UNIVERSITY OF CALIFORNIA

Los Angeles

Opportunistic Memory Systems
in Presence of Hardware Variability

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Electrical Engineering

by

Mark William Gottscho

2017

ABSTRACT OF THE DISSERTATION

Opportunistic Memory Systems
in Presence of Hardware Variability

by

Mark William Gottscho
Doctor of Philosophy in Electrical Engineering
University of California, Los Angeles, 2017
Professor Puneet Gupta, Chair

The memory system presents many problems in computer architecture and system design. An important challenge is worsening hardware variability that is caused by nanometer-scale manufacturing difficulties. Variability particularly affects memory circuits and systems – which are essential in all aspects of computing – by degrading their reliability and energy efficiency. To address this challenge, this dissertation proposes Opportunistic Memory Systems in Presence of Hardware Variability. It describes a suite of techniques to opportunistically exploit memory variability for energy savings and cope with memory errors when they inevitably occur.

In Part 1, three complementary projects are described that exploit memory variability for improved energy efficiency. First, ViPZonE and DPCS study how to save energy in off-chip DRAM main memory and on-chip SRAM caches, respectively, without significantly impacting performance or manufacturing cost. ViPZonE is a novel extension to the virtual memory subsystem in Linux that leverages power variation-aware physical address zoning for energy savings. The kernel intelligently allocates lower-power physical memory (when available) to tasks that access data frequently to save overall energy. Meanwhile, DPCS is a simple and low-overhead method to perform Dynamic Power/Capacity Scaling of SRAM-based caches. The key idea is that certain memory cells fail to retain data at a given low supply voltage; when full cache capacity is not needed, the voltage is opportunistically reduced and any failing cache blocks are disabled dynamically. The third project in Part 1 is X-Mem: a new extensible memory characterization tool. It is used in a series of case studies on a cloud server, including one where the potential benefits of variation-aware DRAM latency tuning are evaluated.

Part 2 of the dissertation focuses on ways to opportunistically cope with memory errors whenever they occur. First, the Performability project studies the impact of corrected errors in memory on the performance of applications. The measurements and models can help improve the availability and performance consistency of cloud server infrastructure. Second, the novel idea of Software-Defined Error-Correcting Codes (SDECCs) is proposed. SDECC opportunistically copes with detected-but-uncorrectable errors in main memory by combining concepts from coding theory with an architecture that allows for heuristic recovery. SDECC leverages available side information about the contents of data in memory to essentially increase the strength of ECC without introducing significant hardware overheads. Finally, a methodology is proposed to achieve Virtualization-Free Fault Tolerance (ViFFTo) for embedded scratchpad memories. ViFFTo guards against both hard and soft faults at minimal cost and is suitable for future IoT devices.

Together, the six projects in this dissertation comprise a complementary suite of methods for opportunistically exploiting hardware variability for energy savings, while reducing the impact of errors that will inevitably occur. Opportunistic Memory Systems can significantly improve the energy efficiency and reliability of current and future computing systems. There remain several promising directions for future work.

The dissertation of Mark William Gottscho is approved.

Glenn D. Reinman

Mani B. Srivastava

Lara Dolecek

Puneet Gupta, Committee Chair

University of California, Los Angeles

2017

To my wife, who has inspired me,

and to whom I have vowed to expand our horizons.

TABLE OF CONTENTS

List of Figures

List of Tables

ACKNOWLEDGMENTS

This dissertation is the culmination of many years of hard work. Countless late nights, weekends, and vacation days were spent thinking, reading, modeling, designing, coding, debugging, evaluating, and writing about my research ideas – not necessarily in that order – often at the unfortunate expense of maintaining my relationships with people. Navigating the waters of research in engineering always seemed fraught with risk and resulted in many struggles. But weathering each storm in my PhD inevitably paid off. Certainly, a wide spectrum of emotions were experienced throughout the course of completing my research, including the elation of newfound understanding and the satisfaction of prototyping. I owe gratitude to a number of people and organizations, without whom my doctoral studies would not have been possible.

First, and most importantly, I wish to thank my doctoral committee at UCLA: Profs. Puneet Gupta (chair and advisor), Lara Dolecek, Mani Srivastava (all from the Electrical Engineering department), and Prof. Glenn Reinman (from the Computer Science department). I thank each of them for their outstanding teaching, long-term research guidance, and for constantly pushing me to do better. Much of their precious time has been spent guiding, critiquing, and vetting the quality of my work (even the tiniest of details). Prof. Reinman introduced me to interesting research topics in computer architecture. Discussions from his graduate-level special topics course have inspired me to explore new directions in my professional career. Prof. Srivastava has supported and mentored me in my transition from an undergraduate student at UCLA into a PhD candidate under the umbrella of the multi-year and multi-institutional Variability Expedition project that was funded by the NSF (of which Prof. Srivastava was a leading PI). He also helped supervise my research on a battery charging-aware mobile device project (which is outside the scope of this dissertation). Prof. Dolecek has been an instrumental mentor and collaborator in the SDECC project, which I consider to be the key achievement of my dissertation. She helped introduce me to the fundamentals of coding theory, which were essential to our collaboration and were initially far outside of my intellectual comfort zone. Finally, Prof. Gupta has been a fantastic advisor and mentor ever since he generously took me under his wing when I was an undergraduate student in the winter of 2011. Without him, I would not have grown into the researcher and engineer that I am today. In the beginning of my graduate studies, he warned me that the PhD degree would not be easy and certainly not a guaranteed achievement, and that it would require a very high level of commitment. But he assured me that the intellectual rewards would be worth it. He was right!

My work would not have been possible without funding from several sources. From 2011 until the first half of 2016, the NSF supported me as a Graduate Student Researcher at UCLA through the Variability Expedition (grant no. CCF-1029783). Much of the research comprising Chapters 2 and 3 were drawn from collaborations with Prof. Nikil Dutt and Prof. Alex Nicolau from the University of California at Irvine (UCI), as well as Dr. Luis A. D. Bathen and Dr. Abbas BanaiyanMofrad who have since graduated from UCI. Our partnership was facilitated by the Variability Expedition. Thanks to each of my UCI collaborators, who provided valuable input during the formative years of my PhD.

In the summers of 2014 and 2015, I was humbled to join Microsoft Research (MSR) in Redmond, WA as a PhD intern in Dr. Jie Liu's Sensing and Energy Research Group and working with Microsoft's Cloud Server Infrastructure (CSI) team. The works performed in these months were the basis for research that continued when I returned to school and now comprise Chapters 4 and 5 in this dissertation. I wish to thank Dr. Mohammed Shoaib from MSR and Dr. Sriram Govindan from CSI for taking a personal interest in mentoring me on my internship projects and beyond on my professional development. I also had the pleasure of collaboration and support from other talented and experienced folks during my stints at Microsoft. Thanks to Drs. Bikash Sharma and Di Wang for their direct research guidance and contributions. Mike Andrewartha, Mark Santaniello, and Dr. Laura Caulfield each provided valuable input on several technical aspects of my work. Thank you to Drs. Adrian Sampson (now at Cornell) and Doug Burger, who included me in fantastic discussions as part of a fantastic computer architecture paper reading group. Drs. Jie Liu, Badriddine Khessib, and Kushagra Vaid fundamentally made my internships possible.

Many thanks to Qualcomm and UCLA for generously supporting my research for the 2016-2017 academic year. The Qualcomm Innovation Fellowship (QInF, $100K award) supported both Clayton Schoeny (currently a PhD candidate and my close collaborator at UCLA) and I to pursue our research proposal for the design of novel Software-Defined Error-Correcting Codes (SDECC). Research on SDECC has resulted in all of Chapter 6 and half of Chapter 7; it has also spawned numerous ideas for future work. Thanks to Dr. Greg Wright from Qualcomm Research for mentoring Clayton and I during the fellowship. I also wish to thank UCLA for awarding me the 2016-2017 Dissertation Year Fellowship (DYF, $20K stipend plus tuition award), to which I had applied and was subsequently awarded concurrently with QInF. The DYF enabled me to complete the other half of Chapter 7 and complete this dissertation.

I wish to thank Irina Alam, Dr. Shaodi Wang (PhD'17), Dr. Liangzhen Lai (PhD'15), Yasmine Badr, Saptadeep Pal, Weiche Wang, Dr. Abde Ali Kagalwalla (PhD'14), Dr. John Lee (PhD'12), and Dr. Rani Ghaida (PhD'12) from the UCLA NanoCAD Lab, as well as Salma Elmalaki and Dr. Lucas Wanner (PhD'14) from the UCLA Networked and Embedded Systems Laboratory for their friendship and research companionship over the years. Thanks also to Dr. Fedor Pikus from Mentor Graphics, who graciously volunteered his time over many weeks to teach the NanoCAD Lab expert-level topics in writing software for high-performance computers.

The staff of the UCLA Electrical Engineering department have been outstanding. Few graduate students, myself included, would have survived the program if it were not for the hard work and compassion of the Office of Student Affairs staff in particular: Deeona Columbia, Ryo Arreola, Julio Vega Romero, as well as previous staffers Mandy Smith and Michelle Welch. Sandra Bryant has also helped to procure lab equipment and assisted with conference travels.

I appreciate the invitations in 2016 by Prof. Jörg Henkel from the Karlsruhe Institute of Technology and Prof. Subhasish Mitra from Stanford to present work from this dissertation. Thanks also to Dr. Greg Wright for inviting me to present at Qualcomm Research (San Diego) in 2017. Their comments on my work have each added valuable perspectives.

Thanks also to my best friends, who have remained so even with months or years in between the making of quality memories (which sometimes had to be put on hold so that I could push through my research). They have kept me rooted in reality and provided me the perspective to continually rediscover who I am — always forcing me to keep one foot outside of the ivory tower.

I am deeply grateful to my mother (Katherine Paszkolovits), father (Dr. Richard Gottscho), brother (Dr. Andrew Gottscho) for setting incredibly high bars for educational and intellectual achievement. My wife (Dr. Maria Yefimova) not only set a high bar for accomplishments. She has also been alongside me every single day of my PhD: soothing countless frustrations, critiquing my ideas, proofreading my papers, and helping me rehearse my talks. Maria continues to deeply inspire me. I am most grateful to my family for their unwavering support, for exemplifying the meaning of human compassion, and for always believing in me.

Copyrights and Re-use of Published Material

This dissertation contains significant material that has been previously published or is intended to be published in the future. Chapter 2 is mostly based on content that was published in the IEEE Transactions in Computers (TC) in 2015 [1]. The chapter contains an additional part in Sec. 2.3 that is based on a short paper in the IEEE Embedded Systems Letters (ESL) from 2012 [2]. The ideas described in Chapter 2 have also been discussed in other publications in which I am a contributing author [3–7], but that content has not been included in this dissertation.

The material on DPCS, covered in Chapter 3, is based on a journal article published in the ACM Transactions on Architecture and Code Optimization (TACO) in 2015 [8]. DPCS concepts have also been described in other publications of mine [7, 9, 10], but again, that content is not included here.

Chapter 4 discusses X-Mem, which is based on material that appeared at the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) in 2016 [11].

Finally, material that was published in the IEEE Computer Architecture Letters (CAL) in 2016 [12] re-appears in roughly the first half of Chapter 5. Most of the content in Chapters 6 and 7 is being prepared for publication. The concepts are loosely based on preliminary work that first appeared at the Silicon Errors in Logic – System Effects (SELSE) workshop [13] and then re-appeared at the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W) [14].

The copyrights on published research that re-appears in this dissertation is with either the IEEE and/or the ACM where appropriate. The respective copyright agreements allow for derivative works by the author with attribution, so no explicit permission was required for inclusion of material in this dissertation. The titles of each chapter have been changed somewhat to differentiate them from the published versions of the respective manuscripts where applicable.

Much of the work in my PhD was conducted in collaboration with various individuals. Only the research for which I led appears in this dissertation. Publications and other work to which I have contributed, but did not lead, are not included in the body of this dissertation.

Open-Source Code and Datasets

All material artifacts that were created in this dissertation are available online. The following table lists the locations of source code repositories for each chapter. Datasets for each chapter are available online at the UCLA NanoCAD Lab website: `http://nanocad.ee.ucla.edu`.

Project	**Chapter**	**URL**
ViPZonE	2	`https://github.com/nanocad-lab?&q=vipzone`
DPCS	3	`https://github.com/nanocad-lab?&q=dpcs`
X-Mem	4	`https://nanocad-lab.github.io/X-Mem/`
		`https://github.com/Microsoft/X-Mem`
		`https://github.com/nanocad-lab?&q=xmem`
Performability	5	`https://github.com/nanocad-lab?&q=performability`
SDECC	6	`https://github.com/nanocad-lab?&q=sdecc`
ViFFTo	7	`https://github.com/nanocad-lab?&q=viffto`

VITA

2009, 2010	Co-op, NASA Dryden Flight Research Center
2011	BS Degree, UCLA Electrical Engineering
2012	Intern, Altera
2012	PhD Department Fellow, UCLA Electrical Engineering
2014	MS Degree with Great Distinction, UCLA Electrical Engineering
2014	Honorable Mention, NSF Graduate Research Fellowship Program
2014, 2015	PhD Intern, Microsoft Research
2016	Qualcomm Innovation Fellow
2016	UCLA Dissertation Year Fellow

PUBLICATIONS

Mark Gottscho, Mohammed Shoaib, Sriram Govindan, Bikash Sharma, Di Wang, and Puneet Gupta. "Measuring the Impact of Memory Errors on Application Performance," in *IEEE Computer Architecture Letters (CAL)*, 4 pages. Pre-print available online August 2016. DOI: 10.1109/LCA.2016.2599513

Mark Gottscho, Clayton Schoeny, Lara Dolecek, and Puneet Gupta. "Software-Defined Error-Correcting Codes," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 276-282, Best of SELSE Special Session. Toulouse, France, June 2016. ISBN: 978-1-5090-3688-2, DOI: 10.1109/DSN-W.2016.67

Mark Gottscho, Sriram Govindan, Bikash Sharma, Mohammed Shoaib, and Puneet Gupta. "X-Mem: A Cross-Platform and Extensible Memory Characterization Tool for the Cloud," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 263-273. Uppsala, Sweden, April 2016. ISBN: 978-1-5090-1953-3, DOI: 10.1109/ISPASS.2016.7482101

Mark Gottscho, Clayton Schoeny, Lara Dolecek, and Puneet Gupta. "Software-Defined Error-Correcting Codes," in *Silicon Errors in Logic – System Effects (SELSE)* workshop, 6 pages. Austin, Texas, USA, March 2016.

Qixiang Zhang, Liangzhen Lai, **Mark Gottscho**, and Puneet Gupta. "Multi-Story Power Distribution Networks for GPUs," in *IEEE Design, Automation, and Test in Europe (DATE)*, pp. 451-456, Dresden, Germany, March 2016. ISBN: 978-3-9815-3707-9

Mark Gottscho, Abbas BanaiyanMofrad, Nikil Dutt, Alex Nicolau, and Puneet Gupta. "DPCS: Dynamic Power/Capacity Scaling for SRAM Caches in the Nanoscale Era," in *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 12, No. 3, Article 27, 26 pages. Pre-print available online August 2015, in print October 2015. EISSN: 1544-3973, DOI: 10.1145/2792982

Lucas Wanner, Liangzhen Lai, Abbas Rahimi, **Mark Gottscho**, Pietro Mercati, Chu-Hsiang Huang, Frederic Sala, Yuvraj Agarwal, Lara Dolecek, Nikil Dutt, Puneet Gupta, Rajesh Gupta, Ranjit Jhala, Rakesh Kumar, Sorin Lerner, Subhashish Mitra, Alexandru Nicolau, Tajana Simunic Rosing, Mani B. Srivastava, Steve Swanson, Dennis Sylvester, and Yuanyuan Zhou. "NSF Expedition on Variability-Aware Software: Recent Results and Contributions," in *De Gruyter Information Technology (it)*, Vol. 57, No. 3, pp. 181-198. Invited paper. Pre-print available online June 2015. DOI: 10.1515/itit-2014-1085

Salma Elmalaki, **Mark Gottscho**, Puneet Gupta, and Mani Srivastava. "A Case for Battery Charging-Aware Power Management and Deferrable Task Scheduling," in *USENIX Workshop on Power-Aware Computing and Systems (HotPower)*, 6 pages. Bloomfield, Colorado, USA, October 2014.

Mark Gottscho, Abbas BanaiyanMofrad, Nikil Dutt, Alex Nicolau, and Puneet Gupta. "Power / Capacity Scaling: Energy Savings With Simple Fault-Tolerant Caches," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 6 pages. San Francisco, California, USA, June 2014. ISBN: 978-1-4503-2730-5, DOI: 10.1145/2593069.2593184

Nikil Dutt, Puneet Gupta, Alex Nicolau, **Mark Gottscho**, and Majid Shoushtari. "Multi-Layer Memory Resiliency," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 6 pages. Invited paper. San Francisco, California, USA, June 2014. ISBN: 978-1-4503-2730-5, DOI: 10.1145/2593069.2596684

Mark Gottscho, Luis A. D. Bathen, Nikil Dutt, Alex Nicolau, and Puneet Gupta. "ViPZonE: Hardware Power Variability-Aware Virtual Memory Management for Energy Savings," in *IEEE Transactions on Computers (TC)*, Vol. 64, No. 5, pp. 1483-1496. Pre-print available online June 2014, in print May 2015. ISSN: 0018-9340, DOI: 10.1109/TC.2014.2329675

Nikil Dutt, Puneet Gupta, Alex Nicolau, Luis A. D. Bathen, **Mark Gottscho**. "Variability-Aware Memory Management for Nanoscale Computing," in *Proceedings of the ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 125-132. Invited paper. Yokohama, Japan, January 2013. ISBN: 978-1-4673-3029-9, ISSN: 2153-6961, DOI: 10.1109/ASPDAC.2013.6509584

Luis A. D. Bathen, **Mark Gottscho**, Nikil Dutt, Alex Nicolau, and Puneet Gupta. "ViPZonE: OS-Level Memory Variability-Aware Physical Address Zoning for Energy Savings," in *Proceedings of the ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 33-42. Tampere, Finland, October 2012. ISBN: 978-1-4503-1426-8, DOI: 10.1145/2380445.2380457

Mark Gottscho, Abde Ali Kagalwalla, and Puneet Gupta. "Power Variability in Contemporary DRAMs," in *IEEE Embedded Systems Letters (ESL)*, Vol. 4, No. 2, pp. 37-40. Pre-print available April 2012, in print June 2012. ISSN: 1943-0663, DOI: 10.1109/LES.2012.2192414

CHAPTER 1

Introduction

Memories are a key bottleneck in the energy efficiency, performance, and reliability of many computing systems. Hardware variability that stems from the manufacturing process and the operating environment is a root cause of many challenges in memory system design and operation. To contextualize and motivate the research in this dissertation, a brief and broad overview of memory challenges and solutions is provided in Sec. 1.1. Hardware variability in the nanoscale era is identified as a major issue for memories in Sec. 1.2. *Opportunistic Memory Systems in Presence of Hardware Variability* provide a useful arsenal for combating this problem, and is described in Sec. 1.3. The research in this dissertation complements a wide variety of existing techniques for coping with some of the grand challenges that computing faces today and tomorrow.

1.1 Memory Challenges in Computing

The memory bottleneck fundamentally stems from the Von Neumann stored-program computing model [15] wherein computing is broken down into distinct logic and memory components. As computing systems scaled over the decades, there has always been a need for faster computational units which must be able to quickly and randomly access an ever-larger amount of information. Computer architects have responded by optimizing the logical units for maximum performance, and the main memory for maximum information density, which naturally comes at the cost of reduced speed [16]. In fact, memory devices and circuits need not be inherently much slower than logic; rather, this is a natural consequence of system designs that have evolved over time, where areal efficiency of memory cells is paramount.

The computer architecture community has strived to mitigate the logic-memory performance gap through the addition of several small and fast caches, which has created today's *memory hierarchy* [16–18]. Over the years, the native performance gap between logic and memory grew by orders of magnitude [16, 19]; nevertheless, the memory hierarchy approach proved to be effective at improving both performance and energy efficiency while allowing the information capacity of computers to continue growing alongside advances in VLSI design and manufacturing technology. For years, the memory bottleneck was kept at bay.

Unfortunately, the traditional formula for improving performance and energy efficiency of general-purpose computers reached its limits in the early-to-mid-2000s [16, 19]. Dennard scaling [20] – a technology trend where chip areal density improvements gifted performance benefits without worsening chip power density – was no longer possible. Several non-ideal transistor effects became critical in the deep submicron regime (around the 100 nm manufacturing process node). Transistor threshold voltages and supply voltages could no longer continue scaling down without causing an exponential rise in leakage power [21]. Among other consequences, this caused transistor power density to worsen with each successive technology node [16,19,22]. Making matters worse, an increasing fraction of the overall chip power budget was being consumed by static power, primarily in the form of leakage currents [16, 19, 21, 22]. Accompanying the power density problems, chip interconnects became a major limitation on the complexity of logic and the cost of data movement [16] because wire signal delay does not scale with feature size [21], and because wire congestion limited the routability of designs [23]. These issues had a significant impact on the design of processors, but also exacerbated the memory bottleneck: traditional large memories require long wires (stangling access latency) and consume considerable static power (reducing the share of power available to computation).

These challenges meant that Moore's Law [24] would continue delivering cost improvements in integrated circuits, but they would not be directly accompanied by performance benefits. This was a call for significant innovation across the computing stack, from circuits all the way up to software.

In an effort to solve these problems, in the mid-2000s the industry moved to multicore processor architectures with somewhat simpler cores to continue performance and efficiency scaling [16,18]. Accompanying multicore processors, the memory hierarchy evolved to keep pace, with coherent

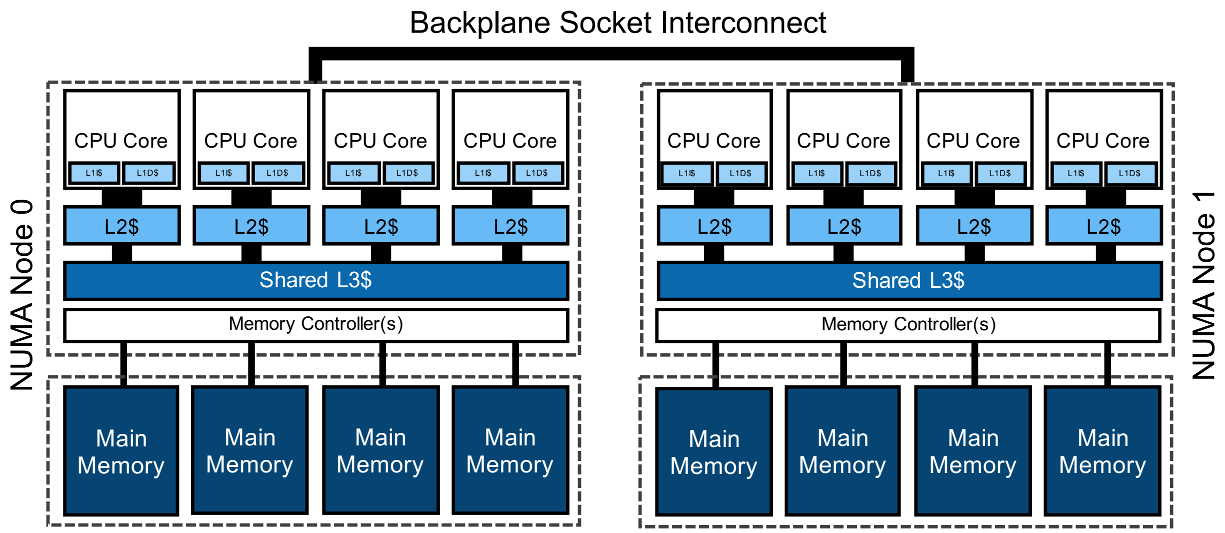


Figure 1.1: Illustration of a typical dual-socket server memory hierarchy with shared L3 caches.

shared caches, distributed caches, standard high-bandwidth off-chip memory interfaces [25, 26], sophisticated latency-hiding techniques [16], and non-uniform cache (NUCA) [27] and memory (NUMA) architectures [28], among many other techniques (a typical modern memory hierarchy is illustrated at a high level in Fig. 1.1). Meanwhile, device and process technologists worked to continue scaling VLSI systems through a myriad of innovations for addressing power density and leakage current, such as strained silicon [29], silicon-on-insulator (SOI) [30], high-*k* metal-gate (HKMG) [31], and FinFETs [32]. The multicore approach was successful at delivering performance and energy efficiency improvements for the past decade, but once again, the industry has become limited by power density in logic and communication bottlenecks to the various system memories [33].

Low power density is key to achieving the vision of both exascale computing and the Internet of Things (IoT) [34]. The computer architecture community has understood that systems must adopt intelligent and dynamic power-aware strategies to keep power density in check. Popular techniques can be categorized into active management (e.g., dynamic voltage/frequency scaling, or DVFS [35]) and idle power-down techniques (e.g., clock gating [36, 37] and power gating [38]). Generally, idle power-down techniques can save more power than active management methods, but they usually incur time and energy penalties to transition between states and prevent any work from being done while idle. Because modern systems often perform work in bursts with short

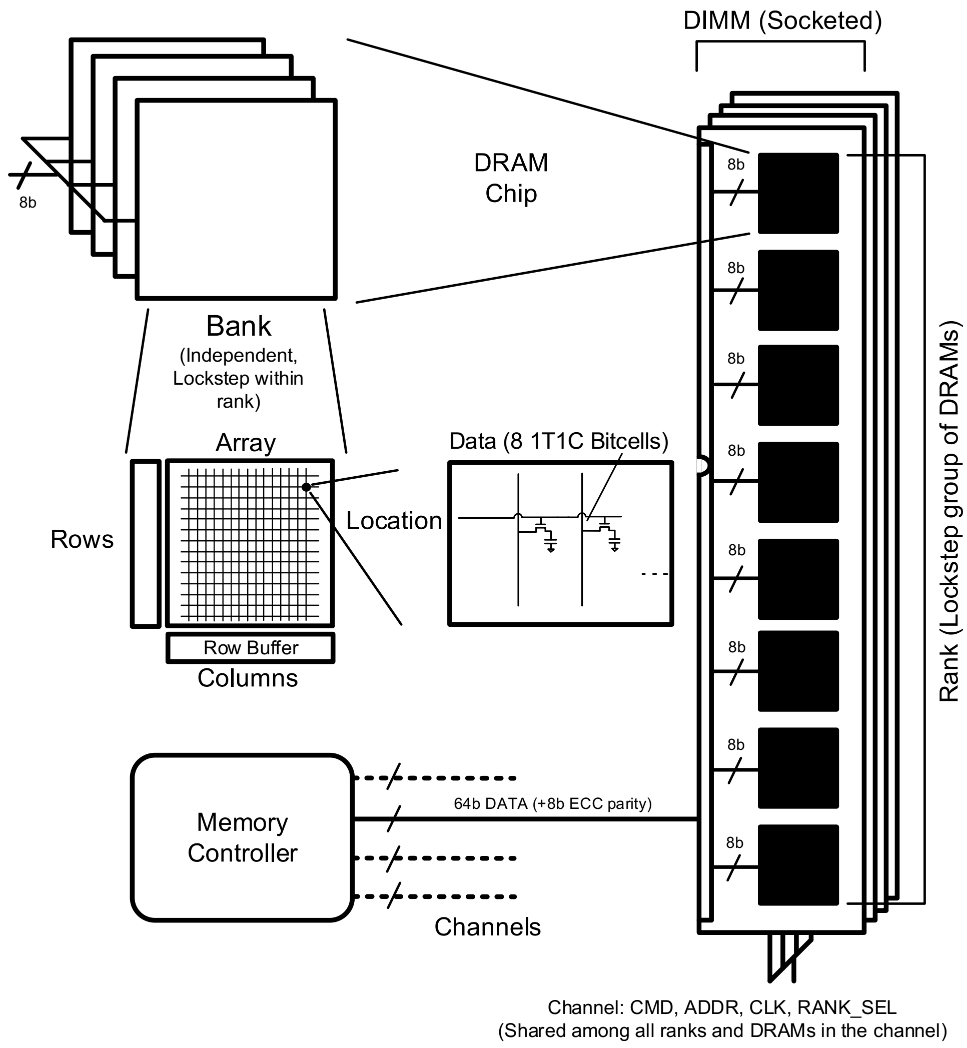

DIMM (Socketed)
DRAM Chip
8b
Bank
(Independent, Lockstep within rank)
Array
Rows
Row Buffer
Columns
Location
Data (8 1T1C Bitcells)
Rank (Lockstep group of DRAMs)
Memory Controller
64b DATA (+8b ECC parity)
Channels
Channel: CMD, ADDR, CLK, RANK_SEL
(Shared among all ranks and DRAMs in the channel)

Figure 1.2: Illustration of a typical DDR3 DRAM memory subsystem, showing independent channels, DIMMs, ranks, banks, rows, and columns.

idle periods, active management techniques are preferred in order to improve energy proportionality [39, 40]. Memories, however, pose particular challenges for power management because they have had poor energy proportionality [40] and increasingly consume a significant fraction of the overall system power budget [33,41,42]. In conventional large and volatile memories, static power often dominates simply because the memories need to retain a large amount of state. Yet memories are not as amenable to active and idle power management techniques as processors. Moreover, off-chip DRAM memory interfaces – an example of which is illustrated in Fig. 1.2 – are standardized through the Joint Electron Device Engineering Council (JEDEC), making their holistic power management challenging to design.

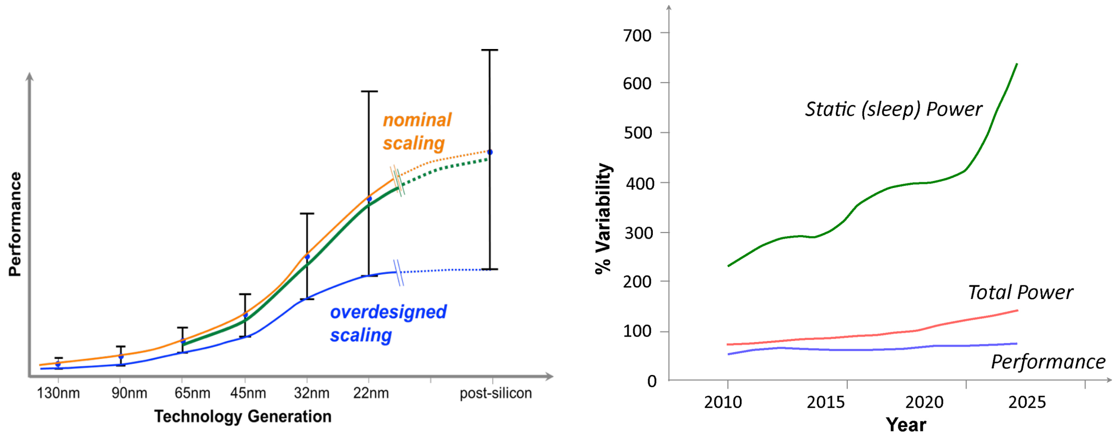
The phenomenon of *dark silicon* has already required large portions of a chip to be disabled at any moment in time to rein in power density [43], even with sophisticated power management techniques in both logic and memory. This is likely to continue, barring major breakthroughs in fundamental device and circuit technologies.

Although dark silicon is foreboding for future performance improvements, the architecture community has recognized that chip area remains plentiful with technology scaling, and that it can be used for *hardware accelerators* via heterogeneity [33,42,44,45], specialization [33,42,46,47], and dynamic reconfiguration [48,49]. For instance, compute-intensive tasks commonly done in software – such as image processing [50], compression [51], encryption [52], machine learning [53], and artificial neural networks [46,54] – are now often accelerated with hardware implementations. This strategy can yield dramatic performance and energy efficiency improvements [42]. Cloud computing companies such as Microsoft [48,55–57], Google [58], and Amazon [59] are all pursuing accelerator-based datacenter architectures.

