0.026	0.043	0.044	0.075	0.002	0.037
VSZ	7.198	127.0	5.323	1.761	23.46	1.252	0.379	0.210	0.882	0.785	0.518	0.082	0.963	13.06	0.253	0.168	0.436	0.018	0.293	0.233	0.023	0.079	0.010	0.026	0.043	0.045	0.075	0.002	0.038

Figure 6.7: MemGuard introduced memory traffic overhead by integrity checking.

simulator to characterize memory traffic of each SPEC 2006 benchmark without error protection. We group the benchmarks into three categories: ILP (computation-intensive), MED (medium) and MEM (memory-intensive) based on MPKI. We define MPKI as memory accesses per 1,000 instructions and ILP benchmarks are those with MPKI less than 1.0. MEM are those that MPKI is greater than 10.0 and MED are those in between the two. Figure 6.5 presents the results. Typically, MEM benchmarks are memory power hungry as they access memory frequently and consume significant operation power, read/write power and IO power.

Figure 6.6 compares the actually touched memory pages in 10 billion instructions, the used physical memory space (RSS: resident set size) and allocated memory space (VSZ: virtual memory size) during its entire lifetime. The physical and virtual memory spaces are mostly consistent with several large exceptions, i.e. gamess, zeusmp, etc. For gamess with cytosine workload, it allocates 651 MB virtual memory and the actual physical memory usage is merely 6.5MB. In addition, across all the benchmarks, the actually touched pages are smaller than total allocated memory space. The reduction is from 201.9MB for VSZ to 47.1MB, from 555.8MB to 199.3MB and from 609.8MB to 552.1MB on average for ILP, MED and MEM, respectively. Therefore, *lazy-scan* can effectively reduce memory traffic overhead.

Figure 6.7 presents the introduced memory traffic of MemGuard design. We limit the y-axis less than 2.0 to make the figure readable and table all data in case it is out of range. For ILP workloads, MemGuard design will introduce 13x memory traffic on average if entire VSZ is scanned. By scanning RSS and touched pages, the overhead is reduced to 2x and 31%,

respectively. As ILP workloads are computation intensive, their memory power consumption is not significant. For MEM workloads, the extra memory traffic is 3.8% and 3.7%, respectively, by scanning VSZ and RSS. The cost is further reduced to 3.4% if *lazy-scan* is applied. This cost is less than conventional SECDED design, which introduces 12.5% overhead. As memory scanning presents a good amount of page localities, the practical power consumption can be lower than traffic overhead. For MED workloads, the overhead is 23.3% by scanning VSZ and it can be reduced to 7.2% by *lazy-scan*. The overhead is also lower than conventional SECDED design. Note that the reported overhead from MemGuard is based on the assumption that the error checking is executed every 10 billion instructions. The checking period can be much longer than that, which will further reduce the overhead.

The figure also compares the actually touched memory data blocks with touched memory pages. On average, the introduced memory traffic can be reduced from 31% to 20%, from 7.2% to 6.9% and from 3.4% to 3.2% for ILP, MED and MEM workloads, respectively. By scanning touched blocks instead of touched pages, the overhead can be further reduced. Also, as discussed previously, scanning RSS can reduce the overhead compared to scanning VSZ.

6.6 Summary

We have presented MemGuard, a system level error protection scheme for main memory system. The scheme is independent of DRAM organizations and isolated in memory system level. Based on WriteHash and ReadHash comparison, the scheme can effectively detect errors in a sequence of memory requests. A detailed analysis of reliability capability and selection of hash functions are presented. The evaluation using mathematical deduction and synthetic simulation proves that MemGuard design is more reliable than conventional SECDED design with lower performance and power overhead given that a strong hash function is carefully selected. The MemGuard design is independent of main memory organizations and the scheme is reliable, effective and power efficient.

CHAPTER 7. CONCLUSION AND FUTURE WORK

Memory error has been a great concern for years. It may lead to severe consequences like data corruption, program and system crashes, and security vulnerabilities in worst case, without error protection. On the other hand, the performance and power consumption are other major considerations in memory system design. As conventional memory reliability design introduces significant storage, cost and power overhead, we develop novel error protection schemes for memory systems taking into considerations of reliability, power efficiency and system performance.