Memory systems will likely become even more of a performance and energy bottleneck in the era of dark silicon. Hardware accelerators will likely have significantly higher memory bandwidth demands than general-purpose processors. Moreover, latency-sensitive applications that previously were bottlenecked by software would depend less on CPU performance and more on the accelerator hardware and the speed of its memory access.

I believe there are three major directions for research and development in future memory systems that promise greater energy efficiency, lower costs, and manageable power density.

- A heterogeneous mix of conventional volatile and emerging non-volatile memory (NVM) devices [60–65] is critical to save power and improve energy proportionality, especially as memory capacities continue to grow and consume larger pieces of the system-level area and power budgets.
- The design of 3D-integrated VLSI systems [66–69] and efficient networks-on-chip [70,71] promise tighter coupling of logic and memory through a significant reduction in wire length (latency improvement) and wider interfaces (bandwidth improvement). They are also an enabler of heterogeneous integration that is likely key to adoption of NVMs.



(a) Performance scaling penalty from overdesign (b) Worsening variation from technology scaling

Figure 1.3: Illustrations based on ITRS data [72] from the NSF Variability Expedition that depict worsening variability as a function of technology scaling.

- Better-than-worst-case operation of memories will be critical to efficient, high-performing, and reliable operation in the nanoscale era, which is plagued by *the problem of hardware variability*.

This last direction is the focus of research in this dissertation: *Opportunistic Memory Systems in Presence of Hardware Variability*.

1.2 Hardware Variability is a Key Issue for Memory Systems

One of the biggest challenges that has arisen in the nanometer-scale manufacturing nodes is hardware variability [7,73–75]. The ITRS predicted dramatic effects from variability in nanoscale technology nodes. Fig. 1.3a shows that in the presence of manufacturing and environmental operating challenges, margining for worst-case outcomes leads to pessimistic performance (blue line) compared to nominal performance scaling (yellow line). The class of underdesigned and opportunistic (UnO) design techniques [75] can help mitigate this problem, bringing performance scaling closer to nominal (green line) or even above it in the best case. Fig. 1.3b depicts how power variation especially worsens as a function of technology scaling. This is primarily because static (leakage-dominated) power has an exponential dependence on several circuit design parameters subject to variation. Because memories consume significant static power and are a major source of reliability failures, addressing variability in them is especially important.

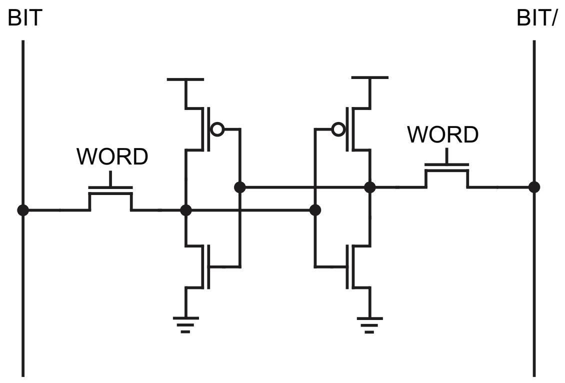
Variability manifests in two primary forms: manufacturing-based, and environment-based. The former can be categorized into functional defects [76] and within-die, die-to-die, and wafer-to-wafer parametric differences [73, 77]. For instance, a surface impurity on the scale of tens of nanometers on a lithographic mask [78] can cause yield loss-inducing defects to affect all printed wafers if steps are not taken to mitigate impurities. In another example, the delay of circuit paths is dependent on the threshold voltage and channel dimensions of the transistors comprising each gate [21, 22]. These parameters are directly affected by the degree of process control. Effects such as lithographic distortions [79], random dopant fluctuation [80, 81], and line edge roughness (LER) [82–84] can impact circuit delays, in turn affecting performance, reliability, and energy efficiency [85]. These challenges are unlikely to ease in the future; it is perhaps inevitable that manufacturing consistency will worsen as devices and circuit features are shrunk closer to atomic scales.

Environmental-based variability primarily consists of fluctuations in operating conditions (i.e., voltage/temperature) and long-term aging effects caused by circuit usage. For instance, supply voltage noise [86] and high temperature operation [87, 88] can affect functional correctness [89], performance [90], and power of circuits [91], particularly memories. They can also influence the circuit aging process [92] through mechanisms such as bias temperature instability (BTI) [93, 94], time-dependent dielectric breakdown (TDDB) [95], hot carrier injection (HCI) [96], and electromigration (EM) in wires [97]. Meanwhile, nanoscale process nodes increase the susceptibility of manufactured ICs to these environmental factors, compounding the variability problem.

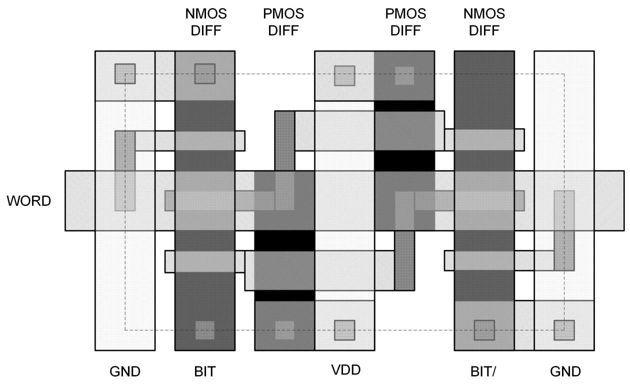
Variability strongly affects memory systems through three fundamental mechanisms that ultimately stem for the need for improved information density.

- Memory cells usually contain the smallest features in the entire chip. This makes them particularly susceptible to manufacturing defects and wearout through circuit aging processes.

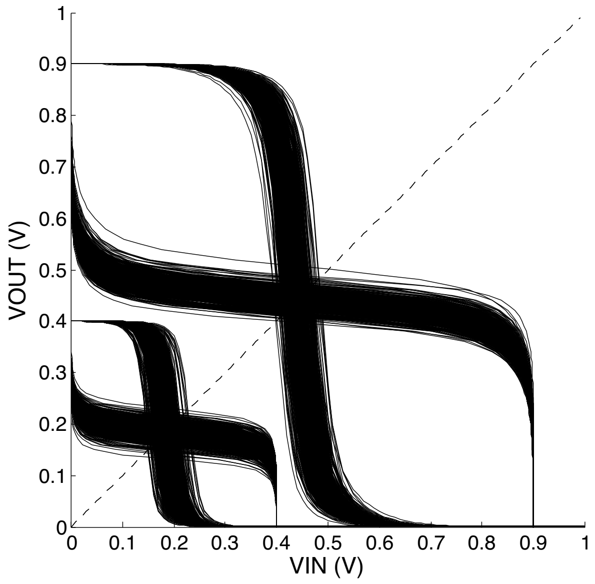
- Memory is often the largest consumer of chip real estate. This increases the likelihood that defects will affect memory, rather than logic, while process variations will have a greater



(a) Cell schematic



(b) Thin cell layout



(c) Impact of variability on cell static noise margins

Figure 1.4: Supply voltage scaling on SRAM cells under the impact of random parameter variation causes stability to degrade. This is a result of smaller worst-case noise margins, which is the separation between the pair of "butterfly curves" in the voltage transfer characteristic for holding state. The sets of butterfly curves were generated with 1000 Monte Carlo simulations on a 6T SRAM cell in a 45nm technology.

effect with respect to individual memory cells.

- By their very nature, memory cells must have multiple stable states such that information can be reliably read and over-written. This increases their susceptibility to changes in environmental operating conditions.

These fundamental mechanisms tend to apply regardless of the underlying memory technology,

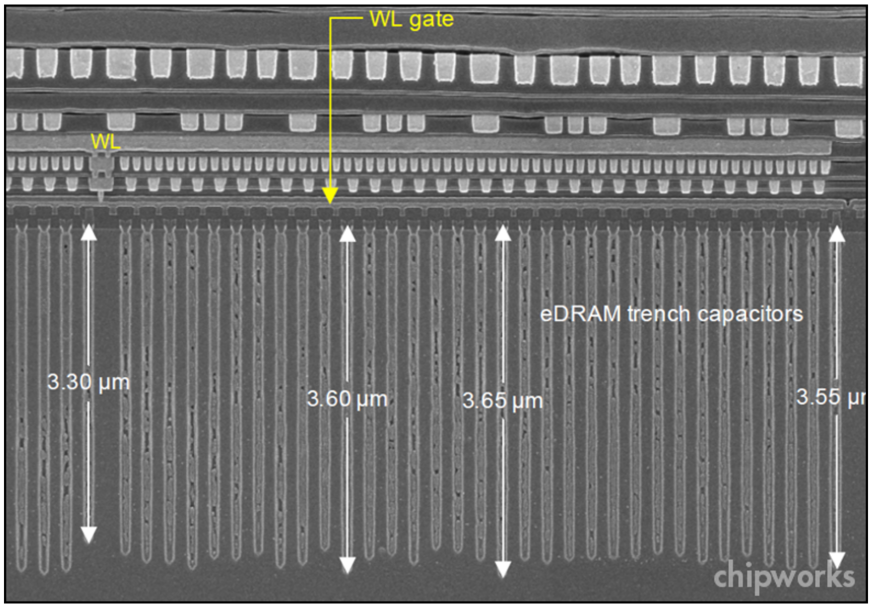


Figure 1.5: Cross-sectional image from ChipWorks and the Semiconductor Manufacturing and Design Community [98] of IBM's 32nm embedded DRAM in the POWER7+ processor. Process variations have a visible impact on the depth of individual trench capacitors.

including stalwarts like SRAM, DRAM, and flash, as well as emerging non-volatile memories (NVMs) like phase-change memory (PCM), spin-transfer torque RAM (STT-RAM), and resistive RAM (ReRAM, or memristor). For example, SRAM stability is highly sensitive to the chip supply voltage, while the magnetic tunnel junction (MTJ) in a spin-transfer-torque RAM (STT-RAM) cell is extremely sensitive to temperature. Fig. 1.4 shows the simulated effect of process variation on SRAM memory cells, where noise margins are degraded at low voltage, reducing their reliability at storing information. In another example, one can directly observe the effects of process variation on the depths of individual embedded DRAM trench capacitors in an IBM processor, as shown by Fig. 1.5. Such variations affect the retention time of individual memory cells. It is clear that memory reliability and energy efficiency pose a major obstacle to VLSI scaling, high-performance computing, warehouse-scale computing, and embedded systems.

Conventional approaches to coping with memory variability focus on improving manufacturing yield and parametric guard-banding for worst-case operation. Unfortunately, this approach is increasingly ineffective. Memory fault rates have been rising and guard-banding sacrifices performance and energy efficiency in most chips for ensured reliability in worst-case conditions. In

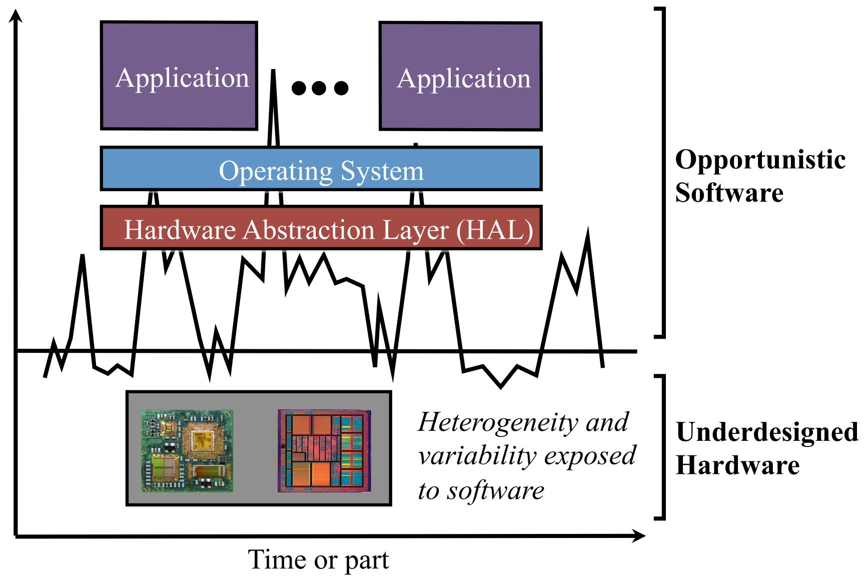


Figure 1.6: Vision for the Underdesigned and Opportunistic (UnO) computing concept [75], where software adapts to underlying hardware variation and by extension, heterogeneity as well. Opportunistic Memory Systems are a manifestation of adaptive architecture and system software.

recent years, researchers have studied many better-than-worst-case VLSI design concepts [99]. Much room still exists for innovation, particularly in memory devices, circuits, and systems. An *Opportunistic Memory System* that adapts to underlying hardware variability – and by extension, intentional heterogeneity – while taking application behavior into account could renew momentum in technology scaling. The Underdesigned and Opportunistic (UnO) concept is depicted in Fig. 1.6 and is based on ideas from [75].

1.3 Opportunistic Memory Systems in Presence of Hardware Variability

The research in this dissertation opportunistically exploits (Part 1) and copes (Part 2) with the unique outcomes that manifest as a consequence of hardware variability in memory systems.

Opportunistic Memory Systems would "nurture" individual chips by harnessing and tolerating their natural hardware variations, so that fewer individual chips are discarded early in life, and surviving chips can perform better and consume less power without harming their reliability and longevity. This would help technology scaling to continue, enabling future uses for computing and making it more sustainable. The high-level vision for Opportunistic Memory Systems is depicted

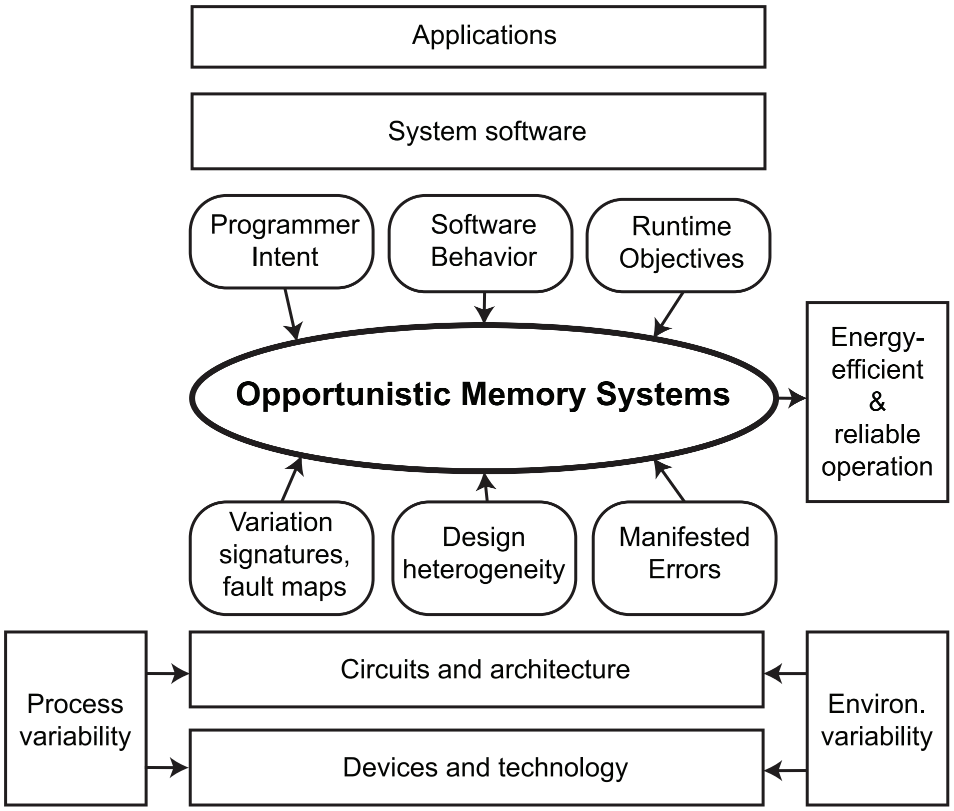


Figure 1.7: High-level concept of Opportunistic Memory Systems proposed in this dissertation. They exploit hardware variability and even heterogeneity to improve energy efficiency and reliability while accounting for software intent and semantics.

in Fig. 1.7. The specific solutions in this dissertation aim to improve both system-level energy efficiency and resiliency using complementary methods. Although these objectives are often conflicting, these projects seek to improve both simultaneously through complementary methods.

In Part 1 of this dissertation, three chapters are dedicated to novel techniques for *opportunistically exploiting memory variability* for lower power and higher performance.

- Chapter 2 presents a way to perform OS-level power variation-aware memory management (*ViPZonE*). A modified version of the Linux kernel's physical page allocator places frequently-accessed application data in DRAM modules that, due to variability, consume less power than others. Applications can then request low power at runtime using modified Linux system calls. ViPZonE can save significant energy when running benchmarks on a real workstation testbed; it has little impact on runtime.

- Chapter 3 outlines a resizable cache architecture that allows for rapid and low-overhead Dynamic Power/Capacity Scaling, known as *DPCS* (DPCS is analogous to processor dynamic voltage/frequency scaling – DVFS – but for on-chip caches.) DPCS leverages pre-characterized fault maps of on-chip SRAM caches to save energy at runtime via a combination of supply voltage scaling and power gating of faulty cache blocks. DPCS minimally degrades performance because it adapts dynamically as a program executes, and it has no impact on hardware reliability. DPCS saves more overall power than several competing approaches that achieve lower usable voltages because it has very low implementation overheads.

- Chapter 4 unveils *X-Mem*, a new extensible memory characterization tool. X-Mem supercedes the memory micro-benchmarking capabilities of all known prior tools. It is applied in a series of three experimental case studies that – among other research findings relevant to both cloud subscribers and providers – explores the efficacy of a latency variation-aware approach that might be used to improve application performance over a population of cloud servers. The tool can also be used by other researchers for a variety of purposes, including the future study of memory variability and possibly even security.

In Part 2 of this dissertation, three major contributions are made that *opportunistically cope with memory errors* for improved reliability at minimal cost.

- Chapter 5 studies *Performability*, i.e., the application-level performance impact of corrected errors (CEs) in memory. The project is motivated by recent observations in the field: memory errors occur quite frequently in datacenters and they degrade availability and performance consistency. Memory errors are injected in a controlled but realistic manner on a state-of-the-art cloud server. The application slowdown is measured and then used to derive analytical models that can be used to reason about the severity of memory errors on cloud systems.

- Chapter 6 presents a novel class of Software-Defined Error-Correcting Codes (*SDECC*). They promise to reduce the rate that systems fail from detected-but-uncorrectable errors (DUEs) in main memory. The approach is a blend of computer architecture and coding theory: it employs side information about the content of messages stored in memory to heuristically recover from the errors. In the vast majority of cases, SDECC is able to recover in a completely correct manner. Silent data corruption occurs in just a small fraction of the infrequent accidental miscorrections caused by failed heuristic recovery. SDECC achieves this without using a more powerful error-correcting code (ECC) that would add considerable storage, energy, latency, and bandwidth overheads.

- Finally, Chapter 7 proposes *ViFFTo*, a holistic approach to achieving virtualization-free fault-tolerance for embedded scratchpad memories at low-cost. It is comprised of two complementary techniques. First, FaultLink builds embedded application software that is custom-tailored to the variability exhibited by each individual chip on which it is deployed. Fault maps are pre-characterized for hard defects and voltage scaling-induced failures in on-chip memories. They are used to create binary images using a new fault-aware extension to the linker such that at run-time, the hard faults are never accessed. To deal with memory soft faults at run-time, ViFFTo then employs Software-Defined Error-Localizing Codes (SDELCs) that are based on novel Ultra-Lightweight Error-Localizing Codes (UL-ELCs). UL-ELCs are conceptually between basic single-error-detecting parity and a full single-error-correcting Hamming code. Loosely building upon the basic ideas proposed for SDECC in Chapter 6, a novel SDELC embedded software library allows embedded systems to recover a majority of random single-bit errors without the associated cost of a Hamming code.

The specific projects discussed throughout this dissertation could be generalized to other systems and memory technologies. When viewed as a suite of complementary techniques, the described research could benefit a broad domain of computing systems – from embedded edge devices in the IoT to warehouse-scale computers that drive the cloud and high-performance computing – with significant energy efficiency and reliability improvements that are critically needed in the nanoscale era.

Part I

Opportunistically Exploiting Memory Variability

CHAPTER 2

ViPZonE: Saving Energy in DRAM Main Memory with Power Variation-Aware Memory Management

Hardware variability is predicted to increase dramatically over the coming years as a consequence of continued technology scaling. In this chapter, we apply the Underdesigned and Opportunistic Computing (UnO) paradigm by exposing system-level power variability to software to improve energy efficiency. We present ViPZonE, a memory management solution in conjunction with application annotations that opportunistically performs memory allocations to reduce DRAM energy. ViPZonE's components consist of a physical address space with DIMM-aware zones, a modified page allocation routine, and a new virtual memory system call for dynamic allocations from userspace. We implemented ViPZonE in the Linux kernel with GLIBC API support, running on a real x86-64 testbed with significant access power variation in its DDR3 DIMMs. On our testbed, ViPZonE can save up to 27.80% memory energy, with no more than 4.80% performance degradation across a set of PARSEC benchmarks tested with respect to the baseline Linux software. Furthermore, through a hypothetical "what-if" extension, we predict that in future non-volatile memory systems which consume almost no idle power, ViPZonE could yield even greater benefits.

Collaborators:

- Dr. Luis A. D. Bathen, then at UC Irvine
- Prof. Nikil Dutt, UC Irvine
- Prof. Alex Nicolau, UC Irvine
- Prof. Puneet Gupta, UCLA

Source code and data are available at:

- `https://github.com/nanocad-lab?&q=vipzone`
- `http://nanocad.ee.ucla.edu`

2.1 Introduction

Inter-die and intra-die process variations have become significant as a result of continued technology scaling into the deep submicron region [73, 77]. The International Technology Roadmap for Semiconductors (ITRS) predicts that over the next decade, both performance and power consumption variation will increase by up to 66%, and 100%, respectively [72]. Variations can stem from semiconductor manufacturing processes, ambient conditions, device aging, and in the case of multi-sourced systems, vendors [5].

System design typically assumes a rigid hardware/software interface contract, hiding physical variations from higher layers of abstraction [75]. This is often accomplished through guard-banding, a method that ensures reliable and consistent components over all operation and fabrication corners. However, there are many associated costs from over-design, such as chip area and complexity, power consumption, and performance. Despite considerable hardware variability, the rigid hardware/software contract results in software assuming strict adherence to the hardware specifications. The overheads of guard-banding are reduced, but not eliminated, through the practice of binning, where manufacturers market parts with considerable post-manufacturing variability as different products. For example, manufacturers have resorted to binning processors by operating frequencies to reduce the impact of inter-die variation [75]. However, even with guardbanding, binning, and dynamic voltage and frequency scaling (DVFS), variability is inherently present in any set of manufactured chips. Furthermore, with the emergence of multi-core technology, intra-die variation has also become an issue. To minimize the overheads of guardbanding, recent efforts have shown that exploiting the inherent variation in devices [100, 101] yields significant improvements in overall system performance.

This has led to the notion of the Underdesigned and Opportunistic (UnO) computing paradigm [75], depicted in Fig. 2.1. In UnO systems, design guardbands are reduced while some hardware variations are exposed to a flexible software stack. This allows the system to tune itself to suit the unique characteristics of its hardware that arise from process variations (part variability), aging effects, and environmental factors such as voltage and temperature fluctuations (time variability).

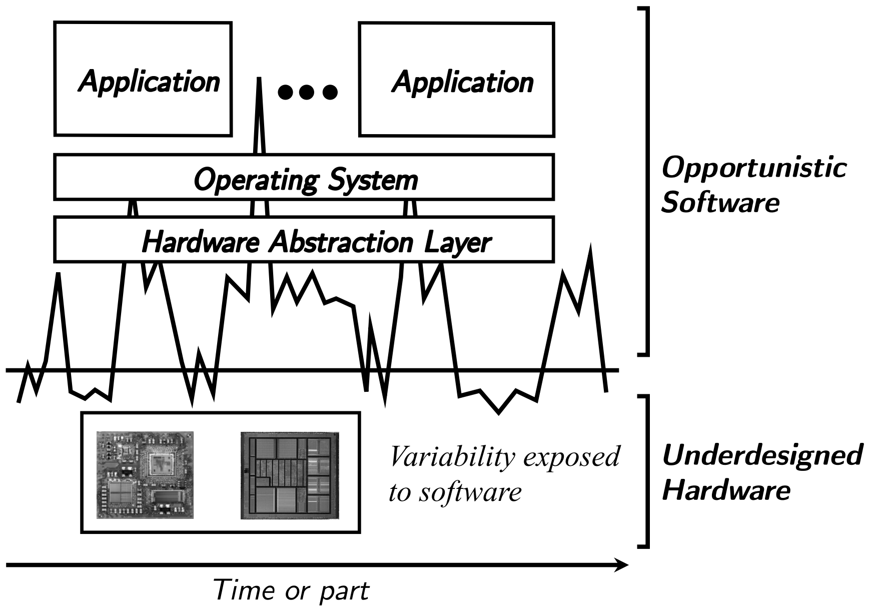


Figure 2.1: The original Underdesigned and Opportunistic (UnO) computing concept [75].

2.1.1 Related Work

There is an abundance of literature highlighting the extent of semiconductor process variability and methods for coping with it; we cite some of the more relevant work here. Recently, there have been several works which noted significant power variability in off-the-shelf parts. Hanson et al. [102] found up to 10% power variation across identical Intel Pentium M processors, while Wanner et al. [103] measured over 5x sleep power variation across various Cortex M3 processors. Hanson et al. [102] also observed up to 2X active power variation across various DRAMs, while Gottscho et al. [2] found up to 20% power variation in a set of 19 commodity 1 GB DDR3 DRAM modules (DIMMs).

Most efforts dealing with variation have focused on mitigating and exploiting it in processors [100, 101, 104, 105] or in on-chip memory [106–110]. Fewer papers have looked at variability in off-chip, DRAM-based memory subsystems. As off-chip DRAM memory may consume as much power as the processor in a server-class system [41], and is likely to increase for future many-core platforms (e.g., Tilera's TILEPro64 and Intel's Single Chip Cloud Computer (SCC)), memory power variations could have a significant impact on the overall system power.

Most works on main memory power management have focused on minimizing accesses to main memory through smart caching schemes and compiler/OS optimizations [111–115], yet none of these methods have taken memory variability into account. Bathen et al. [116] recently proposed the introduction of a hardware engine to virtualize on-chip and off-chip memory space to exploit the variation in the memory subsystem. These designs require changes to existing memory hardware, incurring additional design cost. As a result, designers should consider software implementations of power and variation-aware schemes whenever possible. Moreover, a variability-aware software layer should be flexible enough to deal with the predicted increase in power variation for current and emerging memory technologies.

2.1.2 Our Contributions

This chapter presents ViPZonE, an OS-based, pure software memory (DRAM) power variability-aware memory management solution that was first introduced in [4]; we have extended it with a full software implementation that is evaluated on a real hardware testbed. ViPZonE adapts to the power variability inherent in a given set of commodity DRAM memory modules by harnessing disjunct regions of physical address space with different power consumption. Our approach exploits variability in DDR3 memory at the DIMM modular level, but our approach could be adapted to work at finer granularities of memory, if variability data and hardware support are available. Our experimental results across various configurations running PARSEC [117] workloads show an average of 27.80% memory energy savings at the cost of no more than a modest 4.80% increase in execution time over an unmodified Linux virtual memory allocator.

The key contributions of this work are as follows:

- A detailed description and complete implementation of the ViPZonE scheme originally proposed in [4], including modifications to the Linux kernel and standard C library (GLIBC). These changes allow programmers control of power variability-aware dynamic memory allocation.
- An analysis of DDR3 DRAM channel and rank interleaving advantages and disadvantages using our instrumented x86-64 testbed, and the implications for variability-aware memory systems.

- An evaluation of power, performance, and energy of the ViPZonE implementation using a set of PARSEC benchmarks on our testbed.

- A hypothetical evaluation of the potential benefits of ViPZonE when applied to systems with negligible idle memory power (e.g., emerging non-volatile memory technologies).

ViPZonE is the first OS-level, pure-software, and portable solution to allow programmers to exploit main memory power variation through memory zone partitioning.

2.1.3 Chapter Organization

This chapter is organized as follows. We start with background material discussing the DDR3 DRAM memory system architecture, memory interleaving, and the relevant basics of Linux memory management in Sec. 2.2. Next, motivating results on the actual power variation observed in off-the-shelf DDR3 DRAM memory is presented and analyzed in Sec. 2.3. This is followed by a detailed description of the ViPZonE software implementation in Sec. 2.4, including the target platform and assumptions, kernel back-end, and the GLIBC front-end. In Sec. 2.5, we describe the testbed hardware and configuration, include an analysis of the benefits and drawbacks of memory interleaving for our testbed, and compare the ViPZonE software stack with the vanilla[1] code with memory interleaving disabled. A brief "what-if" study on ViPZonE for emerging non-volatile memories (NVMs) is covered in Sec. 2.5.4. We conclude our work and discuss opportunities for future research in Sec. 2.6.

2.2 Background

In this section, we provide a brief background on typical memory system architecture, memory interleaving, and vanilla Linux kernel memory management to aid readers in understanding our contributions.

[1]In this work, "vanilla" refers to the baseline unmodified software implementation.

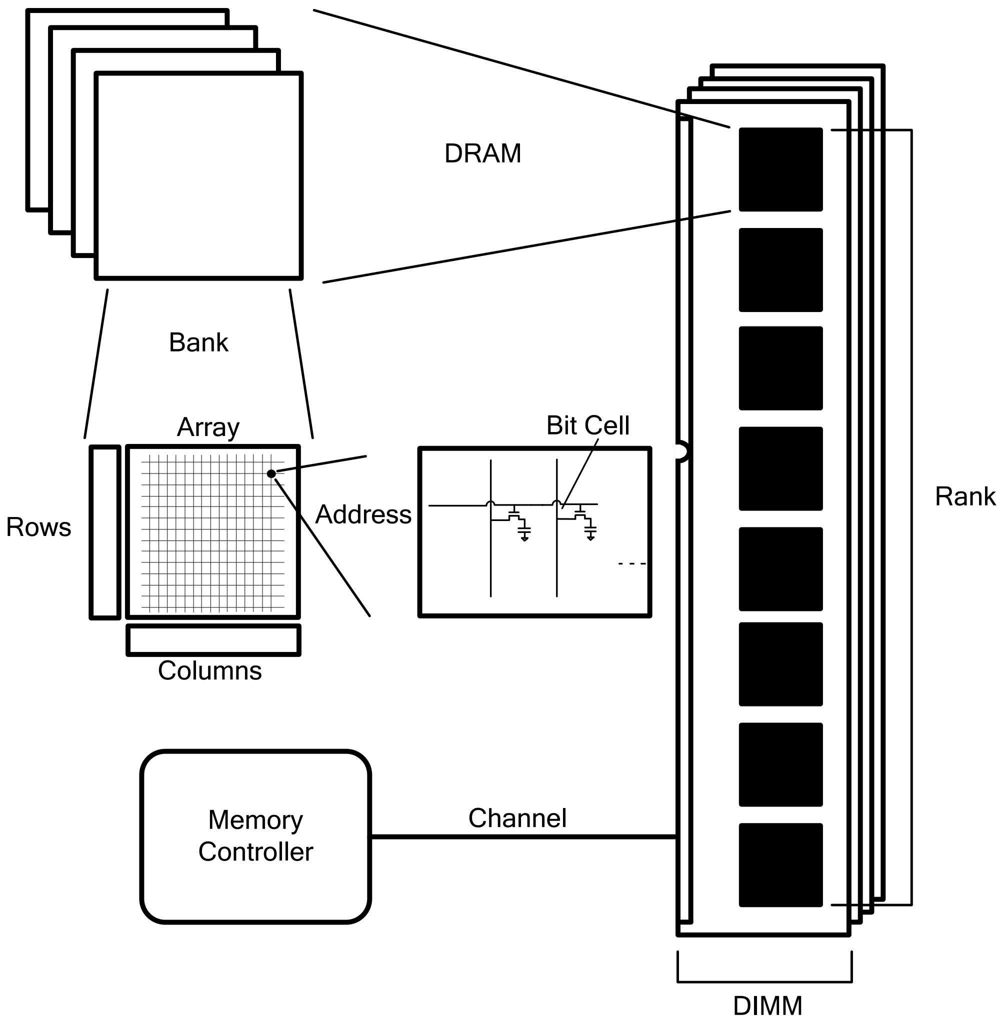


Figure 2.2: Components in a typical DDR3 DRAM memory system.

2.2.1 Memory System Architecture

To avoid confusion, we briefly define relevant terms in the memory system. In this chapter, we use DDR3 DRAM memory technology.

In a typical server, desktop, or notebook system, the memory controller accesses DRAM-based main memory through one or more memory *channels*. Each channel may have one or more *DIMMs*, which is a user-serviceable memory module. Each DIMM may have one or two *ranks* which are typically on opposing sides of the module. Each rank is independently accessible by the memory controller, and is composed of several DRAM devices, typically eight for non-ECC modules. Inside each DRAM are multiple *banks*, where each bank has an independent memory *array* composed of *rows* and *columns*. A memory location is a single combination of DIMM, rank, bank, row, and column in the main memory system, where an access is issued in parallel to all DRAMs, in lockstep, in the selected rank. This organization is depicted in Fig. 2.2.

For example, when reading a DIMM in the DDR3 standard, a burst of data is sent over a 64-bit wide bus for 4 cycles (with transfers occurring on both edges of the clock). During each half-cycle, one byte is obtained from each of eight DRAMs operating in lockstep, thus composing 64 bits that can be sent over the channel. Note that this is not "interleaving" in the sense that we use throughout the chapter, meaning that it is not configurable or software-influenced in any way; it is hard-wired according to the DDR3 standard.

2.2.2 Main Memory Interleaving

Since our scheme as implemented on our testbed requires channel and rank interleaving to be disabled, we include some background material on the benefits and drawbacks of interleaving here.

It is common practice for system designers to employ interleaved access to parallel memories to improve memory throughput, particularly for parallel architectures, e.g. vector or SIMD machines [118]. This is done by mapping adjacent chunks (the size of which is referred to as the *stride*) of addresses to different physical memory devices. Thus, when a program accesses several memory loctions with a high degree of spatial (in the linear address space) and temporal locality, the operations are overlapped via parallel access, yielding a speedup.

Many works have explored interleaving performance, generally in the context of vector and array computers, but also with MIMD machines as the number of processors and memory modules scale [119, 120]. While widely used today, interleaving does not necessarily yield improved performance. For example, the technique makes no improvement in access latency, and there is little performance gain when peak memory bandwidth requirements or memory utilization are low (e.g., high arithmetic intensity workloads as defined by the roofline model of computer performance [16]).

Furthermore, interleaving may yield negligible speedup when access patterns do not exhibit high spatial locality (e.g., random or irregular access), and is also capable of *worse* performance when several concurrent accesses have module conflicts as a result of the address stride, number of modules, and interleaving stride [118, 121]. Researchers have come up with techniques to mitigate or avoid this issue, usually through introducing irregular address mappings. For example, the Burroughs Scientific Processor used a 17-module parallel memory and argued that prime numbers of memory modules allowed several common access patterns to perform well [122]. Other approaches suggested skewed or permutation-based interleaving layouts [123], and clever data array organization for application-specific software routines [124].

In our testbed, interleaving prevents the exploitation of any power or performance variability present in the memory system. When striping accesses across different devices, the system runs all the memories at the speed of the slowest device, thus potentially sacrificing performance of faster modules, and preventing opportunistic use of varied power consumption. Interleaving on our testbed is also inflexible: it is statically enabled/disabled (cannot be changed during runtime), and it could also incur power penalties from the prevention of deeper sleep modes on a per-DIMM basis.

In this chapter, as interleaving and ViPZonE are mutually exclusive on our testbed, we provide an evaluation of power, performance, and energy for different interleaving modes in Sec. 2.5.2. We will discuss possible solutions to allow interleaving alongside variability-aware memory management in Sec. 2.6.

2.2.3 Vanilla Linux Kernel Memory Management

In order to understand our ViPZonE kernel modifications, we now discuss how the vanilla Linux kernel handles dynamic memory allocations from userspace, in a bottom-up manner. For interested readers, [125, 126] are excellent resources for understanding the Linux kernel, while [127] provides an exceptional amount of detail on the inner workings of the Linux memory management implementation.

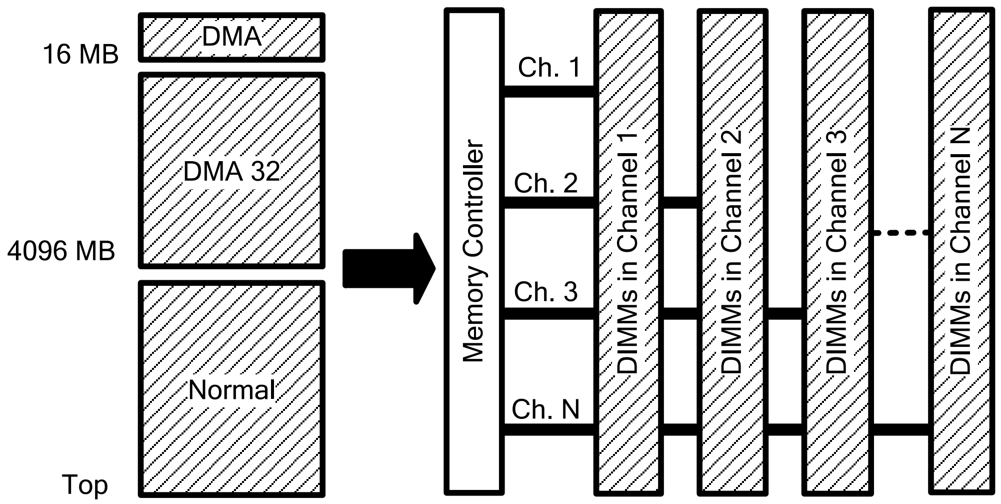


Figure 2.3: Vanilla Linux physical address space zoning for x86-64. Zone boundaries do not necessarily fall between DIMM boundaries.

2.2.3.1 Physical Memory Zones

The physical page allocator is at the core of the Linux kernel memory management subsystem. When presented with an allocation request for one or more pages with certain constraints, the system tries to find the most suitable allocation in the least amount of time. The kernel may pass through multiple stages during an allocation attempt, with greater performance penalties as it tries harder to find suitable memory.

The page allocator relies on several important constructs, including, but not limited to: page structures, memory zones, page freelists, and constraint bitmasks [125]. The kernel utilizes several zones to group regions of contiguous physical memory (see Fig. 2.3), required for legacy hardware support [125]. In direct memory access (DMA), devices talk directly with physical memory, bypassing the CPU. However, many legacy devices can only address the lowest 16 MB[2] and must be able to receive page allocations in this region. The kernel, if configured to support DMA, needs to reserve this space accordingly. There are also newer devices that are capable of addressing up to 4 GB of memory, and the kernel must be able to accomodate these DMA32 devices as well, albeit with more headroom.

[2]In this chapter, we adhere to conventional memory notation, as opposed to networking and storage notation for capacities. For example, we define 1 GB to be 2^{30} bytes of memory, not 10^9 bytes.

The kernel does this by representing these spaces with physically contiguous and adjacent DMA and DMA32 memory zones, each of which tracks pages in its space independently of other zones [125,128]. This allows for separate bookkeeping for each zone as well, such as low-memory watermarks, buddy system page groups, performance metrics, etc. Thus, if both are supported, the DMA zone occupies the first 16 MB of memory, while the DMA32 zone spans 16 MB to 4096 MB. This means that for 64-bit systems with less than 4 GB of memory, all of memory will be in DMA or DMA32-capable zones.

The rest of the memory space not claimed by DMA or DMA32 is left to the "Normal" zone[3]. On x86-64, this will contain all memory above DMA and DMA32. Since the kernel cannot split allocations across multiple zones [125], each allocation must come from a single zone. Thus, each zone maintains its own page freelists, least-recently-used lists, and other metrics for its space.

2.2.3.2 Physical Page Allocation

The kernel tries to fulfill page allocation requests in the most suitable zone first, but it can fall back to other zones if required [125,128]. For example, a user application will typically have its memory allocated in the normal zone. However, if memory there is low, it will try DMA32 next, and DMA only as a last resort. The kernel can also employ other techniques if required and permitted by the allocation constraints (if the request cannot allow I/O, filesystem use, or blocking, they may not apply) [125, 128]. However, the reverse is not true. If a device driver needs DMA space, it must come from the DMA zone or the allocation will fail. For this reason, the kernel does its best to reserve these restricted spaces for these situations [125].

2.2.3.3 Handling Dynamic Virtual Memory Allocations

In Linux systems, there are two primary system calls (syscalls) used for applications' dynamic memory allocations. For small needs, such as growing the stack, `sbrk()` is used, which extends the virtual stack space and grabs physical pages to back it as necessary [125]. `sbrk()` is also used by the GLIBC implementation of `malloc()`, as it is quite fast and minimizes fragmentation

[3]The "HighMem" zone present in x86 32-bit systems is not used in the x86-64 Linux implementation.

through the use of memory pooling [129]. For larger requests, `malloc()` usually resorts to the `mmap()` syscall, which is better suited for bigger, longer-lived memory needs, although is slower to allocate (`mmap()` also has other uses, such as shared memory, memory-mapped files, etc.). Both syscalls merely manage the virtual memory space for a process; they do not operate on the physical memory at all. The kernel generally only allocates physical backing pages lazily on use, rather than at allocation time.

2.3 Motivational Results

To develop effective methods of addressing variations (especially in the software layers), it is important to understand the extent of variability in different components of computing systems and their dependence on workload and environment. Though variability measurements through simple silicon test structures abound (e.g., [130], [131]), variability characterization of full components and systems have been scarce. Moreover, such measurements have been largely limited to processors (e.g., 14X variation in sleep power of embedded microprocessors [105] and 25% performance variation in an experimental 80-core Intel processor [132]). For a large class of applications, memory power is significant (e.g., 48% of total power in [133]) which has motivated several efforts to reduce DRAM (dynamic random access memory) power consumption (e.g., power-aware virtual memory systems [41, 112, 134]). These designs reduced power consumption of main memory, but they did not take into account hardware variability; instead, they assumed all DRAMs to be equally power efficient.

In our work, an Intel Atom-based testbed was constructed, running a modified version of Memtest86 [135] in order to control memory operations at a low level. We analyzed the write, read, and idle power consumption of several mainstream double data rate third generation (DDR3) dual inline memory modules (DIMMs), comprised of parts from several vendors and suppliers.

Table 2.1: DDR3 DIMM selection used for ViPZonE motivational results

Category	**Quantity**
Vendors	4 (V1-V4)
Suppliers	3 known (S1-S3, SU)
Capacities	1 GB (V1-V4), 2 GB (V1 only)
Models	Up to 3 per vendor, 8 total (7 were 1 GB)
Total DIMMs	22 (19 were 1 GB)

2.3.1 Test Methodology

Our DIMMs were comprised of several models from four vendors (see Table 2.1), manufactured in 2010 and 2011 (the particular process technologies are unknown). For five of the DIMMs, we could not identify the DRAM suppliers. Most models were 1 GB[4] DDR3 modules, rated for 1066 MT/s (except for the Vendor 4 models, rated for 1800 MT/s) with a specified supply voltage of 1.5 V. We also included three 2 GB specimens from Vendor 1 to see if capacity had any effect on power consumption. The DIMMs are referred to henceforth by abbreviations such as V1S1M1 for Vendor 1, Supplier 1, Model 1.

The test platform utilized an Intel Atom D525 CPU running at 1.80 GHz, running on a single core. Only one DIMM was installed at a time on the motherboard, and all other hardware was identical for all tests. No peripherals were attached to the system except for a keyboard, VGA monitor, and a USB flash drive containing the custom test routines. An Agilent 34411A digital multimeter sampled the voltage at 10 ksamples/s across a small 0.02 Ω resistor inserted on the V_{DD} line in between the DIMM and the motherboard slot, and this was used to derive the power consumption. Ambient temperature was regulated using a thermal chamber.

Because we required fine control over all memory I/Os, we developed custom modifications to Memtest86 v3.5b, which is typically used to diagnose memory faults [135]. The advantage of using Memtest86 as a foundation was the lack of any other processes or virtual memory, which granted us the flexibility to utilize memory at a low level.

We created a write function which wrote memory sequentially with a specified bit pattern, but

[4]To avoid confusion in terminology, we refer to the gigabyte (GB) in the binary sense, i.e., 1 GB is 2^{30} bytes, not 10^9 bytes.

Table 2.2: Testbed and measurement parameters for generating the motivating results for ViPZonE.

Parameter	**Value**
Testbed CPU	Intel Atom D525 @ 1.8 GHz
Number of CPU Cores Used	1
Cache Enabled	Yes
DIMM Capacities	1 GB, (2 GB)
DIMM Operating Clock Freq.	400 MHz
Effective DDR3 Rate	800 MT/s
DIMM Supply Voltage	1.5 V
Primary Ambient Temp.	30 °C
Secondary Ambient Temp.	-50, -30, -10, 10, 40, 50 °C
Primary Memory Test Software	Modified Memtest86 v3.5b [135]
Custom Test Routines	Seq. Write Pattern, Seq. Read, Idle
Digital Multimeter	Agilent 34411A
Sampling Frequency	10 ksamples/sec
Reading Accuracy	approx. 4.5 mW
Number of Samples Per Test	200000

never read it back. Similarly, a read function was created which only read memory sequentially without writing back. Each word location in memory could be initialized with an arbitrary pattern before the executing the read test. The bit fade test – which was originally designed to detect bit errors over a period of DRAM inactivity – was modified to serve as an idle power test, with minimal memory usage.[5] For all tests, the cache was enabled to allow for maximum memory bus utilization. With the cache disabled, we observed dramatically lower data throughput and were unable to distinguish power differences between operations. As our intent was primarily to measure power variability between different modules, we used sequential access patterns to avoid the effects of caches and row buffers.

Each test was sampled over a 20 second interval, during which several sequential passes over the entire memory were made. This allowed us to obtain the average power for each test over several iterations. Each reading had an estimated accuracy of 0.06 mV [136], which corresponds to approximately 4.5 mW assuming a constant supply voltage and resistor value. Table 2.2 summarizes the important test environment parameters. For further details on the testing methodology, refer to [3].

[5]Although there are different "idle" DRAM states, they are not directly controllable through software; we did not distinguish between them.

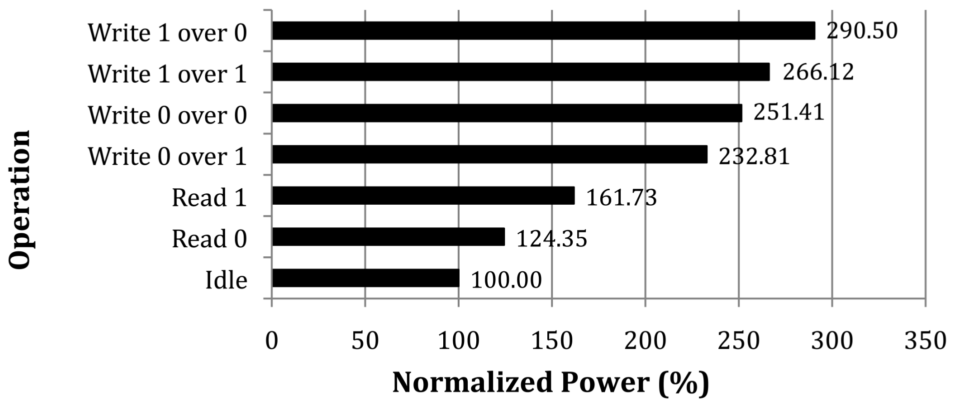


Figure 2.4: Measured data and operation dependence of DIMM power.

2.3.2 Test Results

2.3.2.1 Data Dependence of Power Consumption

Since DRAM power consumption is dependent on the type of operation as well as the data being read or written [137], we conducted experiments to find any such dependencies. Note that the background, pre-charge, and access power consumed in a DRAM should have no dependence on the data [138]. Note that this test is similar to one performed on SRAMs in [139].

Seven tests were performed on four DIMMs, each from a different vendor, at 30°C to explore the basic data I/O combinations. The mean power for each test was calculated from the results of the four DIMMs. Fig. 2.4 depicts the results for each test with respect to the idle case ("Write 0 over 0" refers to continually writing only 0s to all of memory, whereas "Write 1 over 0" indicates that a memory full of 0s was overwritten sequentially by all 1s, and so on). Note that for the idle case, there was negligible data dependence, so we initialized memory to contain approximately equal amounts of 0s and 1s.

Interestingly, the power consumed in the operations was consistently ordered as seen in Fig. 2.4, with significant differences as a function of the data being read or written. There was also a large gap in power consumption between the reading and writing for all data inputs.

We presume that the difference between the write 0 over 0 case and the read 0 test is purely due to the DRAM I/O and peripheral circuitry, as the data in the cell array is identical. This would also apply to the write 1 over 1 case and its corresponding read 1 case. In both the read 1 and write 1 over 1 cases, more power was consumed compared to the corresponding read 0 and write 0 over 0 cases. These deltas may be due to the restoration of cell values. Because a sense operation is destructive of cell data due to charge sharing [137], cells that originally contain 1s must be restored using additional supply current. In contrast, cells containing 0s need only be discharged.

Note that the write 0 over 1 test consumed *less* power than the write 0 over 0 test, whereas the write 1 over 0 case consumed *more* than the write 1 over 1 case. The write 1 over 0 case likely consumes the most power because the bit lines and cells must be fully charged from 0 to 1. In the write 0 over 1 case, it probably uses the least power because the bit lines and cells need only be discharged to 0. Further research and finer-grained measurement capabilities are required to fully explain these systematic dependencies. Nevertheless, these results indicate strong data and operation dependence in DRAM power consumption.

Because of the data dependencies in write and read operations, we decided to use memory addresses as the data for write and read in all subsequent tests, because over the entire address space, there are approximately equal quantities of 1s and 0s. Furthermore, memory addresses are common data in real applications. We verified that the average write and read power using addresses for data is approximately the same as the mean of the values for 1s and 0s as data.

2.3.2.2 Temperature Effects

To determine if temperature has any effect on memory power consumption, we tested four 1 GB modules, one from each vendor. Each DIMM was tested at ambient temperatures from -50 °C to 50 °C.[6] It is clear from Fig. 2.5 that temperature had a negligible effect on power consumption even across a large range. We speculate that this is partially due to the area and leakage-optimized DRAM architecture [21], but more substantially affected by modern refresh mechanisms. The use of rolling refreshes or conservative timings may consume significant dynamic power, overshadowing the temperature dependent components in the background power consumption. Since no DIMM exhibited more than 3.61% variation across a 100 °C range, all further tests were per-

[6]Testing above an ambient temperature of 50 °C was not practical as it caused testbed hardware failure.

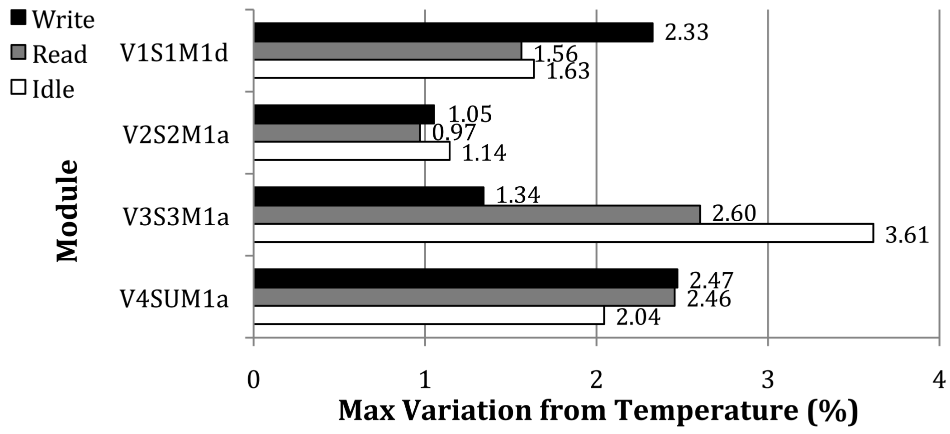


Figure 2.5: Relative temperature effects on write, read, and idle DIMM power, in a -50 °C to 50 °C range.

formed at an ambient temperature of 30 °C.

2.3.2.3 DIMM Power Variations

A plot of write, read, and idle power consumption for all 22 DIMMs at 30 °C is depicted in Fig. 2.6. The variability results are summarized in Fig. 2.7.

Variability within DIMMs of the same model (1 GB). Consider a particular model – V1S1M1 in Fig. 2.6 – of which we had the largest number (five) of specimens. While there was a maximum of 12.29% difference between the five DIMMs, there is a visible gap between the first group of three DIMMs and the second group of two (fourth and fifth in Fig. 2.6). This may be because the DIMMs come from two different production batches, resulting in lot-to-lot variability. The maximum deviations within the first group was only 1.34% for idle, and 1.47% within the second group. This suggests that the majority of the variation in V1S1M1 was between the two batches.

Variability between models of the same vendor/supplier (1 GB). Now consider all DIMMs from Vendor 1. We would expect that there would be more variation in Vendor 1 overall than in V1S1M1 only, and this was confirmed in the data. The maximum variation observed in Vendor 1 (1 GB) was 16.40% for the idle case. This variability may be composed of batch variability or performance differences between models.

Variability across vendors (1 GB). In order to isolate variability as a function of vendors and to

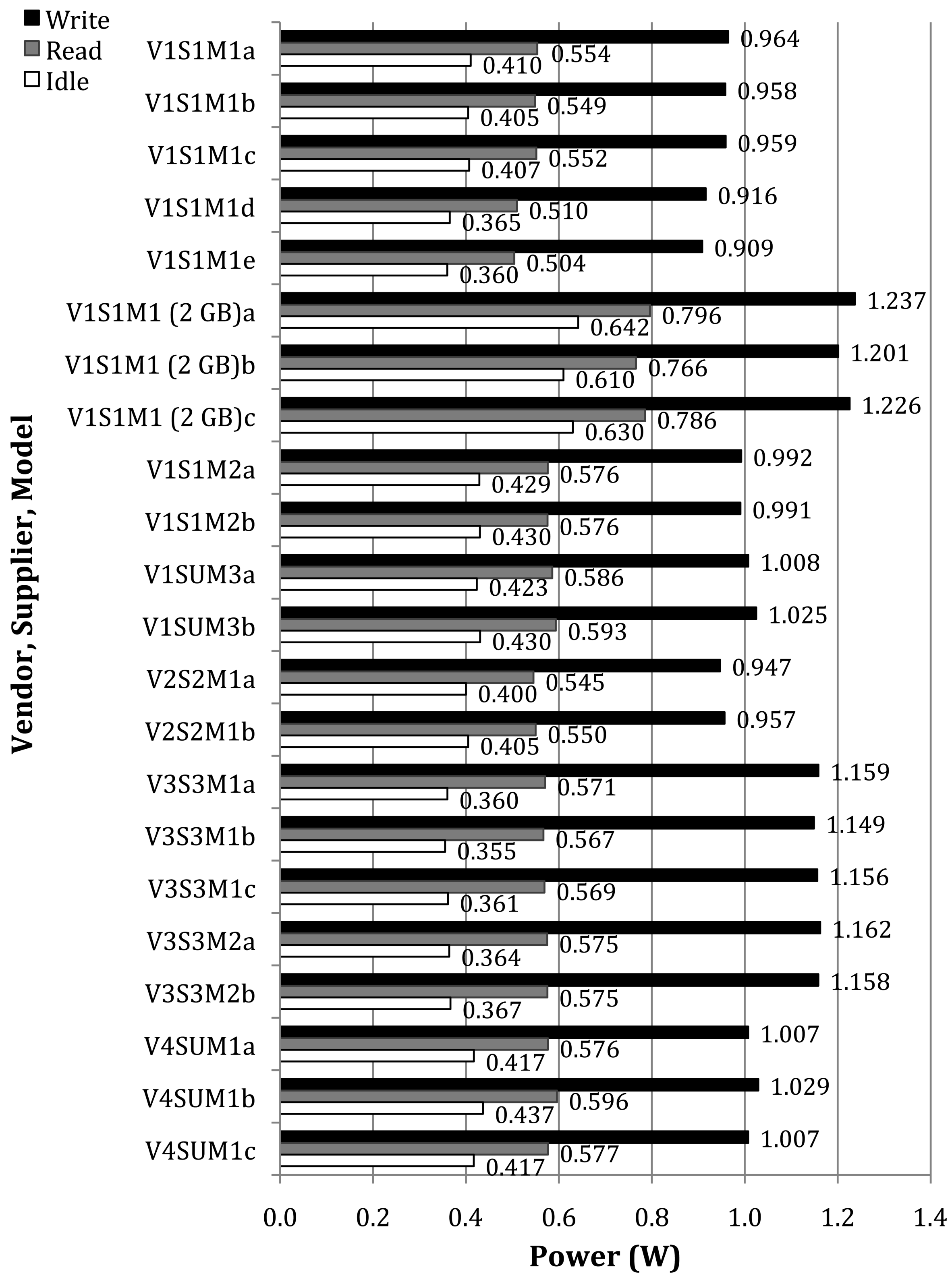


Figure 2.6: Write, read, and idle power for each tested DIMM at 30 °C.

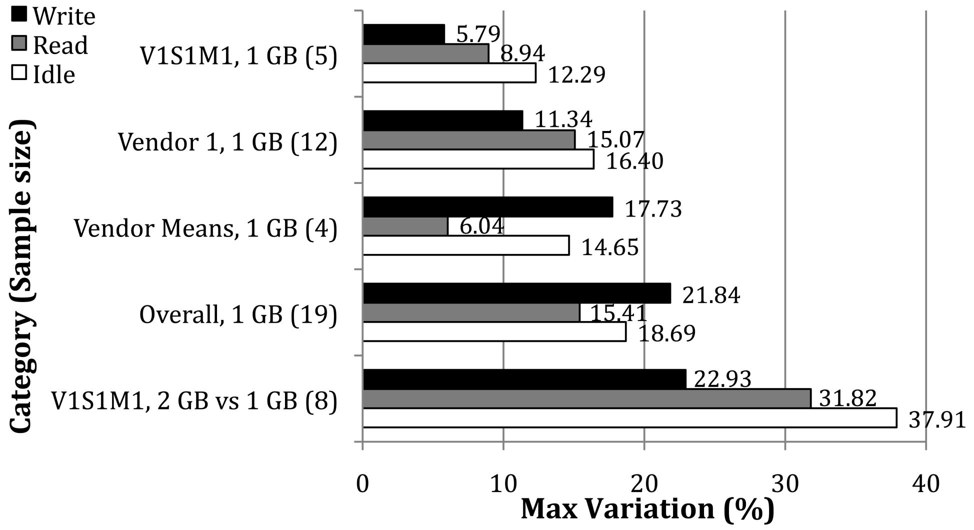


Figure 2.7: Max. variations in write, read, and idle power by DIMM Category at 30 °C.

mitigate any effects of different sample sizes, we computed the mean powers for each vendor (1 GB). Vendor 3 consumed the most write power at 1.157 W. The variations for write, read, and idle power were 17.73%, 6.04%, and 14.65% respectively.

Overall variability amongst 1 GB DIMMs. As one may have expected, the variations across all DIMMs were significantly higher than within a model and among vendors, with the maximum variation occurring for write power at approximately 21.84%.

Effects of capacity on power consumption. It is clear from Fig. 2.6 that the three 2 GB DIMMs of V1S1M1 consumed significantly more power than their 1 GB counterparts. This was expected, as there was bound to be higher idle power with twice as many DRAMs (in two ranks instead of one). Indeed, the maximum variation between the 2 GB and 1 GB versions was 37.91%, which occurred for idle power, whereas write power only differed by half as much. This is because background power is a smaller proportion of overall power when the DIMM is active.

2.3.3 Summary of Motivating Results

We analyzed the power consumption of several mainstream DDR3 DIMMs from different vendors, suppliers, and models, and found several important trends. Firstly, we did not find any significant temperature dependence of power consumption. Manufacturing process induced variation (i.e., variation for the same model) was up to 12.29%. Among models from the same vendor, idle power generally varied the most (up to 16.40% among Vendor 1), followed by read and write power. However, a different trend was evident across vendors, with write power varying the most (up to 17.73%), followed by idle and read power. This pattern was dominant overall amongst all tested 1 GB DIMMs, where we observed up to 21.84% variations in write power. Lastly, we found that a 2 GB model consumed significantly more power than its matching 1 GB version, primarily due to its increased idle power (up to 37.91%). Data-dependence of power consumption was also very pronounced, with about 30% spread within read and 25% spread within write operations. These findings serve as a motivation for variability-aware software optimizations to reduce memory power consumption. In an arbitrary set of DIMMs, there can be considerable variation in power use, and an adaptable system can use this to its advantage. Because we observed negligible temperature dependence, we will not include it in our future models of DIMM power.

2.3.4 Exploiting DRAM Power Variation

We now present the power variation measured among the eight DIMMs that were specifically used for the ViPZonE evaluation discussed later in the chapter. Fig. 2.8 depicts the results. These power deviations arise purely from vendor implementations and manufacturing process variation. Note that using DIMMs from different manufacturers in the same system may be common in a situation where there are many memories, and/or when DIMMs need to be replaced over time due to permanent faults (e.g., in datacenters). Moreover, the variability among DRAMs is expected to increase in the future [72], especially if variation-aware voltage scaling techniques are used, such as those proposed by [140].

In a variability-aware memory management scheme, the upper-bound on power savings is determined by the extent of power variation across the memories in the system. For example, if we assume interleaving to be disabled, the worst case for power consumption would be when the DIMM with the highest power contains all the data that is accessed, while the rest are idle. The best case would be where all of this data is on the lowest-power DIMM. In a case where data is spread evenly across all DIMMs, no power variation can be exploited and the result is similar to

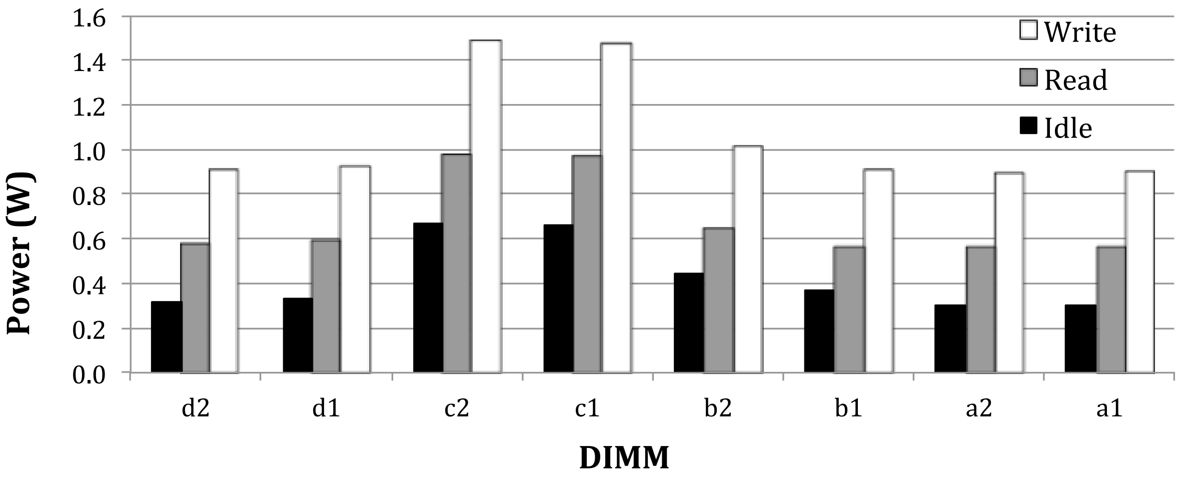


Figure 2.8: Measured power variations in eight identically specified off-the-shelf DDR3 DIMMs, using methods described earlier in Sec. 2.3.1. Letters denote different DIMM models, and numbers indicate instances of a model.

that if interleaving were used.