We first present E^3CC, a complete solution of memory error protection for sub-ranked and narrow-ranked low-power memories. It breaks the rigidity of conventional reliability design by embedding ECC into DRAM devices. In the design, we propose a novel address mapping scheme called BCRM to resolve the address mapping issue efficiently. The design is flexible, efficient, and compatible to conventional non-ECC DIMMs. Secondly, we further explore the address mapping schemes to support selective error protection, which selectively protect critical data only. It thus reduces the inherent overhead with uniform reliability design. All the proposed address mappings are based on modulo operation and they are proved to be efficient. Such mappings facilitate selective protection and thus further improve system power efficiency. Thirdly, we propose *Free ECC* design for compressed last-level cache. It embeds ECC/EDC into the unused fragments in compressed cache so that the dedicated storage is removed, which is required in conventional reliability design. The design thus improves cache power efficiency. In the end, we propose MemGuard design, a system level error protection scheme for main memory system. It is independent of DRAM organizations and isolated in memory system level. Based on incremental hashing scheme for a sequence of memory requests, MemGuard design is demonstrated to be stronger than SECDED in error detection. The scheme is reliable,

effective and power efficient.

All these proposed schemes to main memory protection are flexible and they require no modifications to motherboard or memory modules. They can be selectively applied to a wide spectrum of computer systems. For mobile systems like smartphones and tablets without error protection, the proposed E^3CC, selective error protection and MemGuard schemes can all be applied. As the designs are flexible and compatible with conventional non-ECC memories, the system can be booted with ECC protection for reliability or without ECC for power efficiency. For personal computers like desktop and laptops, all the schemes are efficient to provide error protection without upgrading motherboard to support expensive ECC-DIMM. For large-scale high-performance servers and datacenters, MemGuard design can be employed for stronger error detection capability. It detects error from a system level while introduces minimal overhead.

Throughout the dissertation, we have designed and demonstrated flexible, cost-effective, and power-efficient reliability design. In future, we can further explore our studies on real machines. First, we can evaluate our proposed address mapping schemes to study their effects on distributing memory requests. Second, we can implement the selective error protection on a real mobile system like a smartphone, through user-defined programming. We can mark the critical data and variables by analyzing the program source code and modify the compiler to facilitate placing the data into protected memory region. We can further build an error injector to test its system reliability. Similar to *Free ECC* design to last-level cache, we can explore the unused fragments with compression techniques in main memory systems.

Bibliography

[1] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Future scaling of processor-memory interfaces," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2009, pp. 1–12.

[2] J. H. Ahn, J. Leverich, R. S. Schreiber, and N. P. Jouppi, "Multicore DIMM: an energy efficient memory module with independently controlled DRAMs," *Computer Architecture Letters*, Vol. 8, No. 1, pp. 5–8, 2009.

[3] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," in *Tech. Report University of Wisconsin-Madison*, 2004.

[4] D. C. ans Srinivas Devadas, M. van Dijk, B. Gassend, and G. E. Suh, "Incremental multiset hash functions and their application to memory integrity checking," in *In Advances in Cryptology - Asiacrypt 2003 Proceedings, volume 2894 of LNCS*. Springer-Verlag, 2003, pp. 188–207.

[5] A. Appleby, "Murmurhash," https://sites.google.com/site/murmurhash/, 2011.

[6] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, Vol. 22, No. 3, pp. 258–266, 2005.

[7] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," in *Proceedings of International Cryptology Conference (CRYPTO)*, Vol. 839, 1994, pp. 216–233.

[8] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[9] S. Borkar, "The exascale challenge," 2011, keynote speech, PACT.

[10] L. Borucki, G. Schindlbeck, and C. Slayman, "Comparison of accelerated DRAM soft error rates measured at component and system level," in *Proceedings of IEEE International Reliability Physics Symposium (IRPS)*, 2008, pp. 482–487.

[11] C. Chen and M. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and Development*, Vol. 28, No. 2, pp. 124–134, 1984.

[12] G. Chen, M. Kandemir, M. J. Irwin, and G. Memik, "Compiler-directed selective data protection against soft errors," in *Proceedings of Design Automation Conference (DAC)*, 2005, pp. 713–716.

[13] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 1999, pp. 222–233.

[14] V. Degalahal, N. Vijaykrishnan, and M. J. Irwin, "Analyzing soft errors in leakage optimized SRAM design," in *Proceedings of International Conference on VLSI Design*, 2003, pp. 227–233.

[15] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," 1997.

[16] B. Diniz, D. Guedes, W. M. Jr., and R. Bianchini, "Limiting the power consumption of main memory," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2007, pp. 290–301.