In a multi-programmed system where only a portion of the physical memory is occupied, then we can intelligently save energy. If most of the memory is occupied and accessed frequently, it is harder to exploit the power variations, but as long as there is some non-uniformity in the way different pages are accessed, it remains possible. We believe that the former scenario is a good case for our study, as any system where the physical memory is fully occupied will suffer from large performance bottlenecks due to disk swapping. When this happens, memory power and performance are no longer a first-order concern. Thus, we believe the latter scenario to be less interesting from the perspective of a system designer when considering power variability-aware memory management.

2.4 Implementation

ViPZonE is composed of several different components in the software stack, depicted in Fig. 2.9, which work together to achieve power savings in the presence of DIMM variability. We refer to the lower OS layer as the "back-end" and the application layer along with the upper OS layer as the "front-end". These are described in Sec. 2.4.2 and Sec. 2.4.3, respectively. ViPZonE uses source

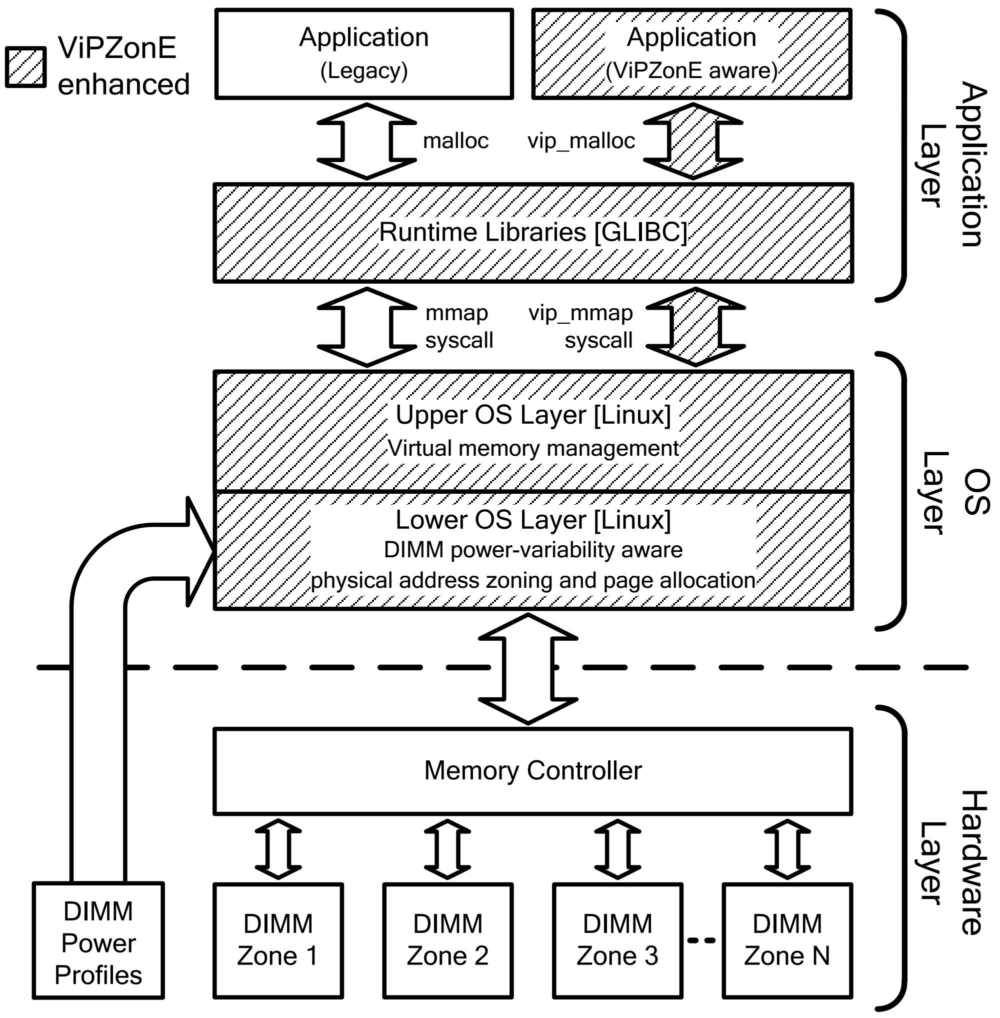


Figure 2.9: ViPZonE concept with a layered architecture.

code annotations at the application level, which work together with a modified GLIBC library to generate special memory allocation requests which indicate the expected use patterns (write/read dominance, and high/low utilization) to the OS[7]. Inside the back-end Linux memory management system, ViPZonE can make intelligent physical allocation decisions with this information to reduce DRAM power consumption. By choosing this approach, we are able to keep overheads in the OS to a minimum, as we place most of the burden of power-aware memory requests to the application programmer. With our approach, no special hardware support is required beyond software-visible power sensors or pre-determined power data that is accessible to the kernel.

[7]Our scheme does not currently support kernel memory allocations (e.g., `kmalloc()`). As the kernel space is generally a small proportion of overall system memory footprint, and invoked by all applications, we statically place the image and all dynamic kernel allocations in the low power zone.

There are alternative approaches to implementing power variation-aware memory management. One method could avoid requiring a modified GLIBC library and application-level source annotations by having the kernel manage all power variation-aware decisions. However, such an approach would place the burden of smarter physical page allocations on the OS, likely resulting in a significant performance and memory overhead. Furthermore, the kernel would be required to continuously monitor applications' memory accesses with hardware support from the memory controller. Nevertheless, ViPZonE's layered architecture means that implementing alternate memory management strategies could be done without significant changes to the existing framework. We leave the study of these alternative methods to future work.

2.4.1 Target Platform and Assumptions

We target generic x86-64 PC and server platforms that run Linux and have access to two or more DIMMs exhibiting some amount of power variability (ViPZonE cannot have a benefit with uniform memory power consumption). If device-level power variation is available, then this approach could be adapted to finer granularities, depending on the memory architecture. We make the following assumptions:

- ViPZonE's page allocator has prior knowledge of the approximate write and read power of each DIMM (for an identical workload). We could detect off-chip memory power variation, obtained by one of the following methods: (1) embedded power data in each DIMM, measured and set at fabrication time, or (2) through embedded or auxiliary power sensors sampled during the startup procedure.
- As DIMM-to-DIMM power variability is mostly dependent on process variations, and weakly dependent on temperature [2], there is little need for real-time monitoring of memory power use for each module. However, if power variation changes slowly over time (e.g., due to aging and wear-out occuring over time much greater than the uptime of the system after a single boot), we assume these changes can be detected through power sensors in each module.
- We can perform direct mapping of the address space[8] (e.g., select a single DIMM for each

[8]Note that address space layout randomization (ASLR) is not an issue, as ViPZonE deals with physical page

read/write request). This is achieved by disabling rank and channel interleaving on our testbed. We verified the direct mapping of addresses to specific DIMMs. Note that the particular division of DIMMs into ranks and channels is not a primary concern to ViPZonE; the only requirement is that only one DIMM is accessed per address.

- Programmers have a good understanding of the memory access patterns of their applications, obtained by some means, e.g., trace-driven simulations, allowing them to decide what dynamic memory allocations are "high utilization", or write-dominated, etc. Of course, in other scenarios, we do allow for annotation-independent policies.

2.4.2 Back-End: ViPZonE Implementation in the Linux Kernel

We discuss the implementation of ViPZonE in a bottom-up manner, in a similar fashion to the background material presented earlier. Our implementation is based on the Linux 3.2 [128] and GLIBC 2.15 [129] source for x86-64.

2.4.2.1 Enhancing Physical Memory Zoning to Exploit Power Variability

In order to support memory variability-awareness, the ViPZonE kernel must be able to distinguish between physical regions of different power consumption. With knowledge of these power profiles, it constructs separate physical address zones corresponding to each region of different power characteristics. The kernel can then serve allocation requests using the suggestions defined by the front-end of ViPZonE (see Sec. 2.4.3).

In the ViPZonE kernel for x86-64, we have explicitly removed the `Normal` and `DMA32` zones, while still allowing for DMA32 allocation support. Regular "DMA-able" space is retained. Zones are added for each physical DIMM in the system (`Zone_1`, `Zone_2`, etc.), with page ranges corresponding to the actual physical space on each DIMM. Allocations requesting DMA32-capable memory are translated to using certain DIMMs that use the equivalent memory space (i.e., addresses under 4 GB). Fig. 2.10 depicts our modified memory zoning scheme for the back-end. For example, in a system supporting DMA and DMA32, with 8 GB of memory located on four

placement only, while ASLR modifies the location of virtual pages.

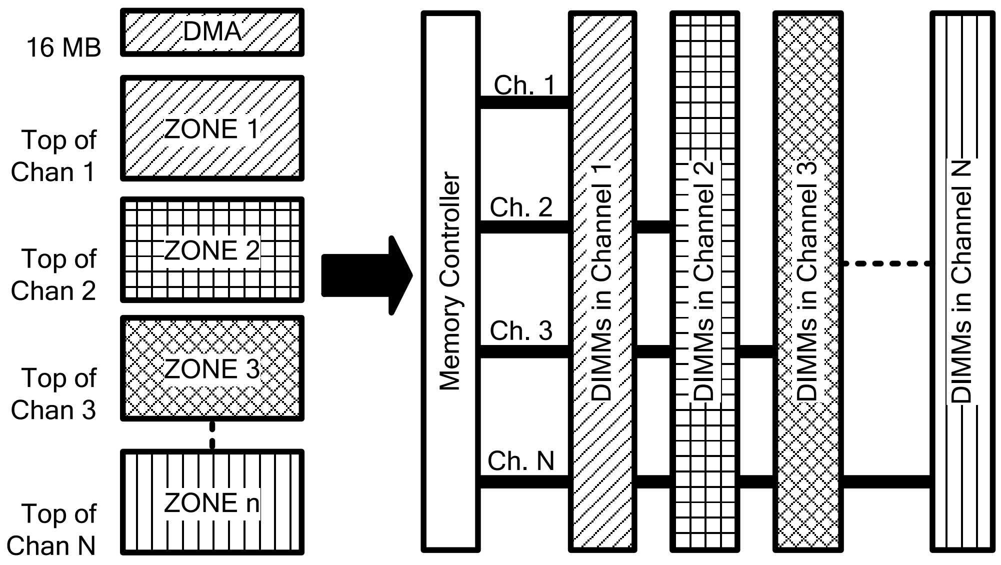


Figure 2.10: ViPZonE modifications to Linux physical address space zoning for x86-64. The kernel maintains separate zones which correspond directly to different physical DIMMs.

DIMMs (4x2 GB), the ViPZonE back-end would divide the memory space into zones as follows (we assume that each DIMM can have different power consumption):

- `Zone_DMA`: 0-16 MB, mapped to `DIMM_1`.
- `Zone_1`: 16-2048 MB, mapped to `DIMM_1`.
- `Zone_2`: 2048-4096 MB, mapped to `DIMM_2`.
- `Zone_3`: 4096-6144 MB, mapped to `DIMM_3`.
- `Zone_4`: 6144-8192 MB, mapped to `DIMM_4`.

2.4.2.2 Modifying the Physical Page Allocator in ViPZonE Linux x86-64

With zones set up for each DIMM, and knowledge of the relative power consumption of each DIMM, the kernel has the essential tools it needs to make power variability-aware page allocations, whereas the vanilla kernel makes no distinction between modular boundaries. There are many possible physical page allocation policies that could be used in the ViPZonE framework.

The ViPZonE kernel makes a distinction between relative write and read power for each DIMM zone. This is done for the hypothetical case where a module that has the lowest write power may not have the lowest read power, etc. Furthermore, it allows for future applicability to non-volatile memories, such as the MTJ-based family (MRAM, STT-RAM, MeRAM) with large differences in read and write power and energy [65].

The default policy that we implemented is *Low-Power-First*. The policy is currently configurable at kernel compile-time, but could be changed to work on-the-fly if policies need to be changed at runtime. In this policy, for high-utilization allocations, the kernel tries to get the lowest read or write power zone available. For low-utilization requests, the kernel still tries to fulfill it in a low read or write power zone, as long as some free space is reserved for high-utilization requests. Allocations requiring DMA32-compatible space are restricted to zones less than 4 GB, but otherwise follow the same utilization-priority rules. Finally, legacy DMA requests (less than 16 MB) always are granted in `Zone_DMA`. The *Low-Power-First* policy is depicted in Fig. 2.11.

For example, in a system with four DIMMs, each with 2 GB of space, the ViPZonE kernel would make allocation decisions as follows:

- *Request for DMA-capable space*: Grant in `Zone_DMA`.
- *Request for DMA32-capable space (superset of DMA)*: Restrict possibilities to `Zone_1` or `Zone_2`. If indicated utilization is `low`, choose the lower `write` (if indicated) or `read` (if indicated, or default) power zone, as long as at least `THRESHOLD` free space is available (generally, we choose `THRESHOLD` to be approximately 20% of DIMM capacity). If indicated utilization is `high`, always choose the lower power (`write/read`) zone if possible. If neither zone has enough free space, the kernel uses the vanilla page reclamation mechanisms or can default to `Zone_DMA` as a last resort.
- *Request for Normal space*: Grant in any zone, with the order determined in the same fashion as the above case for DMA32, without the 4 GB address restriction. `Zone_DMA` is only used as a last resort.

Alternatively, more sophisticated policies could use a variety of tricks. For example, a kernel which tracks page accesses might employ page migration to promote highly utilized pages to low-

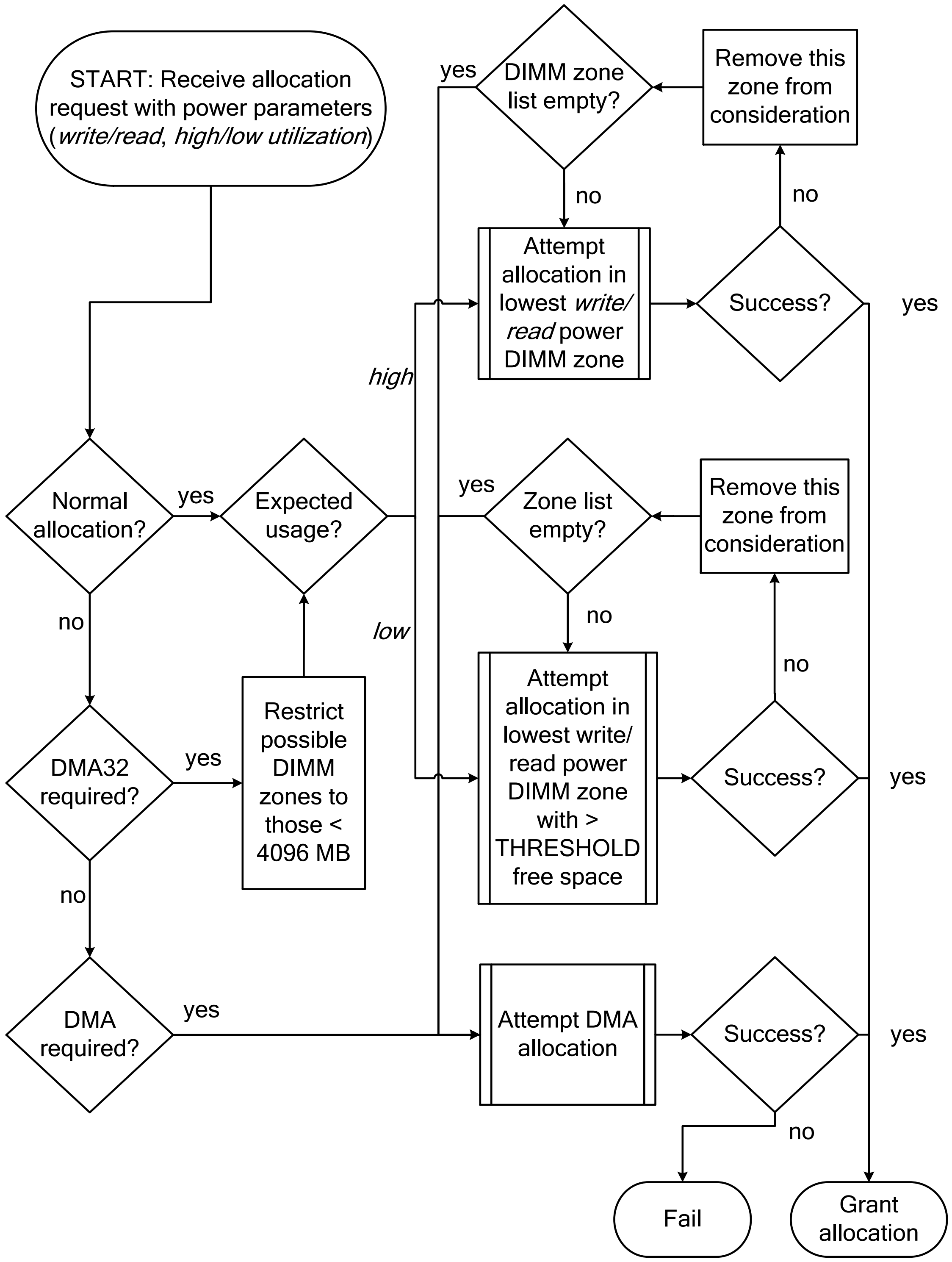

START: Receive allocation request with power parameters (*write/read*, *high/low utilization*)
Normal allocation?
yes
no
Expected usage?
high
low
DMA32 required?
yes
no
Restrict possible DIMM zones to those < 4096 MB
DMA required?
yes
DIMM zone list empty?
yes
no
Remove this zone from consideration
Attempt allocation in lowest *write/ read* power DIMM zone
Success?
no
yes
Zone list empty?
yes
no
Remove this zone from consideration
Attempt allocation in lowest write/ read power DIMM zone with > THRESHOLD free space
Success?
no
yes
Attempt DMA allocation
Success?
yes
no
Fail
Grant allocation

Figure 2.11: *Low-Power-First* ViPZonE physical memory allocation policy.

power space while demoting infrequently used pages to high power space. However, this would need to be carefully considered, as the performance and power costs of tracking and migrating pages might outweigh the benefits from exploiting power variation. We leave the study of these policies to future work.

2.4.3 Front-End: User API and System Call

The other major component of our ViPZonE implementation is the front-end, in the form of upper layer OS functionality in conjunction with annotations to application code. The front-end allows the programmer to provide hints regarding intended use for memory allocations, so that the kernel can prioritize low power zones for frequently written or read pages. The GNU standard C library (GLIBC) was used to implement our power variation enhanced allocation functions as part of the standard library (source code available at [141]). We briefly describe the methods and their use.

We added two new features to the applications' API, described in Table 2.3, allowing the programmer to indicate to the virtual memory manager the intended usage. We implemented a new GLIBC function, `vip_malloc()`, as a new call to enable backwards compatibility with all non-ViPZonE applications requiring the use of vanilla `malloc()`. `vip_malloc()` is essentially a wrapper for a new syscall, `vip_mmap()`, that serves as the hook into the ViPZonE kernel. While a pure wrapper for a syscall is not ideal due to performance and fragmentation reasons, we found it sufficient to evaluate our scheme. `vip_malloc()` can be improved further to implement advanced allocation algorithms, such as those in various C libraries' `malloc()` routines.

Because low power memory space is likely to be precious, memory should be released to the OS as soon as possible when an application no longer needs it. As a result, we preferred the use of the `mmap()` syscall over `sbrk()`, which has the heap grow contiguously. With `sbrk()`, it is often the case that memory is not really freed (i.e., usable by the OS). For this reason, a ViPZonE version of the `sbrk()` syscall was not implemented. This also keeps the `vip_malloc()` code as simple as possible for evaluation. We do not expect that it would have a major effect on power or performance.

`vip_malloc()` can be used as a drop-in replacement for `malloc()` in application code, given

Table 2.3: ViPZonE application programming interface (API)

Function	Parameter	Type	Description
`void* vip_malloc()`	`bytes`	`size_t`	Request size
	`vip_flags`	`size_t`	Bitmap flag used by ViPZonE back-end page allocator
`void* vip_mmap()` (syscall)	`addr`	`void *`	Address to be mapped, typically NULL (best effort)
	`len`	`size_t`	Size to be allocated, in bytes
	`prot`	`int`	Std. `mmap()` protection flags bitwise ORed with `vip_flags`
	`flags`	`int`	Std. `mmap()` flags
	`fd`	`int`	Std. `mmap()` file descriptor
	`pgoff`	`off_t`	Std. `mmap()` page offset

the ViPZonE GLIBC is available. If the target system is not running a ViPZonE-capable kernel, `vip_malloc()` defaults to calling the vanilla `malloc()`. Custom versions of `free()` and the `munmap()` syscall are not necessary to work with the variability-aware memory manager.

The Linux 3.2 `mmap()` code was used as a template for the development of `vip_mmap()`. Furthermore, the kernel includes ViPZonE helper functions that allow it to pass down the flags from the upper layers in the software stack down to the lower levels, from custom `do_vip_mmap_pgoff()`, `vip_mmap_region()` to the modified physical page allocator (`__alloc_pages_nodemask()`). For this purpose, we reserved two unused bits in `vip_mmap()`'s `prot` field to contain the `vip_flags` passed down from the user.

Table 2.4 shows the sample set of flags supported by `vip_malloc()`, passed down to the ViPZonE back-end kernel to allocate pages from the preferred zone according to the mechanism described earlier in Sec. 2.4.2. The flags (`_VIP_TYP_READ`, `_VIP_TYP_WRITE`) tell the allocator that the expected workload is heavily read or write intensive, respectively[9]. If no flags are specified, the defaults of `_VIP_TYP_READ` and `_VIP_UTIL_LOW` are used. We decided to support these flags (only two bits) rather than using a different metric (e.g., measured utilization), since keeping track of page utilization would require higher storage and logic overheads.

An application could use the ViPZonE API to reduce memory power consumption by intel-

[9]It is left to the developer to determine what constitutes read or write dominance depending on the semantics of the code. In our implementation, this did not matter, as DIMMs with high write power also had high read power, etc.

Table 2.4: Supported flags that may be provided by the application programmer to the ViPZonE API

Parameter	Flag	Meaning
Dominant Access Type	_VIP_TYP_WRITE	The memory space will have more writes than reads
	_VIP_TYP_READ	The memory space will have more reads than writes
Relative Utilization	_VIP_LOW_UTIL	Low utilization (low power space when plentiful)
	_VIP_HI_UTIL	High utilization (low power space)

ligently using the flags. For example, if a piece of code will use a dynamically-allocated array for heavy read utilization (e.g., an input for matrix multiplication), then it can request memory as follows:

```
retval = vip_malloc(arraySize*elementSize,
_VIP_TYP_READ | _VIP_HI_UTIL);
```

Alternatively, the application could use the syscall directly:

```
retval = vip_mmap(NULL, arraySize*elementSize,
PROT_READ | PROT_WRITE | _VIP_TYP_READ | _VIP_HI_UTIL,
MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

For each `vip_mmap` call, the kernel tries to either expand an existing VM area that will suit the set of flags, or create a new area. When the kernel handles a request for physical pages, it checks the associated VM area, if applicable, and can use the ViPZonE flags passed from user-space to make an informed allocation.

As shown by this example, the necessary programming changes to exploit our variability-aware memory allocation scheme are minimal, provided the application developer has some knowledge about the application's memory access behavior.

Table 2.5: Common parameters among ViPZonE testbed configurations

Parameter	Value
CPU	Intel Core i7-3820 (Sandy Bridge-E)
CPU supporting features	All enabled (default)
Motherboard	Gigabyte X79-UD5 (socket LGA2011)
Linux kernel version	3.2.1
BIOS version	F8
GLIBC version (baseline)	2.15
Storage	SanDisk SDSSDX120GG2 120 GB SSD (SATA 6 Gbps)
DAQ	NI USB-6215
No. DIMMs	2
DIMM power sample rate	1 kHz per DIMM
No. memory channels	2
Data logging	Second machine
DIMM capacity	2 GB each
Base core voltage	1.265 V
DDR3 data rate	1333 MT/s
DRAM voltage	1.5 V

2.5 Evaluation

In this section, we demonstrate that ViPZonE is capable of reducing total memory power consumption with negligible performance overheads, thus yielding energy savings for a given benchmark compared to vanilla Linux running on the same hardware. We start with an overview of our testbed and measurement hardware and configurations in Sec. 2.5.1. A comparison of the four combinations of memory interleaving modes is in Sec. 2.5.2 to quantify the advantages and disadvantages of disabling interleaving to allow variability-aware memory management. In Sec. 2.5.3 we compare the power, performance, and energy of ViPZonE software with respect to vanilla Linux as a baseline, using three alternate testbed configurations.

2.5.1 Testbed Setup

We constructed an x86-64 platform that would allow us fine control over hardware configuration for our purposes. Table 2.5 lists the important hardware components and configuration parameters used in all subsequent evaluations. The motherboard BIOS allowed us to enable and disable

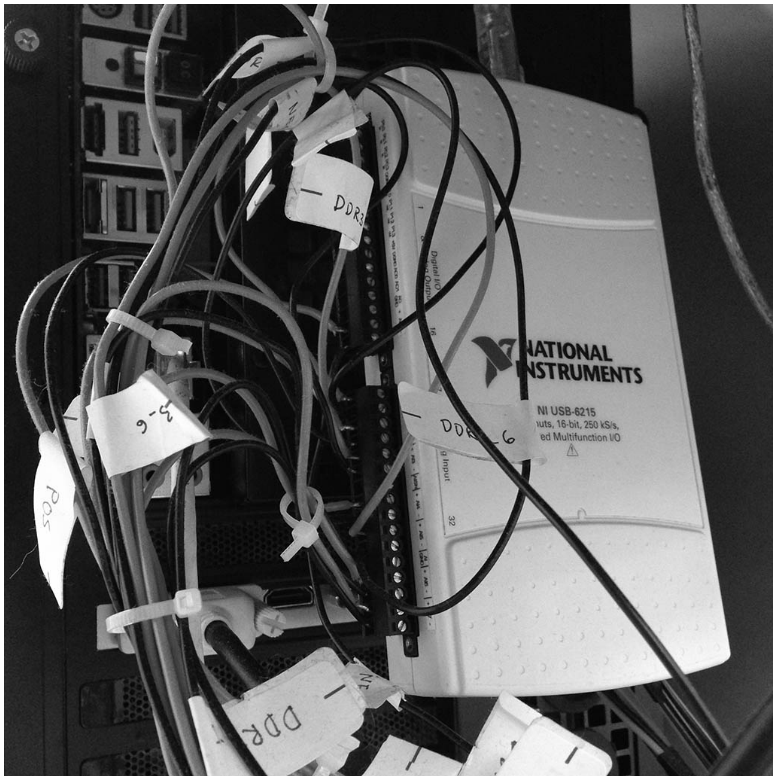


Figure 2.12: Testbed photo showing our DAQ mounted on the reverse of the chassis for DIMM power measurement.

Table 2.6: Two ViPZonE testbed configurations with different CPU performance

Parameter	**Value**	**Parameter**	**Value**
`Fast2` Config			
No. enabled cores	4	TurboBoost	Enabled
Nominal core clock	3.6 GHz	HyperThreading	Enabled
		No. of PARSEC threads	8
`Slow2` Config			
No. enabled cores	2	TurboBoost	Disabled
Base core clock	1.8 GHz	HyperThreading	Disabled
		No. of PARSEC threads	1

channel and rank interleaving independently, as well as adjust all voltages, clock speeds, memory latencies, etc. if necessary. Memory power was measured on a per-DIMM basis using an external data acquisition (DAQ) unit, as shown by the testbed photo in Fig. 2.12. Data was streamed to a laptop for post-processing.

Table 2.6 lists two different CPU and memory configurations that share the same parameters from Table 2.5. Unless otherwise specified in the tables, all minor BIOS parameters were left at default values. In the `Fast2` configuration, two DIMMs populated the motherboard with up to 50% active total power variation (DIMMs `b1` and `c1` as depicted in Fig. 2.8 and measured in the same way as described earlier in our motivating results from Sec. 2.3. In the `Slow2` configuration, the memory was set identically but the CPU was underclocked to 1.8 GHz, with only two cores enabled, while Intel TurboBoost and HyperThreading were disabled. None of these configurations specify anything about the channel and rank interleaving modes, which is evaluated in Sec. 2.5.2. Our testbed configurations were chosen to represent two flavors of systems, those which may be CPU-bound and those which may be memory-bound in performance.

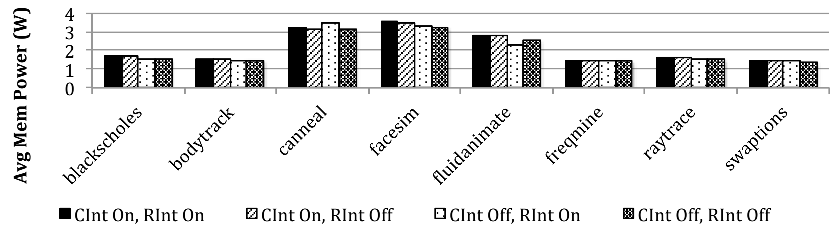
We used eight benchmarks from the PARSEC suite [117] that are representative of modern parallel computing: `blackscholes`, `bodytrack`, `canneal`, `facesim`, `fluidanimate`, `freqmine`, `raytrace`, and `swaptions`.

2.5.2 Interleaving Analysis

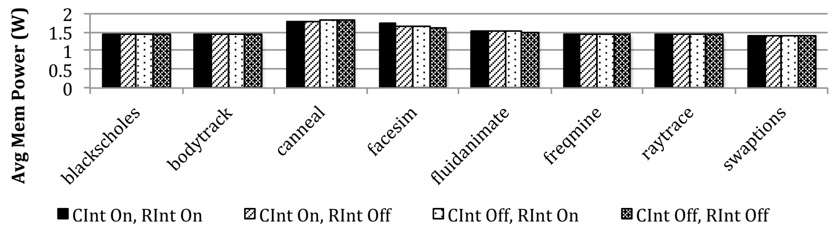
Since disabling interleaving is required for ViPZonE functionality and exploitation of memory variability in our testbed environment, we measured the average memory power, execution time, and total memory energy for different PARSEC benchmarks under both testbed configurations.

The available combinations of interleaving were:

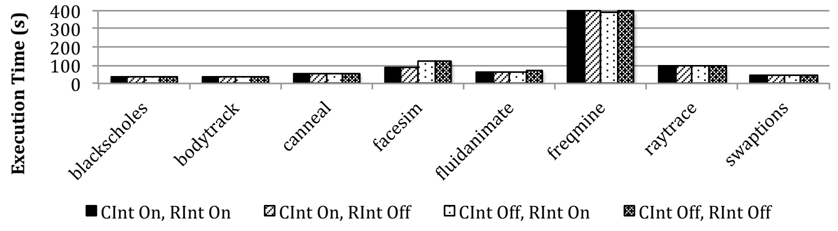
- `Cint On, Rint On`. Channel interleaving and rank interleaving are enabled.
- `Cint On, Rint Off`. Channel interleaving is enabled, while rank interleaving is disabled.
- `Cint Off, Rint On`. Channel interleaving is disabled, while rank interleaving is enabled.
- `Cint Off, Rint Off`. Channel interleaving and rank interleaving are disabled.



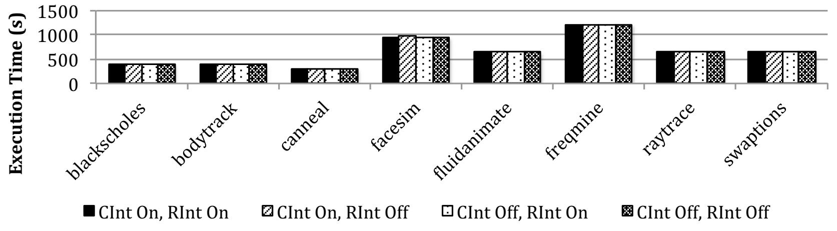
(a) `Fast2` Memory Power



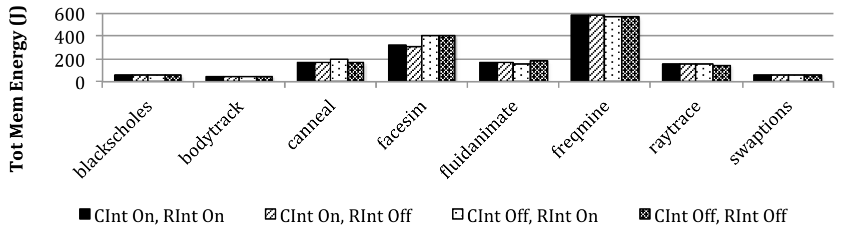
(b) `Slow2` Memory Power



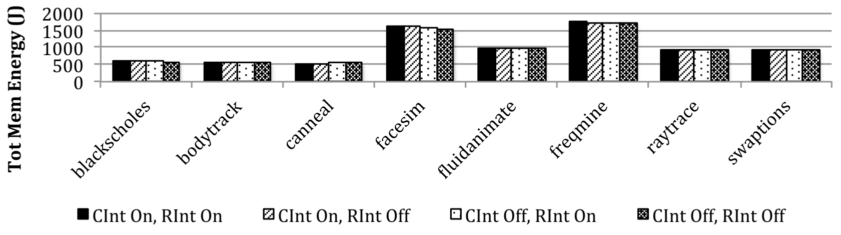
(c) `Fast2` Runtime



(d) `Slow2` Runtime



(e) `Fast2` Memory Energy

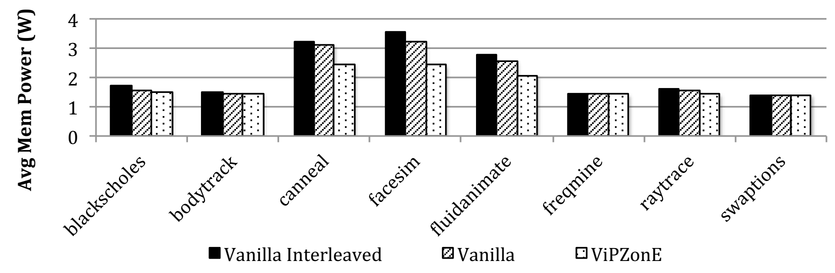


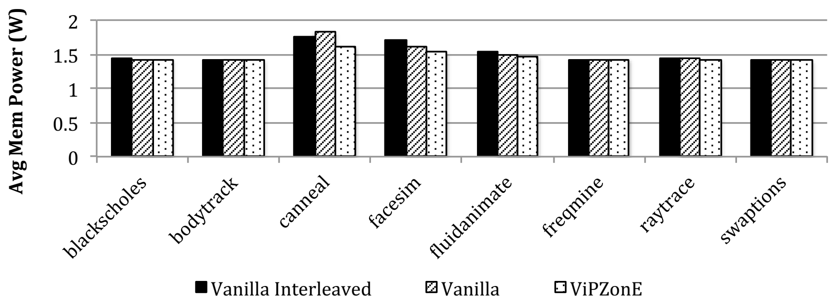
(f) `Slow2` Memory Energy

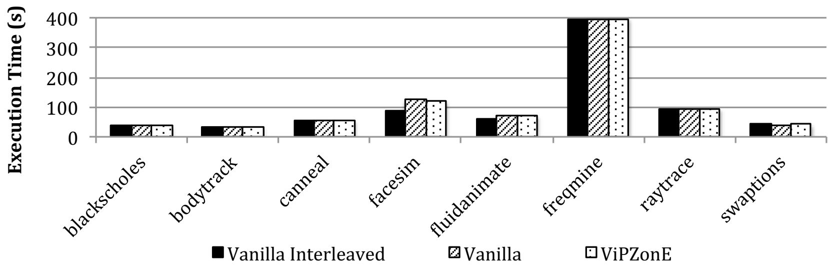
Figure 2.13: Evaluation of channel and rank interleaving for both testbed configurations on vanilla Linux.

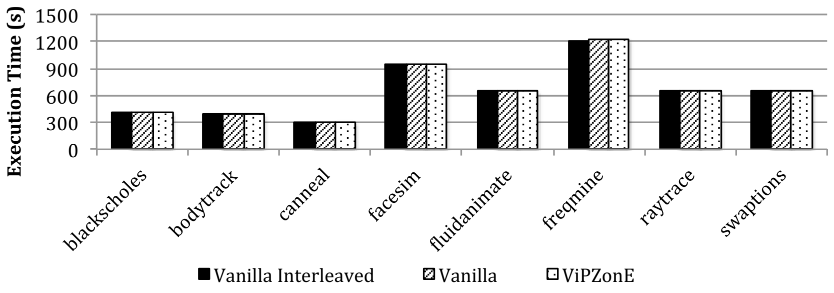
In the high-end CPU configuration, `Fast2` (results in Figs. 2.13a, 2.13c, 2.13e), the CPU was set for maximum performance, with PARSEC running with eight threads to stress the memory system. For benchmarks with high memory utilization, such as `canneal`, `facesim`, and `fluidanimate`, we found that turning off channel interleaving generally reduced power consumption (Fig. 2.13a), while total memory energy increased or decreased depending on the application (Fig. 2.13e) due to degradation in performance from lower main memory throughput (Fig. 2.13c). Rank interleaving had less impact on power or performance, which suggests that the main throughput bottleneck is the effective bus width rather than the devices. Conversely, for workloads with lower main memory utilization, there was negligible difference in power, performance, and energy. This confirms our intuition that the benefits of interleaving are in the improvement of peak memory throughput, which is only a bottleneck for certain workloads where the CPU is sufficiently fast and/or application memory utilization is high.

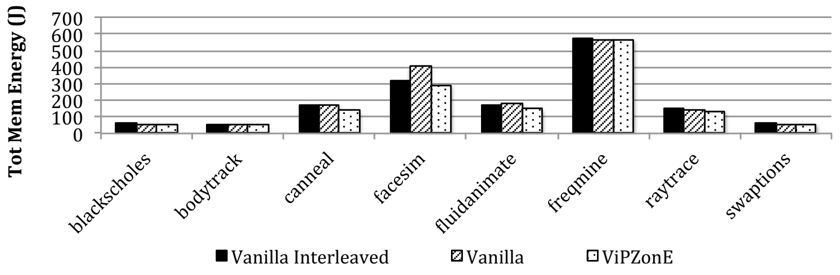
In the slower CPU setup, `Slow2` (results in Figs. 2.13b, 2.13d, 2.13f), running only a single workload thread, interleaving generally had little effect on memory power, runtime, and memory energy, because the processor/application were unable to stress the memory system. In these cases, interleaving could be disabled with no detrimental effect to performance. Note that power savings of disabling interleaving could be higher if the memory controller used effective power management on individual idle DIMMs, as opposed to global DIMM power management. A related work on power aware page allocation [111] also required interleaving to be disabled in order to employ effective DIMM power management. As interleaving remains an issue for general DIMM-level power management schemes, further investigation into the inherent tradeoffs and potential solutions is an open research direction.


(a) `Fast2` Memory Power


(b) `Slow2` Memory Power


(c) `Fast2` Runtime


(d) `Slow2` Runtime


(e) `Fast2` Memory Energy

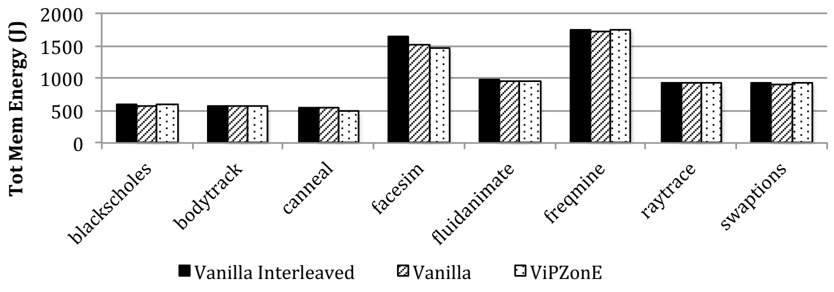

(f) `Slow2` Memory Energy

Figure 2.14: ViPZonE vs. vanilla and interleaved (Cint On, Rint On) vanilla Linux.

2.5.3 ViPZonE Analysis

In the evaluation of ViPZonE software, channel and rank interleaving were always disabled, as it was a prerequisite for functionality. The PARSEC benchmarks were not explicitly annotated (modified) for use with the ViPZonE front-end, although we have tested the full software stack for correct functionality. Instead, we emulated the effects of using all low-power allocation requests by using the default *Low Power First* physical page allocation policy described in Sec. 2.4.2.2. ViPZonE was benchmarked with two system configurations, namely `Fast2` (Fig. 2.14a, 2.14c, 2.14e) and `Slow2` (Fig. 2.14b, 2.14d, 2.14f). By using the two-DIMM configurations, we could better exploit memory power variability by including only the highest and lowest power two DIMMs from Fig. 2.8. Because we cannot harness idle power variability in our scheme, additional inactive DIMMs merely act as a "parasitic" effect on total memory power savings; with more DIMMs in the system, a greater proportion of total memory power/energy is consumed by idle DIMMs.

While ViPZonE does not explicitly deal with idle power management like other related works, it could be supplemented with orthogonal access coalescing methods which exploit deep low-power DRAM states. From the address mapping perspective, the nature of ViPZonE's zone preferences already implicitly allow more DIMMs to enter low-power (or, potentially off) modes by grouping allocations to a subset of the memory.

The results from the `Fast2` configuration indicate that ViPZonE can save up to 27.80% total memory energy for the `facesim` benchmark with respect to the vanilla software, also with interleaving disabled. Intuitively, our results make sense: benchmarks with higher memory utilization can better exploit active power variability between DIMMs. For this reason, lower-utilization benchmarks such as `blackscholes` gain no benefit from ViPZonE, just as they see no benefit from interleaving (refer to Sec. 2.5.2). With the slower CPU configuration, Figs. 2.14b, 2.14f indicate that there is a reduced benefit in power and energy from ViPZonE, for the same reasons. The reduced CPU speed, results in less stress being placed on the memory system, resulting in lower average utilization and a higher proportion of total memory power being from idleness.

It may initially surprise the reader to note that in some cases, vanilla interleaved roughly matches ViPZonE on the energy metric. This is due to the performance advantage of interleaving.

Because we currently have no way to combine ViPZonE with interleaving (although we propose possible solutions in Sec. 2.6), we use vanilla without interleaving as the primary baseline for comparison with ViPZonE. In other words, our primary baseline is on equal hardware terms with ViPZonE. The direct comparison with vanilla interleaved was included for fairness, as it merits discussion on whether a variation-aware scheme should be used over a conventional interleaved memory given the current state of DRAM memory organization. Nevertheless, we believe there is significant potential for further work on power variation-aware memory management, especially if there is a solution to allow interleaving simultaneously.

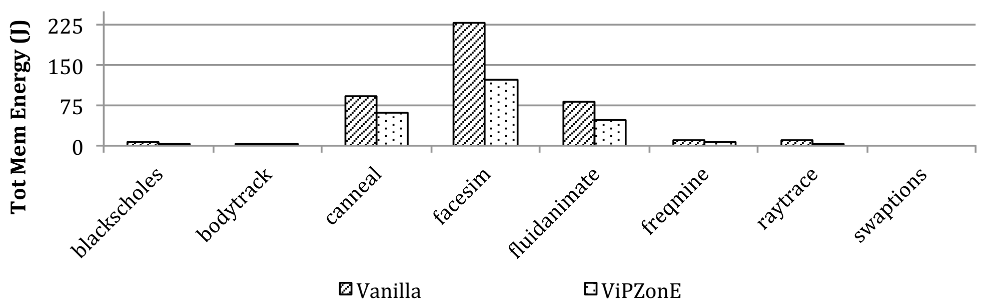
In all cases, ViPZonE running with channel and rank interleaving disabled achieved lower memory energy than the baseline vanilla software, with or without channel and rank interleaving.

2.5.4 What-If: Expected Benefits With Non-Volatile Memories Exhibiting Ultra-Low Idle Power

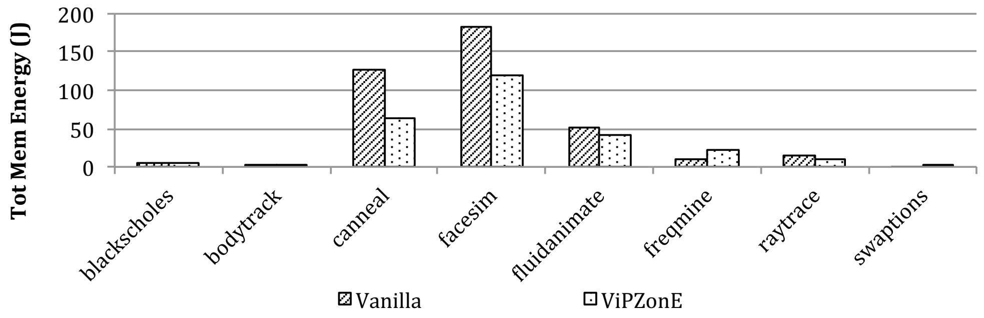
From the results of the ViPZonE comparison on our testbed, we speculate that the benefits of our scheme could be significantly greater with emerging non-volatile memory (NVM) technologies, such as STT-RAM, PCM, etc. We expect that there are two primary characteristics of NVMs that would make ViPZonE more beneficial: (1) extremely low idle power, thus eliminating its aforementioned parasitic effect on access power variability-aware solutions, and (2) potentially higher process variability with novel devices, leading to higher power variability that can be opportunistically exploited.

Thus, we present the results with the idle power component removed[10]. While this by no means an accurate representation of the realities of non-volatile memories, such as asymmetric write and read power/performance and potential architectural optimizations, this is meant to illustrate how active power variability can be better exploited without the parasitic idle power. The idle power was approximately 1.41 W for the two DIMMs used in the `Fast2` and `Slow2` configurations, specifically DIMMs `b1` and `c1` from Fig. 2.8. As can been seen in Fig. 2.15a and Fig. 2.15b, the overall memory energy benefits could increase dramatically, up to 50.69% for the `canneal`

[10]No performance figures are presented, as we did not actually run the system with real non-volatile memories.



(a) `Fast2` Memory Energy, idle energy removed



(b) `Slow2` Memory Energy, idle energy removed

Figure 2.15: ViPZonE vs. Vanilla Linux, "what-if" evaluation for potential benefits with NVMs (CInt Off, Rint Off).

benchmark. Although these numbers do not realistically represent the results with actual NVMs, as they were derived from our DRAM modules, they present a case for variability-aware solutions for future memories with low idle power and higher power variation.

2.5.5 Summary of Results

Table 2.7 summarizes the results from the evaluation of ViPZonE, as well as the theoretical "what-if" study for potential application to NVM-based systems. We expect that with emerging NVMs, the lack of a significant idle power component will result in ViPZonE getting significant energy savings for workloads with a variety of utilizations, even as the number of modules in the system increase. Thus, using variability-aware memory allocation instead of interleaving would likely be a promising option for future systems.

Table 2.7: Summary of ViPZonE results, with respect to vanilla software with channel and rank interleaving disabled

Metric	**Value (Benchmark)**
Max memory power savings, `Fast2` config	25.13% (`facesim`)
Max memory power savings, `Slow2` config	11.79% (`canneal`)
Max execution time overhead, `Fast2` config	4.80% (`canneal`)
Max execution time overhead, `Slow2` config	1.16% (`canneal`)
Max memory energy savings, `Fast2` config	27.80% (`facesim`)
Max memory energy savings, `Slow2` config	10.77% (`canneal`)
Max memory energy savings, `Fast2` config estimated, “NVM” (no idle power)	46.55% (`facesim`)
Max memory energy savings, `Slow2` config estimated, “NVM” (no idle power)	50.69% (`canneal`)

2.6 Discussion and Conclusion

In this work, we implemented and evaluated ViPZonE, a system-level energy-saving scheme that exploits the power variability present in a set of DDR3 DRAM memory modules. ViPZonE is implemented for Linux x86-64 systems, and includes modified physical page allocation routines within the kernel, as well as a new system call. User code can reduce system energy use by using a new variant of `malloc()`, only requiring the ViPZonE kernel and C standard library support. Our experimental results, obtained using an actual hardware testbed, demonstrates up to 27.80% energy savings with no more than 4.80% performance degradation for certain PARSEC workloads compared to standard Linux software. A brief “what-if” study suggested that our approach could yield greatly improved benefits using emerging non-volatile memory technology that consume no idle power, notwithstanding potentially higher power variability compared to DRAMs. As our approach requires that no channel or rank interleaving be used, we also included a comparison of four different interleaving configurations on our testbed to evaluate the impact of interleaving on realistic workloads.

The lack of interleaving support in the current implementation of ViPZonE is its primary drawback. It is a general problem facing DIMM-level power management schemes, and we believe finding good tradeoffs remains an open research question. We do not claim that ViPZonE is the best solution for all applications and systems. Rather, we think that it is an interesting demonstration of a novel memory management concept in a realistic setting, and motivates further research in this space.

There are several opportunities for further research with ViPZonE. First, given the ability to co-design hardware and software, it might be possible to combine the benefits of interleaving for performance while exploiting power variation for energy savings. We can imagine a few ways this could be done. One solution would use a modified memory controller that interleaves different groups of DIMMs independently. This compromise would allow for performance and potential energy savings somewhere between the current interleaving vs. ViPZonE scenario, but would still be a static design-time or boot-time decision. This could be useful in systems that already have clustered memory, such as non-uniform memory access (NUMA) architectures.

Alternatively, hypothetical systems with disparate memory device technologies side-by-side (e.g., a hybrid DRAM-PCM memory as in [142]) may discourage interleaving across device types due to different access power, latency, read/write asymmetry, and data volatility. In this case, interleaving could still be used within each cluster of homogeneous memory technology, and each such cluster could be used as a single zone for ViPZonE. The result would be ViPZonE becoming heterogeneity-aware as a generalization of variability-awareness.

A more radical idea which may allow the full benefits of interleaving alongside ViPZonE would likely require a re-design of the DIMM organization to allow individual DIMMs, where each rank is multi-ported, to occupy multiple channels. However, the major issue we forsee with this is a much higher memory cost due to the multiplied pin requirements.

Aside from enabling interleaving alongside variation-aware memory management, ViPZonE could potentially be improved on the software side. Adding compiler support could take some of the burden off the programmer while expanding the scope to include static program data. Variability-aware page migration schemes might yield further improvements in energy efficiency by augmenting our static allocation-time hints. Our approach could likely be complemented by several other power-aware methods mentioned in Sec. 2.1.1. A simulation study of ViPZonE with detailed models of non-volatile memories could give a better idea of the benefits in the future, where power and delay variation are likely to be higher and there is negligible idle power.

We believe ViPZonE makes an effective case for further research into the Underdesigned and Opportunistic computing paradigm [75] with the goal of improving energy efficiency of systems, while lowering design cost, improving yield, and recovering lost performance due to conventional guardbanding techniques.

CHAPTER 3

DPCS: Saving Energy in SRAM Caches with Dynamic Power/Capacity Scaling

Fault-tolerant voltage-scalable (FTVS) SRAM cache architectures are a promising approach to improve energy efficiency of memories in presence of nanoscale process variation. Complex FTVS schemes are commonly proposed to achieve very low minimum supply voltages, but these can suffer from high overheads and thus do not always offer the best power/capacity tradeoffs. We observe on our 45nm test chips that the "fault inclusion property" can enable lightweight fault maps that support multiple run-time supply voltages.

Based on this observation, we propose a simple and low-overhead FTVS cache architecture for power/capacity scaling. Our mechanism combines multi-level voltage scaling with optional architectural support for power gating of blocks as they become faulty at low voltages. A static (SPCS) policy sets the run-time cache VDD once such that a only a few cache blocks are may be faulty in order to minimize the impact on performance. We describe a static (SPCS) policy and two alternate dynamic power/capacity scaling (DPCS) policies that opportunistically reduce the cache voltage even further for energy savings.

This architecture achieves lower static power for all effective cache capacities than a recent more complex FTVS scheme. This is due to significantly lower overheads, despite the inability of our approach to match the min-VDD of the competing work at a fixed target yield. Over a set of SPEC CPU2006 benchmarks on two system configurations, the average total cache (system) energy saved by SPCS is 62% (22%), while the two DPCS policies achieve roughly similar energy reduction, around 79% (26%). On average, the DPCS approaches incur 2.24% performance and 6% area penalties.

Collaborators:

- Dr. Abbas BanaiyanMofrad, then at UC Irvine
- Prof. Nikil Dutt, UC Irvine
- Prof. Alex Nicolau, UC Irvine
- Prof. Puneet Gupta, UCLA

Source code and data are available at:

- `https://github.com/nanocad-lab?&q=dpcs`
- `http://nanocad.ee.ucla.edu`

3.1 Introduction

Moore's Law has been the primary driver behind the phenomenal advances in computing capability of the past several decades. Technology scaling has now reached the nanoscale era, where the smallest integrated circuit features are only a couple orders of magnitude larger than individual atoms. In this regime, increased leakage power and overall power density has ended ideal Dennard scaling, leading to the rise of "dark silicon" [43].

Owing to the extreme manufacturing control requirements imposed by nanoscale technology, the effects of process variation on reliability, yield, power consumption, and performance is a principal challenge [75] and is partly responsible for the end of Dennard scaling. The typical solution is to leave design margins for the worst-case manufacturing outcomes and operating conditions. However, these methods naturally incur significant overheads.

Memory structures are especially sensitive to variability. This is because memories typically use the smallest transistors and feature dense layouts, comprise a significant fraction of chip area, and are sensitive to voltage and temperature noise. Memories are also major consumers of system power, are not very energy proportional [39, 40], and are frequent points of failure in the field [143, 144]. It is clear that the design of the memory hierarchy needs to be reconsidered with the challenges brought about in the nanoscale era.

One way to reduce the power consumption of memory is to reduce the supply voltage (VDD). Static power, dominated by subthreshold leakage current, is a major consumer of power in memories [145] and has an exponential dependence on supply voltage [22]. Since leakage often constitutes a major fraction of total system power in nanoscale processes [146], even a minor reduction in memory supply voltage could have a significant impact on total chip power consumption.

Unfortunately, voltage-scaled SRAMs are susceptible to faulty behavior. Variability in SRAM cell noise margins, mostly due to the impact of random dopant fluctuation on threshold voltages, results in an exponential increase in probability of cell failure as the supply voltage is lowered [147]. This has motivated research on fault-tolerant voltage-scalable (FTVS) SRAM caches for energy-efficient and reliable operation.

Many clever FTVS SRAM cache architectures use complex fault tolerance methods to lower min-VDD while meeting manufacturing yield objectives. Unfortunately, these approaches are limited by a manifestation of Amdahl's Law [148] for power savings. Large and flexible fault maps combined with intricate redundancy mechanisms cost considerable power, area, and performance. It is important that designers account for these overheads, as they ultimately limit the scalability of such approaches. In this chapter, we make several contributions:

- The SRAM memories on our custom 45nm SOI embedded systems on chip are characterized. We observe the *Fault Inclusion Property*: particular bit cells that fail at a given supply voltage will still be faulty at all lower voltages.
- A new *Static Power vs. Effective Capacity* metric is proposed for FTVS memory architectures, which is a good indicator of run-time power/performance scalability and overall energy efficiency.
- A simple and low-overhead multi-VDD fault tolerance mechanism combines voltage scaling of the data array SRAM cells with optional power gating of faulty data blocks at low voltage.
- We propose static (*SPCS*) and dynamic (*DPCS*) *Power/Capacity Scaling*, two variants of a novel FTVS scheme that significantly reduces overall cache and system energy with minimal performance and area overheads.
- SPCS and DPCS achieve lower static power at all scaled cache capacities than a more complex FTVS cache architecture [149] as well as per-cache way power gating.

This chapter is organized as follows. In Sec. 3.2, we summarize related work and differentiate our contributions. In Sec. 3.3 the Fault Inclusion Property is demonstrated experimentally on a set of test chips. We make the case for the Static Power vs. Effective Capacity metric in Sec. 3.4 and then introduce the Power/Capacity Scaling architecture in Sec. 3.5. Sec. 3.6 presents the evaluation methodology, Sec. 3.7 presents the analytical results, and Sec. 3.8 describes the simulation results. The chapter is concluded in Sec. 3.9.

3.2 Related Work

There is a rich body of literature in circuit, architecture, and error correction coding techniques for FTVS deep-submicron memories. A summary of related work is provided here, with emphasis on architectural solutions. The reader may refer to [150] for a more complete survey of architectural techniques for improving cache power efficiency.

3.2.1 Leakage Reduction

Two of the best-known architectural approaches are those generally based on Gated-VDD [151, 152] and Drowsy Cache [145, 153] approaches. The former dynamically resizes the instruction cache by turning off blocks which are not used by the application, exploiting variability in cache utilization within and across applications. The latter utilizes the alternative approach of voltage scaling idle cache lines, which yields good static power savings without losing memory state. Neither approach improves dynamic power nor accounts for the impact of process variation on noise margin-based faults, which are greatly exacerbated at low voltage [146, 147]. This issue particularly limits the mechanism of Flautner et al. [145].

3.2.2 Fault Tolerance

In the fault-tolerance area, works targeting cache yield and/or min-VDD improvement include error correction codes (ECC) and architectural methods [154–163], none of which explicitly address energy savings as an objective. A variety of low-voltage SRAM cells that improve read stability and/or writability have also been proposed, e.g. 8T [164] and 10T [165], but they have high area overheads compared to a 6T design.

Schemes that use fault-tolerance to achieve lower voltage primarily for cache power savings include Hussain et al. [166], Wilkerson et al. [167], Abella et al. [168], Sasan et al. [109], two very similar approaches from Ansari et al. [169] and BanaiyanMofrad et al. [149], and several others [170–174]. All of these approaches try to reduce the minimum operable cache VDD with yield constraints by employing relatively sophisticated fault tolerance mechanisms, such as address remapping, block and set-level replication, etc. They are also similar in that they either reduce the effective cache capacity by disabling faulty regions as VDD is reduced (e.g., FFT-Cache [149]), or boost VDD in "weak" regions as necessary to maintain capacity (e.g., Sasan et al. [170]). Han et al. [172] and Kim and Guthaus [175] both utilize multiple memory supply voltages. Ghasemi et al. [176] proposed a last-level cache with heterogeneous cell sizes for more graceful voltage/capacity tradeoffs. Finally, variation-aware non-uniform cache access (NUCA) architectures have been proposed [177, 178]. Hijaz et al.'s approach [178] mixes capacity tradeoffs with latency penalties of error correction.

There are two recent papers that contain some similarities to the concepts in this chapter, yet were developed concurrently and independently of ours [179, 180]. Mohammad et al. [179] discuss power/capacity tradeoffs in FTVS caches and a similar architectural mechanism, primarily focusing on yield improvement and quality/power tradeoffs. Ferreron et al. [180] focuses on system-level impacts of block disabling techniques in a chip-multiprocessor with shared caches with parallel workloads.

3.2.3 Memory Power/Performance Scaling

With energy proportionality becoming a topic of interest in the past several years [39, 40], there have been several works that target main memory. Fan et al. [181] studied the coordination of processor DVFS and memory low power modes. In MemScale [182], the authors were some of the first to propose dynamic memory frequency scaling as an active low-power mode. Independently of MemScale, David et al. [183] also proposed memory DVFS. MemScale was succeeded by CoScale [184], which coordinated DVFS in the CPU and memory to deliver significantly improved system-level energy proportionality.

3.2.4 Novelty of This Work

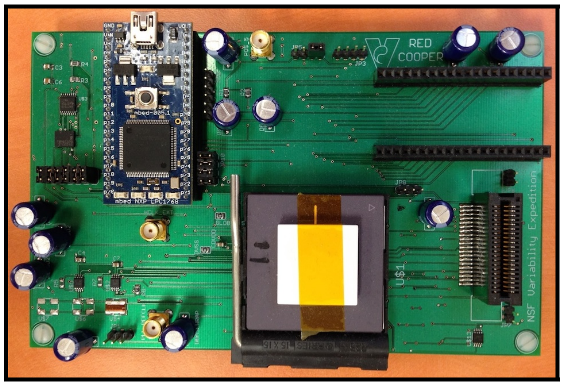
Our approach is different from the related works as follows. The circuit mechanisms are similar to those of Gated-VDD [151] and Drowsy Cache [145], but are combined with fault tolerance to allow lower voltage operation using 6T SRAM cells, although the proposed scheme can also be used with any other cell design. To the best of our knowledge, our scheme is the first to use voltage scaling on data bit cells only, along with optional power gating blocks as they become faulty for additional energy savings. Due to architecture, floorplanning, and layout considerations imposed by per-block power gating, we also discuss the advantages and disadvantages of this feature. Our approach emphasizes simplicity to get good energy savings using low overhead multi-VDD fault maps that can be used for static (SPCS) or dynamic (DPCS) power/capacity scaling. SPCS and DPCS both achieve dynamic and static energy savings, as VDD is not boosted for accesses.

The work in this chapter could be combined with other innovations. Although soft errors are not currently handled, this scheme can be supplemented with ECC. Circuit-level approaches to coping with aging are orthogonal to this work and could be incorporated as well. Our insights could be combined with those of Mohammad et al. [179] and Ferreron et al. [180] to enable knobs for dynamic power/capacity/quality scaling. Although not explored in this chapter, we believe that DPCS can also be used to improve system-level energy proportionality by coordinating with existing CPU and main memory DVFS approaches. We leave these possibilities to future work.

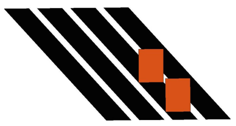
3.3 The SRAM Fault Inclusion Property

Once the noise margin of a memory cell collapses at low voltage, continuing to reduce voltage further will not restore its functionality. We refer to this behavior as the *Fault Inclusion Property*: any faults that occur at a given supply voltage will strictly be a subset of those at all lower voltages. Put more formally:

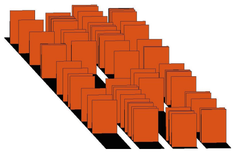
$$\begin{aligned}
&\text{Let } f_i(v) = \begin{cases} 1, \text{ if memory cell } i \text{ is faulty at supply voltage } v \\ 0, \text{ otherwise.} \end{cases} \\
&\text{where } 1 \leq i \leq n, \text{ and } n \text{ is the number of bit cells in the memory.} \\
&\text{Fault Inclusion Property: } f_i(v) \geq f_i(u) \text{ for } v \leq u. \qquad (3.1) \\
&\text{Let the fault map } \mathbf{F}(v) = (f_1(v), f_2(v), \dots, f_i(v), \dots, f_n(v)) \\
&\text{be a length-}n\text{bit-vector. Then } w[\mathbf{F}(v)] \geq w[\mathbf{F}(u)] \\
&\text{for } v \leq u\text{, where } w[\cdot] \text{ denotes the Hamming weight.}
\end{aligned}$$



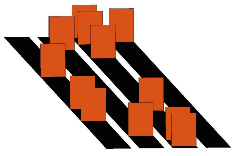
(a) “Red Cooper” 45nm SOI test chip and board



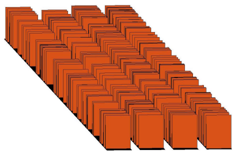
(b) Faults at 550 mV



(d) Faults at 525 mV



(c) Faults at 500 mV



(e) Faults at 475 mV

Figure 3.1: Graphical depiction of faulty byte locations – represented as vertical orange rectangles – that illustrate the fault inclusion property. Results are from a 45nm SOI test chip SRAM bank. Similar trends were observed on our other test chips.

No prior work on FTVS SRAM cache architectures appears to take advantage of this property. With typical existing approaches, there is no way to operate the memory at additional intermediate voltage levels without including extra fault maps for each run-time voltage level. A key insight in this work is that complete fault map duplication for each run-time supply voltage is unnecessary.

To verify the above formulation, tests were performed on four ARM Cortex M3-based “Red Cooper” test chips manufactured in a commercial 45nm SOI technology [185,186] in collaboration with colleagues in the NSF Variability Expedition. Each test chip had two 8 kB SRAM scratchpad banks and no off-chip RAM. A board-level mbed LPC1768 microcontroller accessed the SRAM and controlled all chip voltage domains through JTAG. The board and a test chip are shown in Fig. 3.1a.

With each test chip’s CPU disabled, March Simple-Static (SS) tests [187] were run on both banks using the mbed to characterize the nature of faults as the array VDD was reduced. For each SRAM bank on each chip, a test suite was repeated five times. In each run, VDD scaled down one step from nominal 1 V in 25 mV increments. For each voltage level, faulty SRAM locations were logged at byte granularity.[1] No distinction was made between different underlying physical causes of failure (read stability, writability, retention failure, inter-cell interference, etc.). As dynamic

[1]Due to run-time and storage limitations of our mbed-based testing, we manually verified the fault inclusion property at the bit-level, while recording fault maps at the byte-level.

failures are not the focus of this study, the test chip was operated at a 20 MHz clock to minimize the chance of delay faults occurring in the SRAM periphery logic. This is because the test chip uses a single voltage rail for the memory cells and periphery.

As expected, faults caused by voltage scaling obeyed the fault inclusion property. This trend is depicted graphically in Figs. 3.1b, 3.1d, 3.1c, and 3.1e for one of the test chips. Faulty byte locations were consistent in each unique SRAM bank at each voltage level. The patterns could be verified with repeated testing after compensating for noise, indicating that the faults were not caused by soft errors or permanent defects, but rather caused by variability in cell noise margins. These results suggested the use of a compact fault map such that multiple VDDs can be efficiently supported.

3.4 A Case for the "Static Power vs. Effective Capacity" Metric

As previously mentioned, related works in FTVS cache architecture commonly use min-VDD as a primary metric of evaluation. While supply voltage is a very important control knob for power, it is only one factor that determines the static and dynamic energy consumption of a cache memory. Most FTVS schemes, including ours, require a portion of the memory to operate at full VDD or otherwise be resilient for guaranteed correctness.

3.4.1 Amdahl's Law Applied to Fault-Tolerant Voltage-Scalable Caches

When evaluating power reduction from voltage scaling, it is important to consider the proportion of cache components operating at high VDD. This is due to a manifestation of Amdahl's Law [148] when interpreted for power and energy. Note the following relationship, which is similar to the traditional version of Amdahl's Law for speedup [16], but for power reduction instead via fault-tolerant (*FT*) voltage scaling (*VS*).

$$\text{PowerReduction}_{\text{FTVS, overall}} = \frac{1}{1 - \text{Frac.}_{\text{VS}} + \text{Frac.}_{\text{FT overhead}} + \frac{\text{Frac.}_{\text{VS}}}{\text{PowerReduction}_{\text{VS}}}} \tag{3.2}$$

Note the additional $\text{Frac.}_{\text{FT overhead}}$ term in the denominator, which accounts for the additional fault tolerance logic needed to scale voltage to the desired min-VDD on part of the the memory (Frac._{VS}). In fact, this modified formulation of Amdahl's Law also holds more generally whenever constant overheads are incurred for speedup or power reduction. This interpretation of Amdahl's Law might be overlooked when trying to achieve a lower min-VDD using complex fault tolerance schemes. Thus, supply voltage alone should not receive too much emphasis in FTVS techniques.

For example, consider two competing FTVS approaches, *Scheme 1* and *Scheme 2*, where only the data array voltage can be scaled. Both Scheme 1 and Scheme 2 have the same cache size, block size, associativity, etc. Assume method Scheme 2's total tag array power overhead is 20% of the nominal data array power (full VDD) due to a large and complex fault map, and Scheme 1's tag power overhead is 5%, thanks to a smaller and simpler fault map. The baseline cache has a tag array power overhead of only 3% because it has no fault map.

Let the data array voltage be scaled independently for Scheme 1 and Scheme 2 such that Scheme 2's voltage is lower than Scheme 1's due to better fault tolerance. Suppose the data array leakage power in Scheme 1 is now 30% of its nominal value, and the data array leakage power in Scheme 2 is now 20% of its nominal value. Scheme 2 will save 61.1% of static power against the baseline cache, while Scheme 1 will save 66.0%, despite operating at a higher data array voltage.

3.4.2 Problems With Yield Metric

Another common metric is memory yield in terms of functional reliability, performance, power, etc. Often, the min-VDD is determined for a particular cache configuration based on expected fault probabilities, the particular fault tolerance mechanism, and a desired target yield such as 99%. While one can claim that a particular scheme is functional within the design envelope, definitions of functionality vary considerably. For example, one cache achitecture may be considered functional if it simply operates in a "correct" manner, regardless of power consumption. Another architecture may define the cache to be functional only if at least 50% of blocks are non-faulty. Thus, in general, min-VDD/yield should not be overemphasized. The min-VDD metric can be more useful for caches that are on the same voltage domain as the processor core. However, in the future, as indicated by industry trends such as Intel's fine-grain voltage regulation (FIVR) design in its

Haswell architecture [188], this is likely not to be the case.

3.4.3 Proposed Metric

We believe that a new metric, *Static Power vs. Effective Capacity*, should be used to guide the design of FTVS cache architectures in addition to the other metrics. This metric accounts for the supply voltage and the efficacy and overheads of the particular fault tolerance or capacity reduction mechanism (e.g., power gating).

Simple FTVS schemes can fare similarly or even better than complex ones in terms of power, performance, and/or area, all with less design and verification effort. This is because the overall failure rate rises sharply in a small voltage range as shown earlier in Fig. 3.1. The complexity required for tolerating such a high memory failure rate may not be worth the small array power savings from an incrementally lower voltage.

With this metric, we focus on static power to allow for analytical evaluation. This is a reasonable simplification in the case of large memories where static power constitutes a large fraction of memory energy. We apply this metric during our evaluation in Sec. 3.7.

Note that static power vs. effective capacity is not necessarily a good metric for choosing the capacity of a cache at design-time and nominal voltage. Nor does the metric imply that smaller caches are more efficient in general. Rather, this metric is meant to capture the *run-time scalability* of the cache power and performance while operating below its designed maximum capacity. This is useful because the full cache capacity may not be needed at all times to deliver good performance, and hence, its capacity can be temporarily reduced to lower power and improve energy efficiency.

3.5 Power/Capacity Scaling Architecture

Our scheme has two main components: the architectural mechanism, described in Sec. 3.5.1, and the policy. We propose a static policy (SPCS) and two different dynamic policies (DPCS Policy 1 and DPCS Policy 2), described in Secs. 3.5.2, 3.5.3.1, and 3.5.3.2, respectively. All three policies share the same underlying hardware mechanism.

3.5.1 Mechanism

Industry trends point towards finer-grain voltage and frequency domains in future chips. For example, Intel has introduced fine-grain voltage regulation (FIVR) that is partly on-chip and on-package with its Haswell architecture [188]. This allows for many voltage domains on the chip on the level of per-core, per-L3 cache, etc. Thus, it is reasonable that future chips could support separate voltage rails for each level of cache in order to further decouple logic V_{min} from memory V_{min}, especially if architectures can exploit this feature.

The power/capacity scaling mechanism primarily consists of a lightweight fault map and a circuit for globally adjusting the VDD of the data cells. The data array periphery, metadata (`Valid`, `Dirty`, `Tag`, etc.) array cells, metadata array periphery, and the processor core are all on a separate voltage domain running at nominal VDD, where they are assumed to be never faulty. Data and metadata arrays otherwise use identical cell designs. Voltage is not boosted for data access, granting both dynamic and static energy savings. To bridge the voltage domains of the data array with its periphery, the final stage of the row decoder is also used as a downward level-shifting gate to drive the wordline. Similarly, column write drivers also are downward level-shifting, while sense amplifiers used in read operations restore voltage levels to the full nominal VDD swing. This circuit approach is validated by a very recent SoC design from Samsung [189], which independently arrived at the decision to employ a dual-rail design to decouple logic and bitcell supply voltage.

Overall access time may be affected by up to 10% in the worst case at low voltage, as found later in Sec. 3.6.3. This is because the impact of reduced cell voltage is only one part of the overall cache access time, and near-threshold operation is avoided. Note that voltage boosting during cell access could be utilized to allow even lower array voltages than used in this chapter, as the proposed design is limited by SRAM read stability.

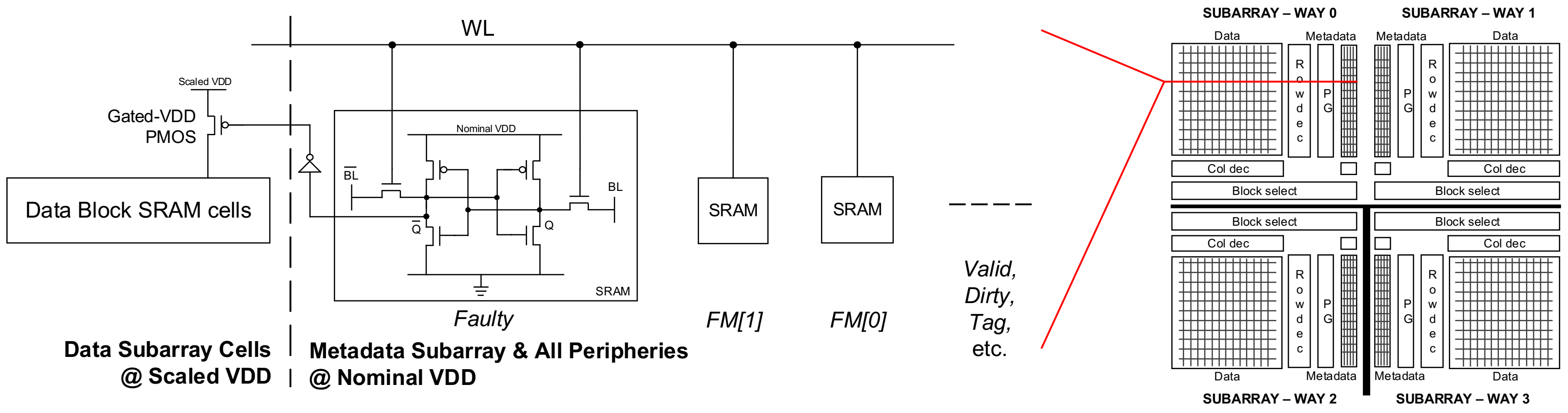


Figure 3.2: Power/Capacity Scaling cache architecture concept with optional power gating of faulty blocks. If this feature is omitted by the designer, the `Faulty` bit SRAM cell does not drive a Gated-VDD transistor for the data block and the cache has relaxed floorplan and layout constraints.

If the number of voltage domains remains constrained, a version of the architecture with a single voltage rail for the core and all peripheries and a shared voltage rail for all cache data arrays is also feasible. However, this can restrict DPCS capabilities for energy savings by coupling cache voltages together while limiting voltage scaling policies. The overall architecture is depicted in Fig. 3.2.

3.5.1.1 Fault Map

The low-overhead fault map includes two fields for each data block that are maintained in the corresponding nearby metadata subarray in addition to the conventional bits (`Valid`, `Dirty`, `Tag`, etc.), as shown in Fig. 3.2. The first entry is a single `Faulty` bit, which indicates whether the corresponding data block is presently faulty. Blocks marked as `Faulty` can never contain valid or dirty data and are unavailable for reading and writing. The second field consists of several fault map (`FM`) bits, which encode the lowest non-faulty VDD for the data block. For V allowed data VDD levels, $K = \lceil log_2(V+1) \rceil$ `FM` bits are needed to encode the allowed VDD levels (assuming the fault inclusion property). Fig. 3.2 depicts the $V = 3$ configuration for a small 4-way cache, requiring one `Faulty` and two `FM` bits per block.

3.5.1.2 Power Gating of Faulty Blocks

Power gating transistors can be used to attain additional power savings for faulty data blocks that occur at reduced voltage. Blocks may span one or more subarray rows. When a block's `Faulty` bit is set in the metadata array, a downward level-shifting inverter power gates the block's data portion. We assume that the power gating implementation is the gated-PMOS configuration from [151], chosen because it has no impact on cell read performance, negligible area overhead [151,152], and good energy savings. A power-gated block at low VDD is modeled as having zero leakage power. This is a reasonable assumption because the block would likely be power gated at a low voltage that caused it to be faulty in the first place. We verified the functional feasibility of this mechanism via SPICE simulations.

Designers might choose to omit power gating of faulty blocks for two major reasons. First, if power gating is used, the metadata subarrays should be directly adjacent to their corresponding data subarrays as shown by Fig. 3.2. This is so the `Faulty` bit can control the power gate mechanism and the row decoder can be shared between subarrays, although the latter benefit is not explicitly modeled. However, this constrains the cache floorplan.

Second, *by allowing for per-block power gating, both power rails must be routed in the wordline direction.* Unfortunately, this is not a common approach in industry, where a thin-cell SRAM cell is preferred for above-threshold operation (e.g., as seen by layouts in [94, 190]). However, others have used wordline-oriented rails successfully in a variety of low-voltage design scenarios [152, 165, 191–193]. Nevertheless, there are other important factors to consider when deciding on power rail orientation.

Without advocating for either approach, we believe that fine-grain power gating of faulty blocks could be viewed as an additional benefit from wordline-oriented rails. Without loss of generality, in the rest of the chapter we assume the presence of per-block power gating, except in Secs. 3.7.2 and 3.8.2, where we directly compare the two approaches.

3.5.1.3 Transitioning Data Array Voltage

It is the responsibility of the power/capacity scaling policy, implemented in the cache controller, to ensure the `Faulty` and `Valid` bits are set correctly for the current data array VDD. After all blocks' `Faulty` bits are properly set for the target voltage, the data array VDD transition can occur. The controller must compare each block's `FM` bits with an enumerated code for the intended VDD level. Eqn. 3.3 captures this logical relationship for all V. We assume that $V = 3$ levels are allowed throughout this chapter, corresponding to $K = 2$ `FM` bits. Note that our fault map approach scales well for more intermediate voltage levels if needed.

$$\texttt{Block}_{\texttt{i}}\texttt{.Faulty} = \texttt{TRUE} \text{ iff } \left(\texttt{VDD}_{\texttt{data, global}}[K-1{:}0] \leq \texttt{Block}_{\texttt{i}}\texttt{.FM}[K-1{:}0]\right) \tag{3.3}$$

The procedure to scale array VDD is described by Alg. 1. A voltage transition has a delay penalty to update the `Faulty` bits and then set the voltage. The cache controller must read out the entire fault map set-by-set, with each way in parallel. Next, it compares the `FM` bits for the

Algorithm 1 Voltage transition procedure for the DPCS architecture.