[17] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proceedings of International Conference on Supercomputing (ICS)*, 2009, pp. 46–55.

[18] S. J. Eggers, F. Olken, and A. Shoshani, "A compression technique for large statistical data-bases," in *Proceedings of International Conference on Very Large Data Bases*, Vol. 7, 1981, pp. 424–434.

[19] P. G. Emma, W. R. Reohr, and M. Meterelliyoz, "Rethinking refresh: Increasing availability and reducing power in DRAM for cache applications," *IEEE Micro*, Vol. 28, No. 6, pp. 47–56, 2008.

[20] C. Estebanez, Y. Saez, G. Recio, and P. Isasi, "Performance of the most common non-cryptographic hash functions," in *Software: Practice and Experience*, 2013.

[21] K. Ferreire, J. Stearley, J. H. L. III, R. Oldfield, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, No. 44, 2011.

[22] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 78:1–78:12.

[23] G. Fowler, P. Vo, and L. C. Noll, "FNV hash," http://www.isthe.com/chongo/tech/comp/fnv/, 1991.

[24] F. Y. C. Fung, *A Survey of the Theory of Error-Correcting Codes*. Harvard-radcliff Math BUlletin, 2008.

[25] Q. Gao, "The Chinese remainder theorem and the prime memory system," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 1993, pp. 337–340.

[26] Google, "Cityhash 1.1," http://code.google.com/p/cityhash/, 2010.

[27] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *Proceedings of IEEE Symposium on Security and Privacy (ISSP)*, 2003, pp. 154–165.

[28] R. Hamming, "Error correcting and error detection codes," *Bell System Technical Journal*, Vol. 29, No. 2, pp. 147–160, 1950.

[29] J. Henning, "SPEC CPU2006 memory footprint," *ACM SIGARCH Computer Architecture News*, Vol. 35, No. 1, 2007.

[30] R. Housley, "A 224-bit one-way hash function: SHA-224," *RFC 3874*, Sep. 2004.

[31] M. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM Journal of Research and Development*, Vol. 14, No. 4, pp. 395–401, 1970.

[32] I. Hur and C. Lin, "A comprehensive approach to DRAM power management," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2008, pp. 305–316.

[33] A. A. Hwang, I. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design," in *Proceedings of International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 111–122.

[34] Hybrid Memory Cube Consortium, "Hybrid memory cube specification 1.0," 2013.

[35] Intel Inc., "Intel E7500 chipset MCH Intel x4 single device data correction (x4 SDDC) implementation and validation," 2002.

[36] Intel Inc., "The problem of power consumption in servers," *Technical Report*, 2012.

[37] Intel Inc., "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, 2014.

[38] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "Mcrengine: A scalable checkpointing system using data-aware aggregation and compression," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 17:1–17:11.

[39] ITRS, "Process integration, device, and structures," *International technology roadmap for semiconductors*, 2011 Edition, 2011.

[40] ITRS, "International technology roadmap for semiconductors," http://www.itrs.net/Links/2012ITRS/Home2012.htm, 2012.

[41] K. Jarvinen, M. Tommiska, and J. Skytta, "Hardware implementation analysis of the MD5 hash algorithm," in *Proceedings of Annual Hawaii International Conference on System Sciences (HICSS)*, 2005, p. 298a.

[42] B. Jenkins, "Hash functions for hash table lookup," http://www.burtleburtle.net/bob/hash/evahash.html, 2009.

[43] B. Jenkins, "Spookyhash: a 128-bit noncryptographic hash," http://www.burtleburtle.net/bob/hash/spooky.html, 2011.

[44] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2012, pp. 1–12.

[45] A. H. Johnston, "Scaling and technology issues for soft error rates," in *Proceedings of Annual Conference on Reliability*, 2000.

[46] P. Jones, "US secure hash algorithm 1 (SHA1)," *RFC 3174*, Sep. 2001.

[47] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: a DRAM page-mode scheduling policy for many-core era," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2011, pp. 24–35.

[48] Y. Katayama, E. J. Stuckey, S. Morioka, and Z. Wu, "Fault-tolerant refresh power reduction of DRAMs for quasi-nonvolatile data retention," in *Proceedings of International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, 1999, pp. 311–318.

[49] K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for HPC," in *Proceedings of International Conference for Distributed Computing Systems (ICDCS)*, 2012, pp. 615–626.