```
//Inputs: $NextVDD$ (integer index), $VoltageTransitionPenalty$ (integer, in cycles)
//Outputs: SRAM data array supply voltage scaled corresponding to $NextVDD$ after delay of
$VoltageTransitionPenalty$ clock cycles
Halt cache accesses (by buffering or stalling)
for Each $Set$ in $CacheSets$ do
    for Each $Assoc\ Block$ in $Set$ do
        //Each loop iteration in parallel
        Read metadata bits for $Block$
        if $NextVDD[1:0] \leq Block.FM[1:0]$ then
            if $Block.Valid$ and $Block.Dirty$ then
                Write back $Block$
            end if
            Invalidate $Block$
            $Block.Faulty \leftarrow$ TRUE
        else
            if $Block.Faulty$ then
                $Block.Faulty \leftarrow$ FALSE
            end if
        end if
    end for
end for
$CurrVDD \leftarrow NextVDD$
Scale cache data array SRAM cells to voltage corresponding to $CurrVDD$
```

set with the target array `VDD` bits. Finally, it writes back the correct `Faulty` bits for each block in the set. We assume that it takes two cycles to do this for each set. After all `Faulty` bits are updated, the voltage regulator can adjust the data array VDD, which takes `VoltageRegulatorDelay` clock cycles. The total `VoltageTransitionPenalty` in Alg. 1 is then equal to $2 * \texttt{NumSets} + \texttt{VoltageRegulatorDelay}$ clock cycles. Note that `VoltageTransitionPenalty` in Alg. 1 does not include the time to write back any cache blocks. The performance and energy impact of any such writebacks are captured accurately in the gem5 cache model and simulation framework leveraged in Sec. 3.8.

3.5.1.4 Cache Operation In Presence of Faulty Blocks

Since some blocks may be faulty at any given supply voltage, the cache controller must consult the fault information during an access. Neither a hit nor a block fill are allowed to occur on a block that is marked as `Faulty`. Thus, all blocks that are currently `Faulty` must be marked as not `Valid` and not `Dirty`, and their `Tag` bits should be zeroed. When fulfilling a cache miss, the traditional LRU scheme is applied on each set, with the special condition that any blocks marked as `Faulty` are omitted from LRU consideration. This can change the effective associativity on each set.

3.5.1.5 3C+F: Compulsory, Capacity, Conflict, and Faulty Misses

Cache misses can be categorized into three buckets, commonly known as the "3 Cs": compulsory, capacity, and conflict misses [16].[2] Note that in practice, it is not feasible to classify misses in this fashion at run-time, as the full access history for all referenced data must be maintained.

When disabling faulty blocks, it is not clear how resulting misses should be classified. One pitfall would simply attribute misses caused by faulty blocks as capacity misses. However, faulty blocks also reduce local associativity. Moreover, consider blocks that are only temporarily faulty during execution with DPCS. An evicted block from a faulty set might be referenced later, causing a miss, even though the set may no longer contain faulty blocks.

Thus, a fourth category of misses can be defined for theoretical purposes. A *faulty miss* is one that occurs because the referenced data was evicted or replaced earlier from a set containing faulty blocks. To be considered a faulty miss, the reference should otherwise have hit in a cache where faulty blocks never previously occurred in the set. Note that disabling faulty blocks in the L1 cache can affect miss behavior in the L2 cache. For simplicity, we ignore inter-cache dependencies in the 3C+F classification.

3.5.1.6 Variation-Aware Selection of Allowed Runtime Voltage Levels

The proposed mechanism requires that each set must have at least one non-faulty block at all allowed voltages, as there is no set-wise data redundancy. This is the constraint that limits the min-VDD.[3] Higher associativity and/or smaller block sizes naturally result in lower min-VDD as shown later in Sec. 3.8.1, but they incur other design tradeoffs. Nevertheless, as demonstrated

[2]In this work, for simplicity, we do not consider multiprocessor machines, so the "4th C" for compulsory misses is not included.

[3]If uncacheable blocks are permitted, then the voltage can be lowered further.

later in the evaluation, a good power/capacity tradeoff can be achieved even for 4-way caches with blocks of 64 bytes.

The allowed run-time voltage levels for the SPCS and DPCS policies may be decided at design-time by margining for yield, at test-time to opportunistically remove some yield constraints, or at run-time, which can also account for aging effects. However, the design-time approach falls short of the other two, as it cannot exploit the individualized outcome of each chip. The test-time approach might considerably increase manufacturing cost. In this work, run-time choice of VDD levels is used (opposed to the design-time choice from our prior work [9]). This eliminates yield margins by customizing the desired voltage levels based on the manifested fault patterns on each SRAM array. Note that voltage margins may still be left for noise resilience.

3.5.1.7 Populating and Maintaining Fault Maps with BIST

To populate the cache fault maps, one can use any standard built-in self-test (BIST) routine that can detect faults with minimum granularity of a single block (e.g., the march tests from Sec. 3.3). The BIST routine can then apply the fault inclusion property to compress the representation of the fault maps by only encoding the minimum non-faulty voltage for each block.

A drawback of fault map approaches in general is the BIST run-time and storage overhead. The proposed approach incurs longer testing time than typical single-voltage solutions. The overall testing time is $O(N*V)$, where N is the size of the cache, and V is the number of run-time voltage levels. However, the actual run-time complexity is likely less than this, as fewer blocks must be tested at successively lower voltages due to the fault inclusion property.

One approach is to run BIST at every system startup, which requires no permanent fault map storage and allows aging effects such as bias temperature instability (BTI) to be considered. Particular data patterns stored in the SRAM cells are not expected to have a major impact on aging caused by BTI. This is because data tends to only briefly “live” in the cache and is assumed to be random across many workloads and over time. If data dependence does result in a noticeable impact on aging, the DPCS architecture can compensate with more frequent fault map characterizations.

Note that the fault inclusion property is still valid under aging conditions that affect threshold voltage (such as BTI or hot carrier injection). This is because static noise margins are strongly dependent on threshold voltages and decrease monotonically with VDD [147].

If testing time is an issue, the impact could be partly mitigated by proceeding concurrently with other system startup procedures, e.g., DRAM fault testing and initialization of peripheral devices. If the system is rarely or never power cycled, then BIST can be run periodically during run-time using a temporary way-disabling approach resembling that in [162]. Owing to the relatively low frequency and duration of such testing, the impact of BIST on overall system performance is expected to be small.

If on-chip or off-chip non-volatile memory is available, the low-overhead fault maps could be characterized at test-time or first power-up and stored permanently (similar to online self-test methods from Li et al. [194]). This has the advantage of eliminating run-time BIST and its associated performance overheads, but forgoes any compensation for aging aside from design guardbanding.

Idle low-power states are another consideration for the fault map design. In the event of a complete core shutdown without state retention, the fault maps must be: (1) maintained in the tag arrays, costing idle power; (2) written to non-volatile storage; or (3) sacrificed, in which case BIST must be run again when the core is brought online. In the first case, tag array static power can limit the effectiveness of long-term core shutdowns. In the latter two cases, there is a significant energy and performance cost to shut down a core for a short duration. We leave the coordination of DPCS and CPU/memory power states to future work.

3.5.2 Static Policy: SPCS

In the static policy, the lowest VDD level is used on a per-chip basis that has at least 99% effective capacity (also subject to the yield constraint described in Sec. 3.5.1.6). This is set once for the entire system run-time. The only performance overhead to this policy is due to any additional misses that may be caused by the few faulty blocks.

The primary benefit of using SPCS is that voltage guardbands could easily be reduced to suit the unique manufactured outcome of each cache, while only minor modifications to the cache controller and/or software layers are needed. Power can be reduced with negligible performance impact. Because the SRAM cell and block failure rates rise exponentially as the supply voltage is lowered, additional voltage reduction beyond the 99% capacity point brings more power savings, but potentially a significant loss in performance. This phenomenon is described in more detail during the analytical evaluation in Sec. 3.7, and is one reason why more sophisticated fault tolerance methods get diminishing improvements from voltage scaling.

3.5.3 Dynamic Policies: DPCS

A basic guiding principle behind cache operation is that programs tend to access data with spatial and temporal locality. If a program accesses a large working set of data in a short period of time, then large caches are likely to improve performance. Conversely, if a smaller working set is accessed during an equal length of time, then the cache capacity is less important. Note that in both cases, it is important that data is reused for the cache to be effective.

For these reasons, the static power/capacity scaling (SPCS) policy described in Sec. 3.5.2 can be too conservative. This is because the full cache capacity may be overkill for a given application in a particular phase of execution. The primary motivation to consider a dynamic power/capacity scaling (DPCS) policy over SPCS is to exploit these situations when the whole cache is unnecessary to deliver acceptable performance.

The proposed DPCS policies described next in Secs. 3.5.3.1 and 3.5.3.2 are only two possibilities among many. In both cases, we emphasize simple policies that can be easily implemented.

3.5.3.1 DPCS Policy 1: Access Diversity-Based

This DPCS policy (described by Alg. 2) tries to balance energy efficiency with performance by using spatial locality as the primary guiding principle. At the end of every time interval, if the fraction of *available cache capacity that was touched* falls below some fixed `LowThreshold` (`LT`), the DPCS policy reduces voltage and sacrifices cache capacity for the next interval. Conversely, if the cache pressure is high, then DPCS boosts voltage for the next interval and re-enables previously faulty blocks, which increases the capacity once again.

This policy benefits workloads that are relatively localized over discrete time intervals (small

Algorithm 2 DPCS Policy 1 (access diversity-based).

```
//Inputs: $CyclesPerInterval$ (integer, in cycles), $HighThreshold$ ($HT$: high policy threshold, arbitrary units), $LowThreshold$ ($LT$: low policy threshold, arbitrary units)
if $ClockCycleNum \bmod CyclesPerInterval == 0$ then
    Compute $IntervalBlockTouchedRate$ as fraction of blocks that were touched at least once during previous interval
    Compute $IntervalCapacityRate$ as fraction of full cache capacity that was available during previous interval
    if $IntervalBlockTouchedRate \geq HT * IntervalCapacityRate$ then
        DPCSTransition(min($CurrVDD + 1$, $SPCS\_VDD$))
    else
        if $IntervalBlockTouchedRate \leq LT * IntervalCapacityRate$ then
            DPCSTransition(max($CurrVDD - 1$, 1))
        end if
    end if
end if
```

working sets), or those workloads that access memory infrequently. In both cases, the cache capacity sacrifice can yield significant energy savings with little visible performance impact at the system level.

However, the downside of this policy is that it is oblivious to the actual impact of faulty blocks. For example, a piece of code may frequently access a region of memory that maps to only a few cache sets, which may happen to contain some faulty blocks. As a result, miss rates and average access time will dramatically increase, hampering performance. The policy is incapable of reacting to this scenario, as it might only see 5% of the available cache capacity being used for this piece of code. It will not transition to a higher VDD, even though the decreased capacity on a few select sets matters a great deal.

3.5.3.2 Dynamic Policy 2: Access Time-Based

In contrast to DPCS Policy 1, this policy (described by Alg. 3) tries to trade off energy efficiency and performance using relative average access time as the main metric. The policy is similar to that of Sec. 3.5.3.1, but it uses different performance counters to guide DPCS transitions.

The benefit of this policy is that it focuses on the "actual" cache performance as measured by average access time. Thus, the real impact of faulty blocks is captured. Revisiting the example

Algorithm 3 DPCS Policy 2 (access time-based).

```
//Inputs: $CyclesPerInterval$ (integer, in cycles), $HitLatency$ (integer, in cycles) $HighThreshold$
($HT$: high policy threshold, arbitrary units), $LowThreshold$ ($LT$: low policy threshold, arbitrary
units)
if $ClockCycleNum$ mod $CyclesPerInterval == 0$ then
    Compute $AverageAccessTime$ during previous interval
    if $AverageAccessTime \geq HT * HitLatency$ then
        DPCSTransition(min($CurrVDD + 1$, $SPCS\_VDD$))
    else
        if $AverageAccessTime \leq LT * HitLatency$ then
            DPCSTransition(max($CurrVDD - 1$, 1))
        end if
    end if
end if
```

from Sec. 3.5.3.1, this approach will scale up VDD to relieve the performance bottleneck. Furthermore, because this policy does its best to bound access time, it may be easier to estimate the impact of DPCS on system-level application performance.

However, this policy also has a significant drawback. Cache miss rates are strongly dependent on application behavior. Miss rates and average access time might increase simply due to the application referencing a previously unused data structure, causing compulsive misses. In such a case, the access time-based policy will increase the supply voltage in order to reduce the miss rate, but it would have little to no effect on performance. Thus, neither policy is superior in all cases.

3.6 Modeling and Evaluation Methodology

We assessed SPCS and DPCS using a combination of analytical and simulation-based evaluations. The overall framework is depicted in Fig. 3.3. We now describe the models and procedures used in the evaluation.

3.6.1 Modeling Cache Fault Behavior

Probabilistic failure models were used to analytically compare power/capacity tradeoffs, predict yield, and guide the generation of random fault map instances for the simulation part of our eval-

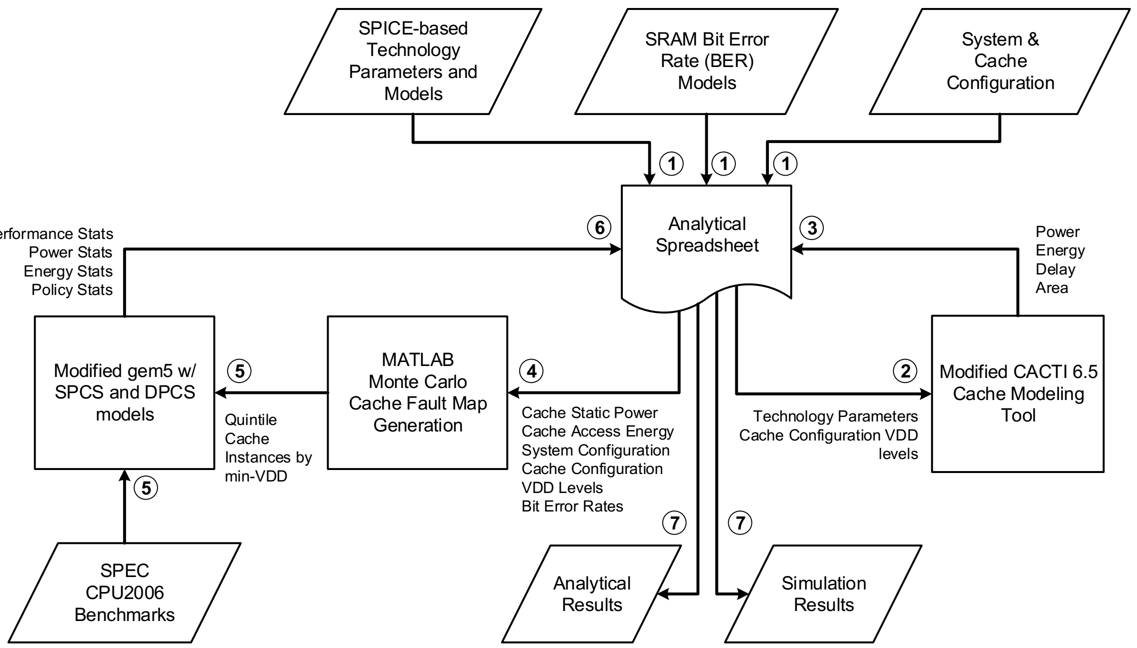

SPICE-based Technology Parameters and Models
SRAM Bit Error Rate (BER) Models
System & Cache Configuration
Analytical Spreadsheet
Performance Stats Power Stats Energy Stats Policy Stats
Power Energy Delay Area
Modified gem5 w/ SPCS and DPCS models
MATLAB Monte Carlo Cache Fault Map Generation
Modified CACTI 6.5 Cache Modeling Tool
Quintile Cache Instances by min-VDD
Cache Static Power Cache Access Energy System Configuration Cache Configuration VDD Levels Bit Error Rates
Technology Parameters Cache Configuration VDD levels
SPEC CPU2006 Benchmarks
Analytical Results
Simulation Results
1
2
3
4
5
6
7

Figure 3.3: Modeling and evaluation framework. Circled numbers indicate the order of tasks performed.

uations. Eqn. 3.4 summarizes the essential fault models, where $p_f(v)$ is the probability of cell failure (BER) at supply voltage v.

$$
\begin{aligned}
p_{fs}(v) &= 1-(1-p_f(v))^k && \text{is Pr[faulty subblock of length } k \text{ bits].}\\
p_{fb}(v) &= 1-(1-p_{fs}(v))^m && \text{is Pr[faulty block of length } m \text{ subblocks].}\\
p_{ft}(v) &= p_{fb}(v)^a && \text{is Pr[completely faulty set of associativity } a\text{].}\\
p_{yield}(v) &= (1-p_{ft}(v))^s && \text{is the yield of an } a\text{-way DPCS cache with } s \text{ sets.}
\end{aligned}
\tag{3.4}
$$

For our yield comparisons with SECDED and DECTED ECC, we assumed each method maintained ECC at the subblock level, which is significantly stronger than conventional block-level application. The number of ECC bits $n-k$ required per subblock of length k bits with total subblock codeword of length n bits, as well as ECC yield, is given by Eqn. 3.5:

Table 3.1: System and cache configurations for DPCS evaluations. Some parameters only pertain to the architectural simulations.

Parameter	Config. A	Config. B
Clock Freq.	2 GHz	3 GHz
L1$ Size, Assoc., Hit Lat.	64 KB by 4, 2 cycles	256 KB by 8, 4 cycles
L2$ Size, Assoc., Hit Lat.	2 MB by 8, 8 cycles	8 MB by 16, 16 cycles
L1 Interval (cycles)	61,200	122,400
L2 Interval (cycles)	859,200	1,838,400
L1 VoltageTransitionPenalty (cycles)	2 * No. Sets + 100	2 * No. Sets + 200
L2 VoltageTransitionPenalty (cycles)	2 * No. Sets + 400	2 * No. Sets + 2000
L1 DPCS Policy 1 Threshold Low, High (A.U.)	0.75, 0.85	0.75, 0.85
L2 DPCS Policy 1 Threshold Low, High (A.U.)	0.05, 0.15	0.05, 0.15
L1 DPCS Policy 2 Threshold Low, High (A.U.)	1.10, 1.30	1.10, 1.30
L2 DPCS Policy 2 Threshold Low, High (A.U.)	1.10, 1.30	1.10, 1.30

$$\begin{aligned} n_{\text{SECDED}} - k &= log_2(n_{\text{SECDED}}) + 1 \\ n_{\text{DECTED}} - k &= log_2(n_{\text{DECTED}}) + 2 \\ p_{\text{yield,SECDED}} &= \text{Pr[All subblocks have } \leq 1 \text{ faulty bit].} \\ p_{\text{yield,DECTED}} &= \text{Pr[All subblocks have } \leq 2 \text{ faulty bits].} \end{aligned} \tag{3.5}$$

3.6.2 System and Cache Configurations

The proposed mechanism as well as the SPCS and DPCS policies were evaluated for two system configurations: *Config. A* and *Config. B*. These are described by Table 3.1. Each system configuration had a split L1 cache and an L2 cache. SPCS and DPCS policies were only applied to the L1D and L2 caches. The same policy was used at both levels. The policies' decisions were not coordinated between cache levels.

3.6.3 Technology Parameters and Modeling Cache Architecture

To maintain relevance to the test chip experiments from Sec. 3.3, MOSFET saturation and leakage current were drawn from the corresponding industrial 45nm SOI technology library. All other parameters are based on ITRS data. Bit error rates (BER) were computed using models from Wang et al. [147] and are shown in Fig. 3.4a. Since the proposed architectural mechanism allows for SRAM access at reduced voltage, we adopted read stability for the BER calculations, which was the worst case for static noise margin (SNM). Otherwise, we did not distinguish between causes of cell failure.

We calculated delay, static power, dynamic energy per access, and area of the modified and baseline cache architectures using CACTI 6.5 [195]. CACTI generated the optimal design for the nominal VDD of 1 V using the energy-delay product as the metric. We modified CACTI to re-evaluate the energy and delay as the data array VDD was scaled down without re-optimizing the design. The L2 cache used slower and less leaky transistors than the L1 cache to keep power in check.

Our CACTI results indicated that as long as the data array VDD is above near-threshold levels, reducing the data cell VDD impacts the total cache access time by approximately 10% in the worst case of four cache configurations at min-VDD. This performance impact is modeled as a 1-cycle hit time penalty when the cache operates at the lowest DPCS voltage level in our simulations. This is based on a conservative assumption that the cache access critical path dictates overall cycle time.

3.6.4 Simulation Approach

To simulate the *baseline* (no VDD scaling and no faulty blocks), SPCS, DPCS Policy 1, and DPCS Policy 2 caches, we carefully modified the cache architecture in the gem5 [196] framework to implement SPCS and DPCS in detail as well as instrument it for cache, CPU, and DRAM power and energy. These used simple first-order energy models, with constants for static power and dynamic energy/operation. This enabled the overall energy impact of extra CPU stall cycles and DRAM accesses to be captured in addition to the direct cache energy savings from SPCS and DPCS.

Table 3.2: Common parameters used in all gem5 simulations for DPCS evaluations.

Parameter	**Value**	**Parameter**	**Value**
ISA	Alpha	Simulation Mode	Syscall Emulation
CPU Model	Detailed (OoO)	Block Replacement Policy	LRU
Number of Cores	1	Cache Configuration	L1 (Split), L2
Number of Memory Channels	1	Cache Block Size	64 B
Memory Model	DDR3-1600 x64	Physical Memory Size	2048 MB
Fast-forward	2 billion instructions	Simulate	2 billion instructions
Benchmark Compile Optimizations	Base	Input	First reference
Full VDD (Baseline)	1 V	Number of Data VDDs, Added Bits/Block.	3, 3

We used 16 SPEC CPU2006 benchmarks cross-compiled for the Alpha ISA using base-level optimization using gcc. The benchmarks were fast-forwarded for two billion instructions, then simulated in maximum detail for two billion instructions using the first SPEC reference data inputs. We modeled a single-core system to run the single-threaded benchmarks. The gem5 settings common to all system configurations are summarized in Table 3.2.

For each of the non-baseline policies, 16 SPEC benchmarks, and 2 system configurations, gem5 was run five times for five unique fault map inputs. Thus, in total, we performed 512 gem5 simulations to provide some insight on the impact of manifested process variations.

3.6.4.1 Fault Map Generation and Selection of Runtime VDDs

For accurate simulation in the presence of faulty blocks at low voltage, probabilistic power/capacity scaling data based on CACTI results were input to a MATLAB model that generated 10,000 random fault map instances for each cache configuration. A script chose five fault maps corresponding to each quintile in the distribution as sorted by the min-VDD of each cache instance. For each of these five selected fault map instances, three run-time VDDs were selected based on the criteria described earlier in Sec. 3.5.1.6. The total static power of each cache instance was then computed based on the number of power-gated faulty blocks. The particular locations of faulty blocks were assumed to not matter, because eventually all blocks will be referenced roughly uniformly [197].

The SPCS voltage for each fault map was minimized such that the capacity was at least 99%. The lower DPCS voltage was set such that capacity was at least 75% and no set was totally faulty (recall that we require each set to have at least one non-faulty block at all voltages). In practice, out of 40,000 total generated fault map instances, the lower DPCS voltage was always limited by the latter constraint.

3.6.4.2 Parameter Selection for DPCS Policies

The DPCS policy parameters were set to reasonable values to reduce the huge design space, based on our analysis of policy behaviors on different workloads (examples are given later in

Sec. 3.8.2.3). We assumed that the `VoltageRegulatorDelay` to scale the data array VDD is enough to minimize noise-induced SRAM errors and allow for a stable transition. For each cache configuration, the `Interval` parameter was set to be roughly `VoltageTransitionPenalty` $\times 100$ in clock cycles. This is done so that the worst-case system-level performance impact of DPCS transitions would be bounded to 1% of overall run-time. Our choices of `Interval` are not guaranteed to be optimal across all applications, but they do provide some guarantees on the high-level performance impact of DPCS.

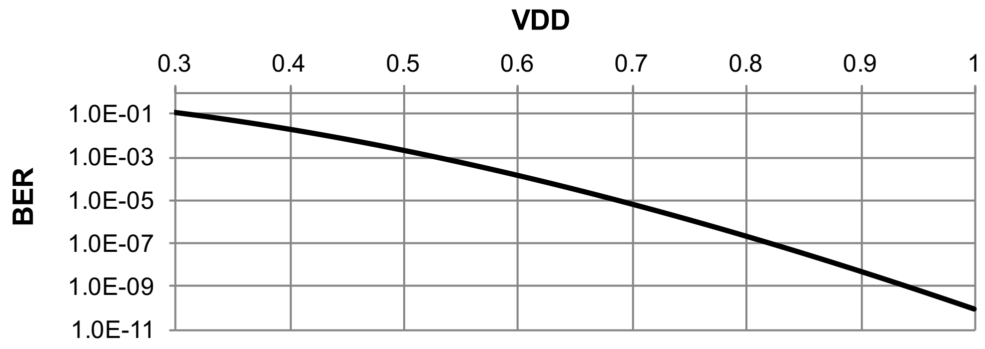
DPCS `HighThreshold` (`HT`) and `LowThreshold` (`LT`) for both policies were set to provide sufficient hysteresis, avoiding voltage level oscillation for workloads in general over different phases of execution. For DPCS Policy 1 (access diversity-based), the thresholds were set such that mild cache utilization (as measured by the working set size over an `Interval` [198]) would result in a voltage reduction. For DPCS Policy 2 (access time-based), we ensured that the cache average access time would not suffer too greatly in relation to its hit time.

The choice of policy parameters might be chosen in other ways, e.g., by leveraging other techniques for detecting phases of execution [199, 200]. The DPCS policy parameters could also be dynamically adjusted by online monitoring of workload behaviors and/or OS-based management. We leave these possibilities to future work, where we aim to adapt DPCS to the context of multicore systems.

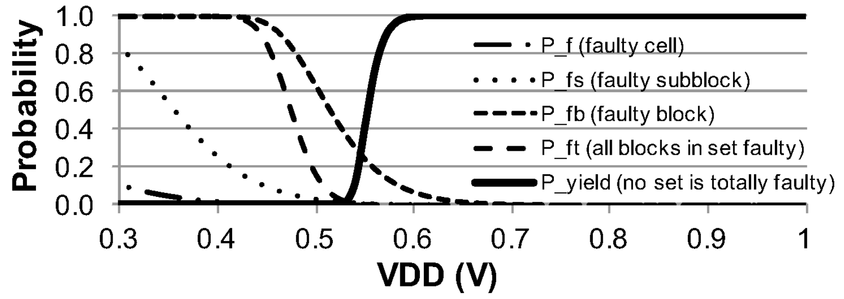
3.7 Analytical Results

In this section, results pertaining to yield, static power vs. effective capacity, and area overheads are discussed without the use of architectural simulation.

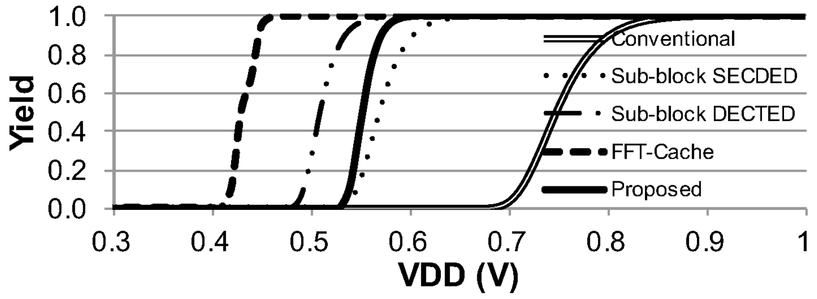
3.7.1 Fault Probabilities and Yield



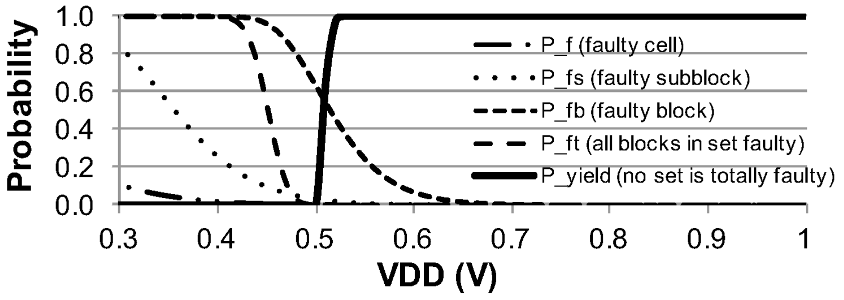
(a) SRAM bit error rate (BER)



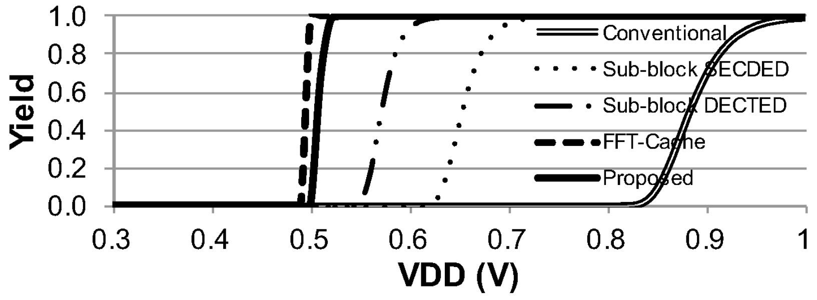
(b) L1 Config. A fault probabilities



(d) L1 Config. A yield comparison



(c) L2 Config. B fault probabilities



(e) L2 Config. B yield comparison

Figure 3.4: Analytical failure characteristics for each configuration.

Using the derived static power from our technology data integrated into CACTI as well as the analytical fault models from Sec. 3.6.1, we were able to compare our proposed mechanism analytically with alternative approaches. The results for L1 Config. A and L2 Config. B are shown in Fig. 3.4. Similar trends are observed for the L1 Config. B and L2 Config. A caches, whose results are not shown for brevity. For yield vs. design-time fixed VDD, we compare with SECDED/DECTED ECC and FFT-Cache [149], which is a recent FTVS approach achieving one of the lowest min-VDDs. For FFT-Cache and both ECC schemes, fault tolerance is applied at the subblock level of eight bytes (where the full block size is 64 bytes) for additional resilience. For the static power vs. effective capacity metric described in Sec. 3.4, we compare with FFT-Cache and a generic way-granularity (associativity reduction) power gating scheme.

The probability of faulty bits (BER) is shown in Fig. 3.4a, while the probabilities of faulty subblocks, blocks, sets, and DPCS yield are depicted in Figs. 3.4b and 3.4c. It is clear that in the region of 500 to 600 mV, a steep dropoff in yield is encountered. This is due to exponential fault rates at the bit level which becomes critical in this region. This trend agrees with our test chip measurements in Sec. 3.3 which showed that SRAM failures experience “avalanche-like” behavior.

In Figs. 3.4d and 3.4e, the yield of DPCS is compared with alternative approaches. Although our approach does not achieve the best min-VDD for constant yield, it did better than SECDED in all cache configurations, even though ECC is applied at 8-byte subblock granularity. In the L1 *Config. A* configuration, DECTED achieves moderately better min-VDD than our mechanism due to low associativity, which impacts yield of our approach. The 16-way set associativity of L2 *Config. B* results in lower min-VDD than both ECC schemes and nearly matches FFT-Cache. Note that DECTED typically incurs high area, power, and performance overheads to achieve the yields shown [157], [160]. Furthermore, SECDED/DECTED may be overkill for sparse voltage-induced faults, and as voltage is reduced, tolerating bit cell failures reduces the ability of these ECC schemes to tolerate soft errors. Nevertheless, these ECC schemes could be combined with our approach to handle both voltage-induced faults as well as transient soft errors. We leave this to future work.

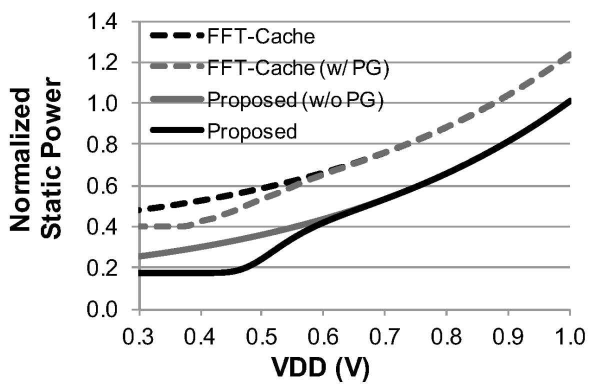
3.7.2 Static Power vs. Effective Capacity

Despite some yield weaknesses for low-associativity cache configurations, the proposed mechanism still achieves lower total static power at all voltages compared to FFT-Cache [149]. This is due to the lower overheads of DPCS' fault tolerance compared to the complex approach of FFT-Cache. The difference arises from a significantly smaller fault map with only three extra bits per 64-byte block. In contrast, FFT-Cache needs two entire fault maps for each of the lower VDDs, compounding its existing high overheads and extra necessary logic on the critical path.

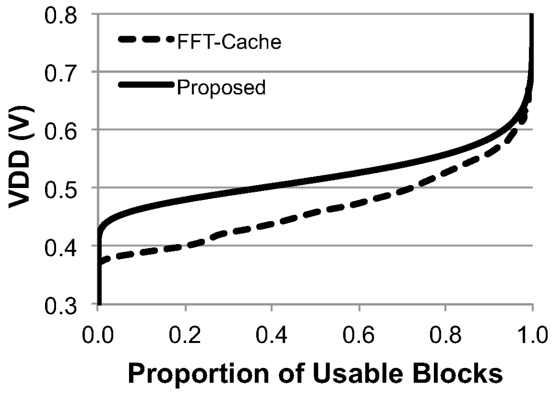
Furthermore, under certain floorplan and layout conditions described earlier in Sec. 3.5.1.2, our approach can power gate blocks as they become faulty at low voltage, providing additional power reduction at low capacities compared to pure voltage-scaling. This result is depicted in Fig. 3.5a along with a hypothetical version of FFT-Cache with power gating of lost capacity (gray dashed line). If the power gating circuitry described in Sec. 3.5.1.2 is omitted, then power savings will be moderately worse at capacities less than 100%, as shown by Fig. 3.5a. However, overall area overheads would decrease by roughly 2%, as described below in Sec. 3.7.3. This simpler architecture may be better for SPCS, where power gating logic is only used for up to 1% of blocks that are allowed to be faulty. While the per-block power gating feature is useful for an improved power vs. capacity curve, it is not absolutely essential to DPCS. This makes our architecture compelling even for traditional thin-cell SRAM layouts with almost no modifications to the macro floorplan.

Although our approach achieves lower power than FFT-Cache, it also has a lower effective capacity at all voltage levels than FFT-Cache due to a much simpler but weaker fault tolerance mechanism, as shown by Fig. 3.5b. This reaffirms the better yield of FFT-Cache. These results clearly illustrate a tradeoff in power and capacity at each voltage level between the two approaches.

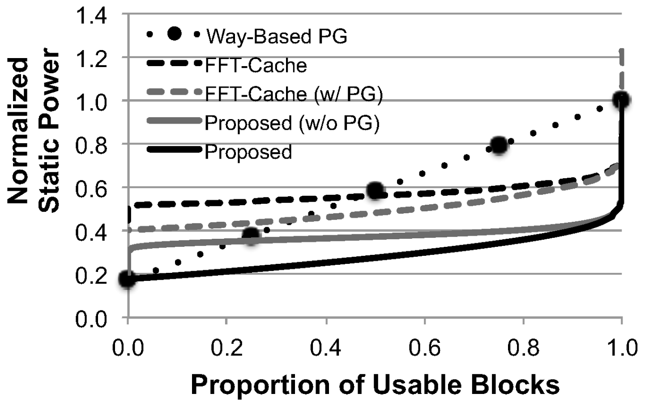
In the end, the superior power savings of the proposed approach make up for the capacity deficit with respect to FFT-Cache, as depicted by Fig. 3.5c. Moreover, our approach with power gating of faulty blocks does better than generic associativity reduction via per-way power gating at all cache capacities. This is unlike FFT-Cache, which does worse than power gating below 50% cache capacity because of its high overheads. We found that for the L1 Config. A cache, our mechanism



(a) Static power vs. VDD



(b) VDD vs. effective capacity



(c) Static power vs. effective capacity comparison

Figure 3.5: Analytical power/capacity results for the L1 Config. A cache with and without power gating of faulty blocks. Similar normalized trends were observed for the other three cache configurations.

achieves 31.1% lower static power than FFT-Cache at the same 99% effective capacity. In addition, as cache capacity is scaled down with power, the gap increases in favor of our approach.

3.7.3 Area Overhead

Our CACTI results indicate that from the fault map alone, our area overheads compared to a baseline cache lacking fault tolerance do not exceed 4% in the worst case of all configurations. The additional area overheads from the power gating transistor [151] plus small inverter are estimated to be no more than 2%. The DPCS policy implementation is assumed to have negligible area overhead, due to its simplicity and that it could be implemented in software, as the cache controller typically includes the necessary performance counters. Furthermore, the fault map comparison logic is only a few gates per cache way. Thus, we estimated the total area overhead to be up to 6% among all tested cache configurations. These area overheads are a significant improvement compared to the reported overheads of other FTVS schemes such as 10T SRAM cells (66%), ZerehCache (16%), Wilkerson et al. (15%), Ansari et al. (14%), and FFT-Cache (13%) [149].

3.8 Simulation Results

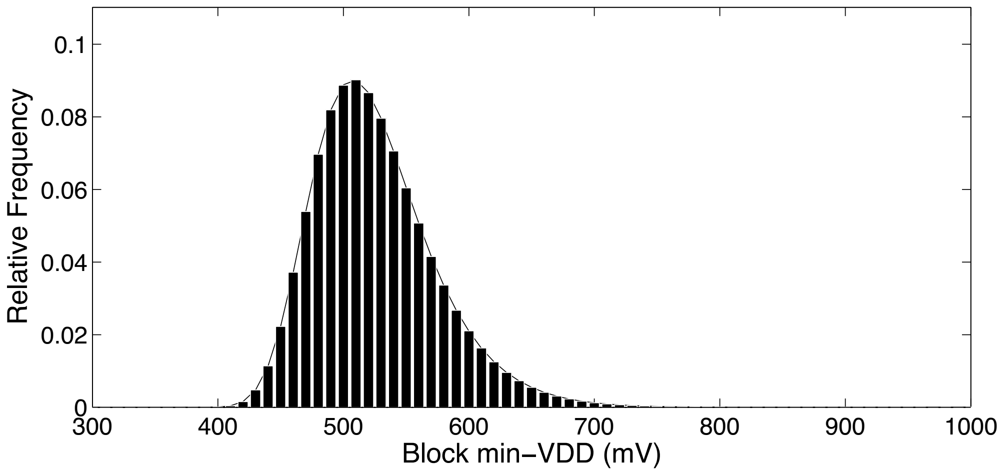
In this section, we discuss aggregate energy savings and performance overheads across a portion of the SPEC CPU2006 benchmark suite on both system configurations for several different fault map instances.

3.8.1 Fault Map Distributions

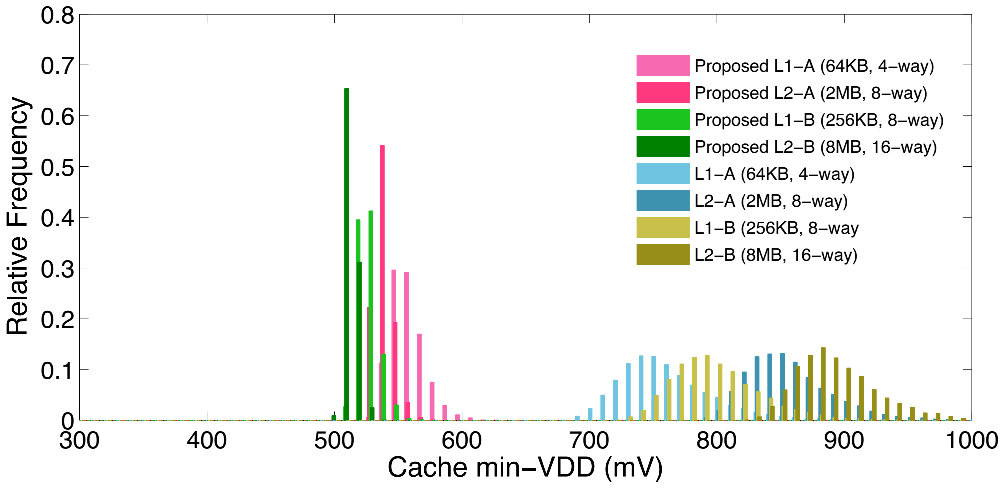
Using the fault models described in Sec. 3.6.1 and the methodology from Sec. 3.6.4.1, we generated 10,000 complete fault map instances for each of the four cache configurations, i.e., *L1-A*, *L2-A*, *L1-B*, and *L2-B*. Each fault map contained the min-VDD for each cache block at a resolution of 10 mV. Fig. 3.6a depicts the aggregate faulty block distribution across all 40,000 randomly generated fault maps.

Fig. 3.6b depicts the histogram of the minimum global VDD that can be used across a cache instance. For the baseline case, any faulty block at a given voltage will limit the overall min-VDD. Clearly, the long tail from Fig. 3.6a hampers the traditional design approach (these numbers agree with our 45nm process guidelines for min-VDD as well). In the baseline caches, as nominal cache capacity increases, so does the min-VDD, regardless of associativity.

However, the cache min-VDD distributions for the proposed architecture follow a very different trend, as indicated by Fig. 3.6b. Unlike traditional caches, the benefits of increased associativity outweigh increased nominal capacity for reducing VDD. This is because the voltage scalability of our approach is only limited by the constraint that no set may be totally faulty (recall that the min-VDD requirement for our approach from Sec. 3.5.1.6 and Sec. 3.6.1). As associativity increases,



(a) Block granularity



(b) Cache granularity

Figure 3.6: Manifested distribution of min-VDD at the block and cache granularities.

the likelihood of any set being completely faulty rapidly decreases.

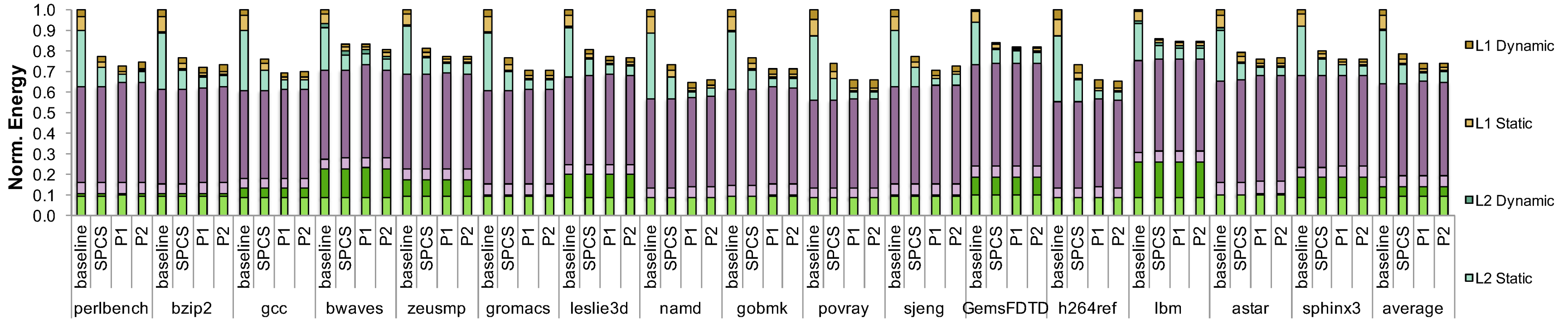
Thus, our simple fault tolerance scheme exhibits good scalability as cache memories increase in size and associativity. Moreover, the variance in min-VDD across many cache instances is significantly less than conventionally designed caches, meaning that even design-time choice of VDD can be done with narrower guardbands while maintaining high yield.

3.8.2 Architectural Simulation

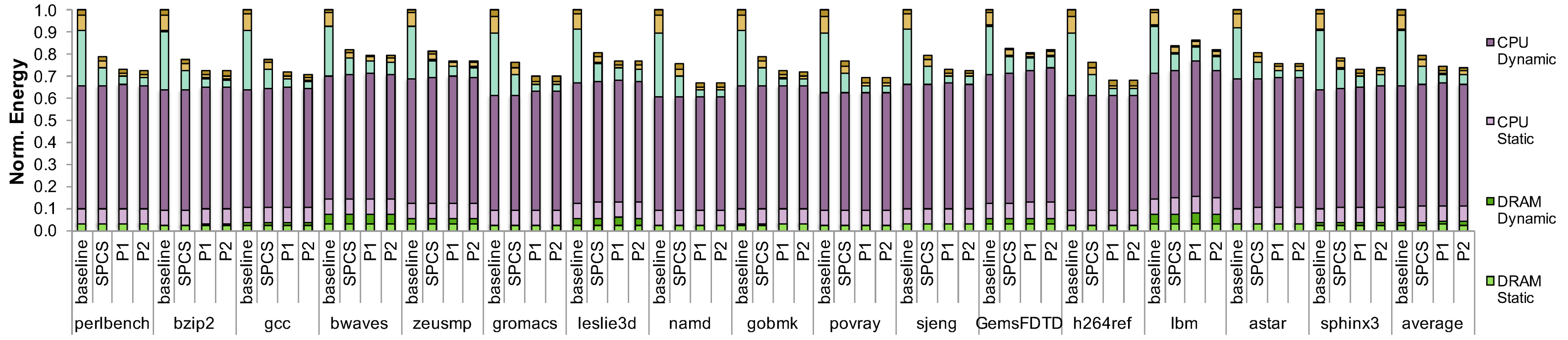
We now discuss the impact of SPCS and DPCS on energy and performance. All the depicted results include the architectural design choice to power gate faulty blocks. We also conducted the same set of experiments without per-block power gating; the results are not shown for brevity. For the worst-case fault maps at the lowest voltages, the total cache static energy difference between the two approaches did not exceed 5%. At the system level, the overall energy gap between the two design choices was less than 1%. Henceforth, we only discuss the architecture with the power gating feature included.

3.8.2.1 Breakdown of Energy Savings

Both SPCS and DPCS deliver static and dynamic power savings due to low voltage operation even during cache accesses. The breakdown of total system energy is depicted in Fig. 3.7 for the *baseline*, *SPCS*, *DPCS Policy 1* (*P1*), and *DPCS Policy 2* (*P2*) caches using the mean of five unique faultmap runs. The variations in energy breakdowns across benchmarks in total baseline and SPCS energy are due to workload dependences, i.e., the extent to which they are CPU or memory-bound.



(a) Config. A, CPU power was normalized to 50% and DRAM energy to 15% of total average baseline energy



(b) Config. B, CPU dynamic power was scaled up by 3x, and static power by 5x compared to Config. A in accordance with 50% higher clock frequency. DRAM energy was unchanged

Figure 3.7: Breakdown of averaged total system energy for *baseline*, *SPCS*, and DPCS policies *P1* and *P2*, all normalized to the respective benchmarks' baseline system total energy. Note that because DPCS has little impact on overall runtime and DRAM accesses, the relative energy breakdown across CPU and DRAM can be re-normalized to other systems in a straightforward manner.

SPCS achieved good and consistent energy savings across all benchmarks, achieving an average of 62% total cache and 22% total system energy savings with respect to the baseline in both Configs. A and B. This is because cache efficacy is not significantly impacted at the 99% capacity point, and no voltage transitions occur to other power/capacity points.

Energy savings were almost always better for both DPCS policies P1 and P2 compared to SPCS and the baseline. This was in line with our expectations, as both policies never go above the SPCS voltage level because little performance would be gained. However, note that DPCS has some system-level energy overhead due to extra time spent in DPCS transitions and potentially more cache misses and CPU stall cycles compared to SPCS or baseline. For example, reduced power and capacity in L1 due to DPCS can cause more L2 accesses, resulting in a tradeoff between L1 static power and L2 dynamic energy.

In nearly all cases, the access diversity-based DPCS Policy 1 (P1 in Fig. 3.7) achieved equal or slightly lower total cache energy than the access time-based DPCS Policy 2 (P2). On average, P1 reduced total cache (system) energy by 78% (27%) compared to baseline on Config. A, and by 80% (26%), on Config. B. On average across all benchmarks and configurations, P2 saved nearly identical system-level energy as P1. However, in some benchmarks, such as `perlbench` on Config. A or `lbm` on Config. B, the gap between policies was noticeable.

Discrepancies in energy savings between DPCS Policy 1 and DPCS Policy 2 could be explained by a combination of threshold parameters and the method of inferring performance impact at low voltage. For example, the cache capacity utilization measured by DPCS Policy 1 loosely relates to performance through working set analysis. In contrast, DPCS Policy 2 measures performance more directly through average access time. However, owing to the huge design space already presented, we do not have the resources to exhaustively test other policy threshold parameters. We leave static and dynamic policy optimization to future work.

Regardless of differences in policies, SPCS and DPCS demonstrate reduced total cache and system energy overall with little impact on the CPU or DRAM. As we will see in the next section, this can be attributed to low performance overheads while still reducing static cache power and some dynamic access energy.

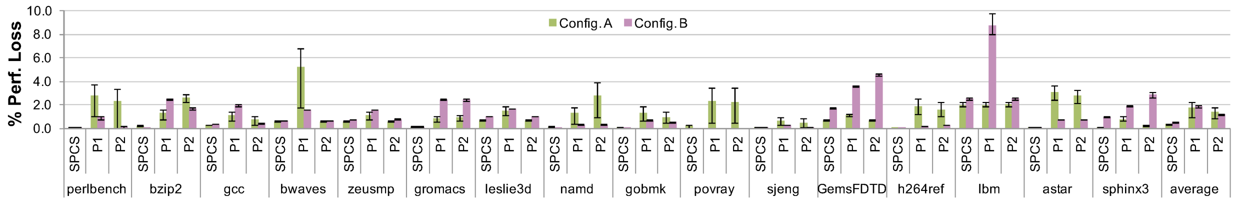


Figure 3.8: Average performance degradation of SPCS and DPCS policies P1 and P2. Error bars indicate the range of performance impact across the five quintile fault maps.

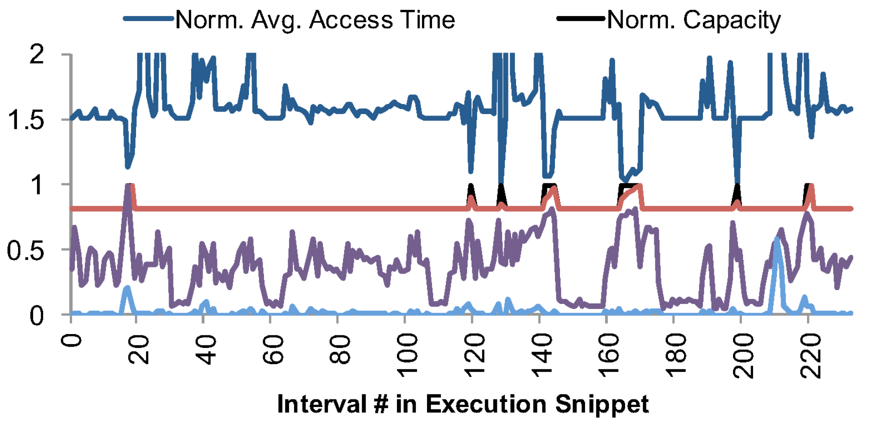
3.8.2.2 Performance Overheads

The energy savings presented above come at a mild performance cost as measured by execution time. Fig. 3.8 depicts the average performance penalty for SPCS, DPCS Policy 1, and DPCS Policy 2 for each benchmark with respect to the baseline system.

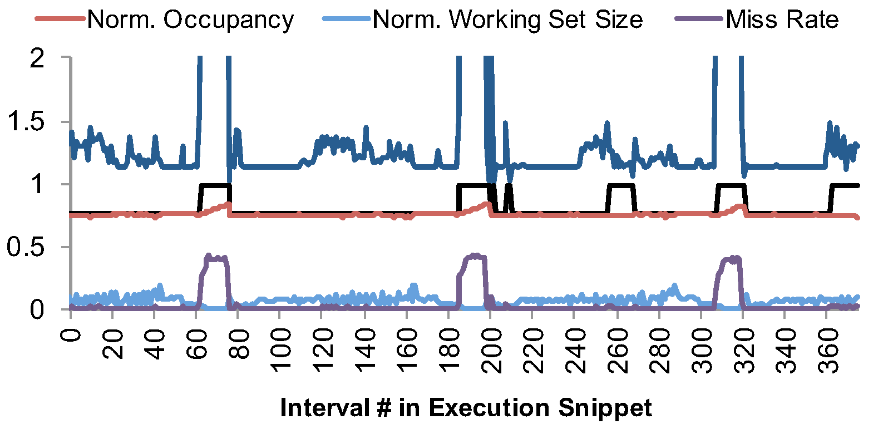