[50] S. Kim and A. K. Somani, "Area efficient architecture for information integrity in cache memories," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 1999, pp. 246–255.

[51] S. Kim, "Area-efficient error protection for caches," in *Proceedings of Design Automation and Test in Europe (DATE)*, 2006, pp. 1–6.

[52] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.

[53] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2007.

[54] D. H. Lawrie and C. R. Vora, "The prime memory system for array access," *ACM Transactions on Computers (TC)*, Vol. C-31, pp. 435–442, 1982.

[55] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens, "Eager writeback - a technique for improving bandwidth utilization," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2000, pp. 11–21.

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

[57] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, Vol. 5, pp. 874–879, 1994.

[58] L. Li and N. V. Vijay Degalahal, "Soft error and energy consumption interactions: A data cache perspective," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2004, pp. 132–137.

[59] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *Proceedings of the USENIX Conference on USENIX annual technical conference*, 2010, pp. 6–6.

[60] J. Lin, H. Zheng, Z. Zhu, E. Gorbatov, H. David, and Z. Zhang, "Software thermal management of DRAM memory for multicore systems," in *Proceedings of International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008, pp. 337–348.

[61] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Multu, "An experimental study of data retention behavior in modern DRAM devices: implications for retention time profiling mechanisms," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2013, pp. 60–71.

[62] J. Liu, B. Jaiyen, R. Veras, and O. Multu, "RAIDR: retention-aware intelligent DRAM refresh," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2012, pp. 1–12.

[63] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM refresh-power through critical data partitioning," in *Proceedings of International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 213–224.

[64] D. Locklear, "Chipkill correct memory architecture," 2000.

[65] "Reliability data sets," http://institutes.lanl.gov/data/fdata/, Los Alamos National Laboratory, 2011.

[66] S. Mathew, M. Anders, R. Krishnamurthy, and S. Borkar, "A 6.5GHz 54mW 64-bit parity-checking adder for 65nm fault-tolerant microprocessor execution cores," in *Proceedings of IEEE Symposium on VLSI Circuits*, 2007, pp. 46–47.

[67] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, Vol. 26, No. 1, pp. 2–9, 1979.

[68] M. Mehrara and T. Austin, "Exploiting selective placement for low-cost memory protection," *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 5, No. 3, 2008.

[69] Micron Technology, Inc., "DDR3 SDRAM MT41J256M8-32 Megx8x8Banks." http://www.micron.com/, 2006.

[70] Micron Technology, Inc., "DDR3 SDRAM system-power calculator," http://www.micron.com/, 2007.

[71] R. H. Morelos-Zaragoza, *The Art of Error Correcting Coding*, 2nd ed. Wiley, 2006.

[72] S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2005, pp. 243–247.

[73] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2013, pp. 72–83.

[74] X. Ni, E. Meneses, N. Jain, and L. V. Kale, "ACR: automatic checkpoint/restart for soft and hard error protection," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, No. 7, 2013.

[75] D. J. W. Noorlag, L. M. Terman, and A. G. Konheim, "The effect of alpha-particle-induced soft errors on memory systems with error correction," *IEEE Journal of Solid-State Circuits*, Vol. 15, No. 3, pp. 319–325, 1980.

[76] I. Oz, H. R. Topcuoglu, M. Kandemir, and O. Kandemir, "Reliability-aware core partitioning in chip multiprocessors," *Journal of System Architecture*, Vol. 58, No. 3-4, pp. 160–176, 2012.

[77] I. Oz, H. R. Topcuoglu, M. Kandemir, and O. Tosun, "Quantifying thread vulnerability for multicore architectures," in *Proceedings of Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2011, pp. 32–39.

[78] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86 CPUs," in *Proceedings of Design Automation Conference (DAC)*, 2011, pp. 5–9.

[79] P. K. Pearson, "Fast hashing of variable-length text strings," in *Proceedings of International Conference on Database Systems for Advanced Applications*, 1990.

[80] G. Pekhimenko, V. Seshadri, O. Multu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2012, pp. 377–388.

[81] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," in *In proceedings of IRE*, Vol. 49, 1960.

[82] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kinsley, "Memory exclusion: optimizing the performance of checkpointing systems," *Software: Practice and Experience*, Vol. 29, pp. 125–142, 1999.

[83] T. R. N. Rao and E. Fujiwara, *Error Control Coding For Computer Systems.* Prentice Hall, 1989.

[84] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Math*, Vol. 8, No. 2, 1960.

[85] R. Rivest, "The MD4 message-digest algorithm," *RFC 1320, MIT Laboratory for Computer Science and RSA Data Security, Inc.*, April 1992.

[86] R. Rivest, "The MD5 message-digest algorithm," *RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc.*, April 1992.

[87] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2000, pp. 128–138.

[88] J. C. Sancho, F. Petrini, G. Johnson, J. Fernandez, and E. Frachtenberg, "On the feasibility of incremental checkpointing for scientific computing," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2004, pp. 26–30.

[89] A. Satoh and T. Inoue, "ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS," in *Proceedings of International Conference on Information Technology: Coding and Computing (ITCC)*, Vol. 1, 2005, pp. 532–537.

[90] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2010, pp. 141–152.

[91] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, No. 1, 2007.

[92] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," in *Proceedings of International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Vol. 37, 2009, pp. 193–204.

[93] L. Semiconductor, "ECC module," *Reference Design 1025*, 2012.

[94] N. H. Seong *et al.*, "SAFER: Stuck-at-fault error recovery for memories," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2010, pp. 115–124.

[95] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "The visual vulnerability spectrum: characterizing architectural vulnerability for graphics hardware," in *Proceedings ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware*, 2006, pp. 9–16.

[96] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *Proceedings of International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2002, pp. 66–76.

[97] V. Sridharan and D. R. Kaeli, "Quantifying software vulnerability," in *Proc. workshop on Radiation effects and fault tolerance in nanometer technologies*, 2008, pp. 323–328.

[98] *SPEC CPU2006*, http://www.spec.org, Standard Performance Evaluation Corporation, 2011.

[99] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2003, pp. 339–350.

[100] N. Takagi, S. Kadowaki, and K. Takagi, "A hardware algorithm for integer division," in *Proceedings of IEEE Symposium on Computer Arithmetic*, 2005, pp. 140–146.

[101] M.-H. Teng, "Comments on 'the prime memory system for array access'," Vol. C-32, p. 1072, 1983.

[102] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.3," HP Laboratories, Tech. Rep., 2008.

[103] A. N. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi, "LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2012, pp. 285–296.

[104] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2010, pp. 175–186.

[105] F. A. Ware and C. Hampel, "Improving power and data efficiency with threaded memory modules," in *Proceedings of International Conference on Computer Design (ICCD)*, 2006, pp. 417–424.

[106] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, 1984.

[107] C. Wilderson, A. R. Alameldeen, Z. Chrishti, W. Wu, D. Somasekhar, and S. lien Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2010, pp. 83–93.

[108] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer, "An experimental study of security vulnerabilities caused by errors," in *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2001, pp. 421–430.

[109] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2000, pp. 258–265.

[110] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," in *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 431–436.

[111] D. H. Yoon and M. Erez, "Flexible cache error protection using an ECC FIFO," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, No. 49, 2009.

[112] D. H. Yoon and M. Erez, "Memory mapped ECC: Low-cost error protection for last level caches," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2009, pp. 116–127.

[113] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," in *Proceedings of International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 397–408.

[114] D. H. Yoon and M. Erez, "Virtualized ECC: Flexible reliability in main memory," *IEEE Micro*, Vol. 31, No. 1, pp. 11–19, 2011.

[115] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, "FREE-p: protecting non-volatile memory against both hard and soft errors," in *Proc. HPCA*, 2011.

[116] W. Zhang, "Enhancing data cache reliability by the addition of a small fully-associative replication cache," in *Proceedings of International Conference on Supercomputing (ICS)*, 2004, pp. 12–19.

[117] W. Zhang, "Replication cache: A small fully associative cache to improve data cache reliability," in *ACM Transactions on Computers (TC)*, Vol. 54, No. 12, 2005, pp. 1547–1555.

[118] W. Zhang, S. Gurumurthi, and M. Kagdemir, "ICR: In-cache replication for enhancing data cache reliability," in *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2003, pp. 291–300.

[119] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2000, pp. 32–41.

[120] G. Zheng, X. Ni, and L. V. Kale, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *Proceedings of International Conference for Dependable System and Networks Workshops (DSN-W)*, 2012, pp. 1–6.

[121] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu, "Mini-rank: adaptive DRAM architecture for improving memory power efficiency," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2008, pp. 210–221.

[122] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," Vol. 23, No. 3, pp. 337–343, 1977.

[123] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," Vol. 24, No. 5, pp. 530–536, 1978.