Across all benchmarks on Config. A, the slowdown of SPCS is only $0.32 \pm 0.06\%$ with respect to baseline. For Config. B, the slowdown is $0.50 \pm 0.02\%$. This shows that a 1% loss of cache capacity has a negligible impact on performance. Thus, even with the worst case fault maps, the per-chip SPCS approach is effective at achieving energy savings with very little overhead. Thanks to the low variance in min-VDD with SPCS (see Sec. 3.8.1), an aggressive design-time choice of SRAM VDD (as opposed to our run-time approach) could also result in negligible performance impact while maintaining high yield and good energy savings.

Fig. 3.8 also shows how DPCS performance degrades compared to SPCS. On average, the access time-based DPCS Policy 2 (0.77% to 1.76%) does equal or better than the access diversity-based DPCS Policy 1 (0.94% to 2.24%) on Config. A. This small performance advantage of P2 holds for Config. B. For both SPCS and DPCS, higher cache associativity improves performance consistency in addition to min-VDD across a variety of fault maps.

Interestingly, the per-benchmark performance of both policies appears to be dependent on the system configuration. For example, in Config. A, P1 does poorly on `bwaves` and is also very sensitive to fault map variation. In contrast, `lbm` performed very consistently for SPCS, P1, and P2 under Config. A, but suffered relatively more under P1 for Config. B. We believe these sensitivities are due to a combination of different CPU frequencies, cache hit times, and possibly fault map instances.



(a) gobmk, DPCS Policy 1, L1 Config. A. Each Interval corresponds to 61,200 cycles at 2 GHz



(b) bzip2, DPCS Policy 2, L2 Config. B. Each Interval corresponds to 1,838,400 cycles at 3 GHz

Figure 3.9: Selected snippets of DPCS traces in mid-execution using median fault map instances.

3.8.2.3 DPCS Policy Behavior

Finally, we illustrate some characteristics of each DPCS policy by examining short snippets of their behavior during execution. Fig. 3.9 shows two trace fragments for different workloads, system configurations, cache levels, and DPCS policies.

For most of the execution trace depicted in Fig. 3.9a, the cache capacity is roughly 80%. This indicates that DPCS is in the lowest voltage mode. DPCS Policy 1 does not boost voltage unless the working set size exceeds 90% of the available capacity. This occurs several times around

`Interval` 150. The cache occupancy then increases rapidly, indicating that some performance was probably recovered. However, higher miss rates and average access times do not necessarily cause high block touch rates. This can be seen around `Interval` 210, where the miss rate jumps to approximately 60%, but only 65% of the nominal capacity is referenced at that time. In this case, a DPCS boost would probably not help performance noticeably.

A very different trend is shown in Fig. 3.9b. Since the cache is relatively large compared to the needs of the application, the working set size always remains below 20% of nominal capacity. However, note that the application regularly has a miss rate of up to 40%. DPCS Policy 2 adapts by transitioning to a higher cache capacity in an attempt to recover performance. However, the downside is that such an action is not guaranteed to affect the miss rate or recover performance.

3.9 Conclusion

In this work, we proposed static (SPCS) and dynamic (DPCS) power/capacity scaling, a novel fault-tolerant, voltage-scalable (FTVS) SRAM architecture for energy-efficient operation. Our proposed mechanism and policies leverage several important observations. First, using our 45nm SOI test chips, we observed the *fault inclusion property*, which states that any block that fails at some supply voltage will also be faulty at all lower voltages. To the best of our knowledge, no other FTVS approaches exploit this behavior. This allows for efficient fault map representation for multiple run-time cache VDD levels. Our approach also has the benefit of a simple and low-overhead implementation, which allows the caches to achieve better power/capacity tradeoffs than some competing approaches that primarily focus on achieving low min-VDD at fixed yield.

We believe that DPCS has the potential to complement conventional dynamic voltage/frequency scaling (DVFS) for improved system-level energy proportionality. The application of DPCS in a real system would bring interesting challenges and opportunities to the software stack. Our future work seeks to develop coordinated DVFS and DPCS policies at the system level, taking into account heterogeneous system architectures, multi-core processors, cache coherence, aging effects, and implications for power management of other system components. This approach to variation-aware design could also be extended to allow for approximate computing.

CHAPTER 4

X-Mem: A New Extensible Memory Characterization Tool used for Case Studies on Memory Performance Variability

Effective use of the memory hierarchy is crucial to cloud computing. Platform memory subsystems must be carefully provisioned and configured to minimize overall cost and energy for cloud providers. For cloud subscribers, the diversity of available platforms complicates comparisons and the optimization of performance. To address these needs, we present *X-Mem*, a new eXtensible open-source software tool that characterizes the Memory hierarchy for cloud computing.

X-Mem is designed to be modular, portable, and extensible while surpassing most capabilities of existing utilities. The tool directly measures a number of statistics for throughput, (un)loaded latency, and power for each level of cache and DRAM through flexible stimuli. Its features include multi-threading, awareness of non-uniform memory architecture, and support for different page sizes. X-Mem can exercise memory using many combinations of load/store width, access pattern, and working set size per thread. The accessibility and extensibility of our tool also facilitates future research. It is written in C++ and currently runs on GNU/Linux and Windows for both x86-64 and ARM. x86-64 is also supported for Mac OS.

We demonstrate the utility of X-Mem through a series of experimental case studies using state-of-the-art platforms. The tool is used to: *(i)* infer details of cache organization, main memory performance, and DRAM power, *(ii)* compare general performance aspects of the memory hierarchy across a variety of real and virtual platforms spanning different cloud providers, instruction sets, and operating systems, and *(iii)* explore the efficacy of tuning certain DRAM parameters for performance *with a focus on determining whether DRAM variability can and should be exploited for application performance*.

Our results show how cloud subscribers could choose a preferred target platform and better optimize their applications even if the hardware/software stack is opaque. Cloud providers could use X-Mem to fine-tune system configurations and to verify machine performance envelopes before deployment. We envision novel ways that researchers could extend X-Mem for purposes such as the characterization of emerging memory architectures.

Collaborators:

- Dr. Mohammed Shoaib, Microsoft Research
- Dr. Sriram Govindan, Microsoft
- Dr. Bikash Sharma, Microsoft
- Prof. Puneet Gupta, UCLA

Source code and data are available at:

- `https://nanocad-lab.github.io/X-Mem/`
- `https://github.com/Microsoft/X-Mem`
- `https://github.com/nanocad-lab/X-Mem`
- `https://github.com/nanocad-lab?&q=xmem`
- `http://nanocad.ee.ucla.edu`

4.1 Introduction

By 2016, over 80% of enterprises are expected to adopt cloud computing [201, 202] because of its economic advantages. *Cloud providers* seek to minimize capital (*CapEx*) and operational (*OpEx*) expenses while satisfying a service-level agreement. *Cloud subscribers* want to extract maximum performance from their resources. These complementary objectives influence the entire hardware/-software stack.

The needs of cloud providers and subscribers particularly pressure the memory subsystem. From the provider's perspective, memory procurement costs dominate CapEx, with 128 GiB of DRAM costing as much as $2000 per server [203]. In OpEx, up to 30 percent of total server power is consumed by memory [39, 40, 204]. To reduce these costs, providers must be able to thoroughly explore the memory design and configuration envelope for each class of applications.

For subscribers, application performance is dependent on the properties of the memory hierarchy, from CPU caches to DRAM [205]. With a wide range of cloud platforms, subscribers need to quickly and easily evaluate general performance aspects of the memory subsystem. They also may require more detailed attributes for a chosen platform – whose designs are increasingly heterogeneous with typically opaque implementations – to optimize their application code. Moreover, making the memory hierarchy more transparent is part of the long-term solution [19] to the well-known "memory wall" problem that causes system performance to scale poorly.

Thus, careful characterization of the memory hierarchy is crucial for both the cloud provider and the subscriber to maximize the performance/cost ratio. *However, existing memory characterization tools fail to meet the following four functional requirements driven by cloud platforms.*

(A) Access pattern diversity. Cloud applications span many domains. They express a broad spectrum of computational behaviors, and access memory in a mix of structured and random patterns. These patterns exhibit a variety of read-write ratios, spatio-temporal localities, and working-set sizes. Replication of these memory access patterns using controlled micro-benchmarks facilitates the study of their performance. This can be used by cloud providers to create cost-effective hardware configurations for different classes of applications, and by subscribers to optimize their applications.

(B) Platform variability. Cloud servers are built from a mix of instruction set architectures (ISAs, e.g., x86-64 [206, 207] and ARM [208, 209]), machine organizations (e.g., memory model and cache configuration), and technology standards (e.g., DDR, PCIe, NVMe, etc.). They also include unique hardware capabilities, such as extended ISAs that feature vectorized loads and stores. Platforms also span a variety of software stacks and operating systems (OSes, e.g., Linux and Windows [206, 207]). The interfaces and semantics of OS-level memory management features such as large pages and non-uniform memory access (NUMA) also vary. In order to objectively cross-evaluate competing platforms and help optimize an application for a particular platform, a memory characterization tool should support as many permutations of these features as possible.

(C) Metric flexibility. Both the subscriber's application-defined performance and the provider's costs depend on memory performance and power. These can be described using statistical distributions of several different metrics. For example, the distribution of DRAM loaded latency might be correlated with the distribution of search query latency in a heavily loaded server. Meanwhile, both the peak and average main memory power consumption are important metrics to the cloud provider, as they impact both CapEx and OpEx respectively. Memory power could also impact application performance indirectly due to a system-level power cap [210]. However, most characterization tools do not expose these flexible statistics or integrate memory power measurement.

(D) Tool extensibility. Cloud platforms have changed considerably over the last decade and will continue to evolve in the future. Emerging non-volatile memories (NVMs), such as phase-change memory (PCM), spin-transfer torque RAM (STT-RAM), and resistive RAM (RRAM) [211, 212] introduce new capabilities and challenges that will require special consideration. The metrics of interest may also change with future applications and cloud management techniques. Unfortunately, most existing characterization tools are not easily extensible, hampering their usefulness in these scenarios.

To address these four key requirements for both providers and subscribers of cloud services, we present X-Mem: an open-source, cross-platform, and eXtensible Memory characterization software tool written in C++. Through a flexible benchmarking framework with multiple degrees of freedom, it can reveal non-trivial relationships between application behavior and attributes of the underlying system. This can provide crucial inputs to optimize cost and performance for both cloud providers and subscribers. X-Mem thus helps providers to intelligently provision cloud servers (e.g., what memory type to buy – when is lower-latency DRAM worth the opportunity cost of higher-bandwidth memory?) and operate them (e.g., what are the optimal DRAM run-time settings for different latency or throughput-sensitive applications?). It also allows subscribers to determine the optimal cloud provider and virtual machine (VM) combination for their workloads. X-Mem can help guide development of future provider server platforms by exposing many relevant aspects of the complex memory hierarchy.

We demonstrate X-Mem's utility by experimentally evaluating several aspects of seven different hardware/software platforms. They span x86-64/ARM, Windows/Linux, (micro)servers/desktop, and physical/virtual machines. We show how to uncover organizational details of the cache hierarchy, discuss the joint impact of NUMA and page size on performance for different OSes, quantify the energy proportionality of server DRAM, and contrast memory hierarchy performance across platforms. We also evaluate the effects of different DDR3 DRAM configurations on memory throughput and (un)loaded latency.

Crucially for research on Opportunistic Memory Systems, X-Mem can be used to explore the performance and power benefits of variation-aware DRAM. In contrast to recent literature [213], we use X-Mem to find that DRAM timing optimizations have a significant benefit only when the memory is lightly loaded. This is an important result for those that wish to exploit variability in DRAM latency for improved performance per dollar by tuning DRAM timings for different cloud applications.

This chapter includes the following contributions:

- A description of the design philosophy and implementation details of X-Mem, that promotes the understanding of its functionality and facilitates rapid modifications by the research community.
- Case Study 1: A characterization of memory hierarchy performance, that demonstrates how cloud subscribers can optimize applications for the memory organization of a particular platform.

- Case Study 2: A comparison of memory hierarchies across seven different platforms, that shows how cloud subscribers can select the appropriate platform for their application.

- Case Study 3: An evaluation of system configurations on main memory performance, that helps cloud providers provision and fine-tune their systems for different applications.

X-Mem source code, binaries, user manuals, programmer documentation, selected datasets, and various scripts are available online. The tool is being actively maintained and extended for ongoing research needs; the contents of this chapter are based on "v2.2.3" of the tool.[1]

This chapter is organized as follows: in Sec. 4.2, we present related work and explain how our work advances the state-of-the-art. It is followed by a description of the design and implementation of X-Mem in Sec. 4.4. In Sec. 4.5, we describe our experimental setup, describe the hardware/software platforms of interest, and validate X-Mem. We use the tool to evaluate several systems of interest in three case studies in Sec. 4.6. Sec. 4.7 concludes the chapter.

4.2 Related Work

In this section, we summarize the pertinent literature on characterization and optimization of cloud platforms and memory systems. We then review current memory benchmarking tools and highlight how X-Mem extends the state-of-the-art.

Several studies have evaluated the performance of cloud-hosted applications [214–216]. Ferdman et al. [217] and Kozyrakis et al. [218] derived infrastructure-level insights by analyzing cloud-scale workloads. CloudCmp [219] contrasted the performance and cost of different providers' platforms. Blem et al. [220] explored differences in CPU energy and performance as a function of the instruction set itself. However, none of these cloud studies focused on memory.

With a broad scope that includes the cloud, there have been many studies that optimize memory systems. Memory-aware policies for dynamic voltage/frequency scaling (DVFS) of CPUs have been suggested [221, 222]. Many techniques for improving DRAM energy efficiency via schedul-

[1]Recently, we have added preliminary support for Intel Xeon Phi many-core co-processors, as well as Mac OS on x86-64, but neither are considered or evaluated further in this chapter.

ing and alternative hardware organizations [112, 223–225] have been explored. After Barroso and Hölzle described the problem of memory energy proportionality [39, 40], other researchers recommended using DVFS for the memory bus as a solution [182–184, 226, 227]. Recently, researchers have taken a different angle, studying how to improve memory and cache energy by opportunistic exploitation of hardware manufacturing variations [1, 2, 8].

With regard to hardware variability, two studies are of particular interest. Chandrasekar et al. [228] described a novel procedure to optimize DDR3 timing parameters for DRAM modules. However, they did not discuss the application-level benefits of their approach. Adaptive-Latency DRAM (AL-DRAM) [213] explored this idea further, evaluating it using a suite of benchmarks on a real system. However, the authors did not study the impact of variation-aware tuning on the memory performance itself. Without this low-level insight, it is unclear *why* applications benefit. We briefly revisit this question at the end of our third case study; our observations appear to contradict the conclusions in AL-DRAM [213].

Table 4.1: High-level feature comparison of X-Mem with other memory benchmarking tools. ❍ indicates partial feature support. No existing tool provides the same breadth of capability as X-Mem.

Tool	Thru-put	Lat.	Loaded Lat.	Multi-Thrd.	NUMA	Lrg. Pages	Power	Cache & Mem.	Native Linux	Native Win.	x86	x86-64	ARM	Vector Inst.	Open Src.	Lang.	**(A)** Acc. Patt. Divers.	**(B)** Platf. Var.	**(C)** Metric Flex.	**(D)** Tool Extens.
STREAM v5.10 [229]	✓			❍				❍	✓		✓	❍	❍		✓	C, FORTRAN				
STREAM2 v0.1 [230]	✓			❍				❍	✓		✓	❍	❍		✓	FORTRAN				
lmbench3 [231]	✓	✓		✓				❍	✓		✓	❍	❍		✓	C				
TinyMemBench v0.3.9 [232]	✓	✓				✓		✓	✓		✓	❍	❍	✓	✓	C				
mlc v2.3 [233]	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓								
X-Mem v2.2.3 [234, 235]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	C++	✓	✓	✓	✓

Several existing micro-benchmark suites are available to quantify memory system performance. They include STREAM [229,236,237], STREAM2 [230], lmbench3 [231], TinyMemBench [232], and mlc [233]. We surveyed their capabilities that relate to the cloud-specific needs described earlier in Sec. 4.1, namely: *(A) access pattern diversity*; *(B) platform variability*; *(C) metric flexibility*; and *(D) tool extensibility*. A high-level comparison of these tools' features is shown in Table 4.1. Intel's Memory Latency Checker (mlc) [233] is the closest tool to X-Mem in terms of feature support, but like the others, does not address these four important requirements.

Some relevant studies used custom micro-benchmarks to study certain aspects of cache and memory performance [238–241]. Murphy et al. [242] proposed idealized analytical models to examine application sensitivity to memory bandwidth and latency in order to better address system bottlenecks. However, none of these works have described or released a tool suitable for use by others.

X-Mem generally surpasses the capabilities of prior tools while being usable by the broader community. To the best of our knowledge, no previous work has created or used a memory characterization tool with the same breadth of features as X-Mem. The studies conducted in this chapter would have been cumbersome if not impossible to achieve without a tool like X-Mem. With the versatility, portability, extensibility, and simple UI provided by X-Mem, it is easy to automate the collection and post-processing of hundreds of thousands of different memory hierarchy measurements on machines deployed in the field. As we will demonstrate in the two case studies later on in this chapter, this flexibility grants researchers and engineers considerable visibility into architectural subtleties and can also reveal opportunities for cloud server provisioning, run-time management, and application code optimization. We believe that our tool will make characterization and evaluation studies easier to conduct in the future, while opening new avenues for exploration through its versatility, portability, and extensibility.

4.3 Background

In this section, we describe the basics of a typical memory hierarchy in a modern non-uniform memory access (NUMA) server as well as double data rate (DDR) DRAM operation.

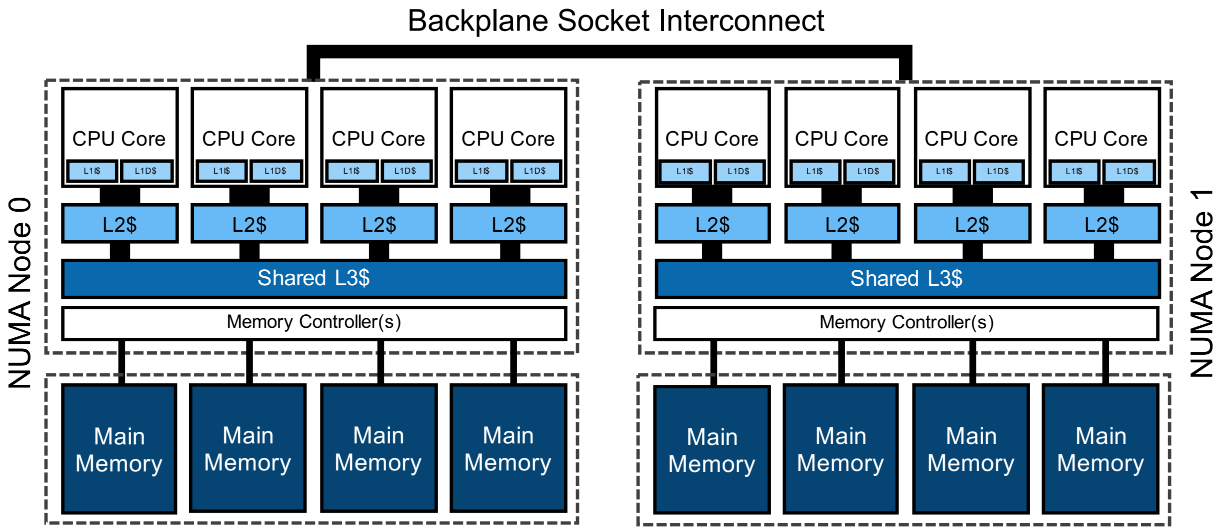


Figure 4.1: Typical server memory hierarchy with NUMA (repeat of Fig. 1.1).

4.3.1 Non-Uniform Memory Access (NUMA)

In most high-end machines typically deployed as enterprise workstations or as servers in the datacenter, multiple processor packages (nodes) are used to achieve higher core counts and off-chip main memory DRAM capacity. Although shared and globally addressable by all CPUs, the main memory is physically partitioned between the processor sockets, resulting in non-uniform memory access (NUMA) latency and throughput from each node.

A typical server with two identical NUMA nodes is depicted in Fig. 4.1.[2] In this cache-coherent example system, each processor core has private L1 data and instruction caches, as well as a unified private L2 cache. A single L3 cache and a portion of system main memory is shared by all cores on the node. For inter-socket cache coherence traffic and remote memory accesses, data crosses the backplane interconnect depicted at the top of the figure. This interconnect is the primary source of NUMA latency and throughput effects.

Because the performance impact of remote memory accesses can be significant, modern OSes provide software mechanisms that allow applications to provide hints for memory allocation and thread scheduling. However, applications that do not leverage these special NUMA APIs could

[2]The depicted architecture is generally similar to that of our experimental *Server* platform that will be described later in Sec. 4.5, except with only four cores per socket instead of 12, and an L3 bus instead of a ring-based interconnect.

degrade performance for themselves as well as other processes by generating unnecessary inter-socket traffic. In symmetric multiprocessing (SMP) NUMA systems, performance is theoretically independent of the exact CPU and memory node used. In our first case study in Sec. 4.6.1, we will show using X-Mem that this is *not* always true in realistic settings.

4.3.2 DDRx DRAM Operation

In a typical server DDR3 memory system [25], each processor package has from one to four *channels* (buses), each 64-bits wide for data and independently managed by integrated memory controllers. Each channel is typically populated with one or two *DIMM(s)*, each of which can have several ranks. A *rank* is an independent group of DRAM chips operating in lockstep that share common address, command, clock, and rank (chip) select signals. DRAM chips in a rank group their data pins together to interface with the 64-bit wide memory channel, which may be shared with other independent ranks. In systems that support error correcting codes (ECC), the bus is widened to 72 bits, of which eight bits are used for parity and stored on extra dedicated DRAM chips in each rank.

An individual DRAM chip is organized into several independent addressable *banks*. The banks share the DRAM's data I/O pins which are connected to the channel. Thus, while the banks can serve requests concurrently, only one bank may be transferring data on the channel I/O on any given clock cycle. Finally, each bank consists of a logical memory *array*, which is addressed by *rows* and *columns*. Thus, a physical memory address can be mapped to a unique location by channel, rank, bank, row, and column. The smallest addressable unit of memory is a byte.

A typical DDR3 DRAM system is depicted in Fig. 4.2. In this illustration, there are four channels of DDR3 DRAM with ECC. Each channel is composed of four DIMMs with one rank per DIMM. The DRAMs have a x8 pin configuration and each consist of four independent banks that operate in lockstep across the rank. The ninth parity DRAM used for ECC in each rank is not shown.

To read or write from a location in DRAM, the memory controller must first *open* (or `activate` with an `ACT` command) the designated row within a bank, which loads the data into a *row buffer*

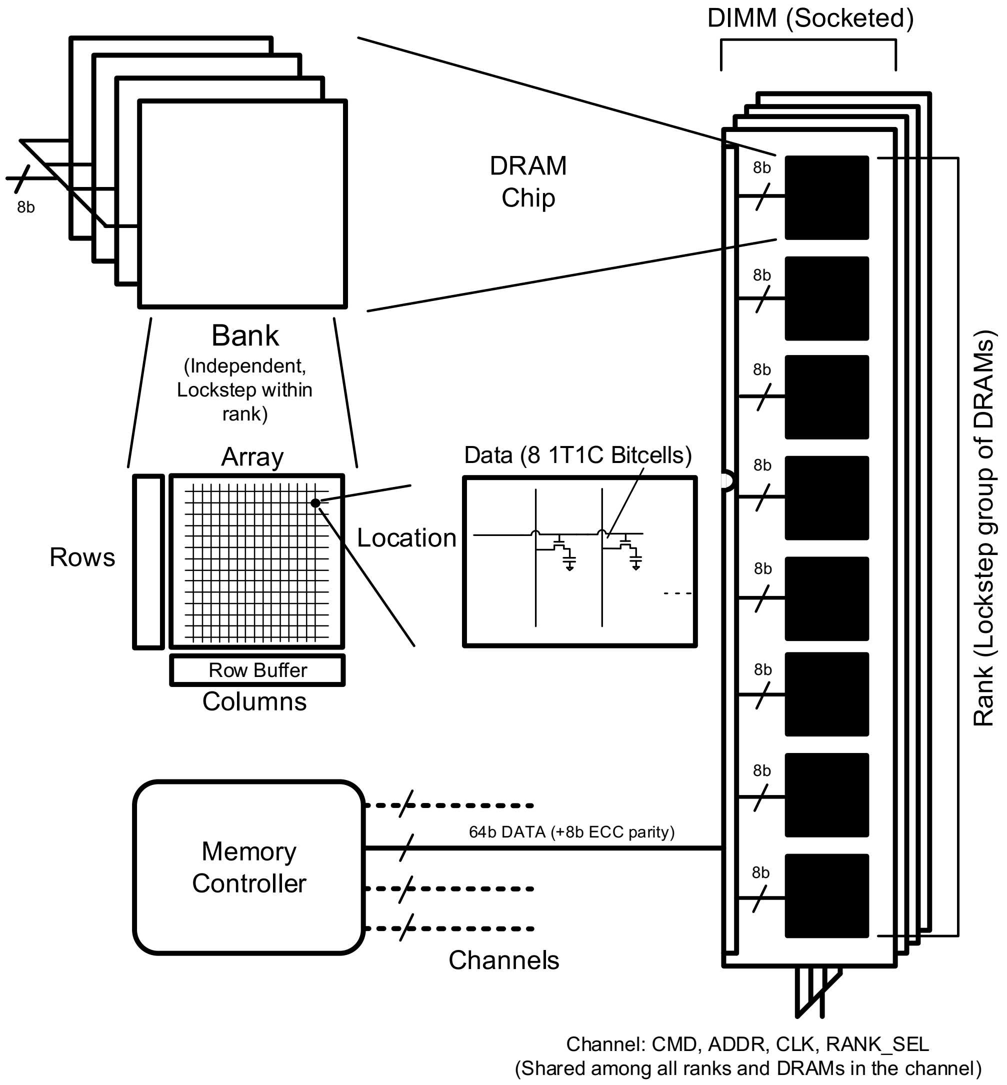


Figure 4.2: Anatomy of a typical DDR3 DRAM memory subsystem (repeat of Fig. 1.2).

within the bank. All banks on a chip share the same I/O. `READ` and `WRITE` commands can then be performed on the data in the row buffer. To access a different row in the bank, the currently open row must be *closed* (or *precharged* with a `PRE` command), as only one row may be open at a time per bank.

The JEDEC DDRx standards – e.g., DDR3 [25] – have many timing constraints between various DRAM commands to ensure correct operation. Of these constraints, several are particularly important to access scheduling in DDR3: `nCAS`, `nRCD`, `nRP`, and `nRAS`. These parameters are explained in Table 4.2.[3] The parameters in Table 4.2 represent integer multiples of the DDRx clock

[3]Other commonly referred parameters are `nRD`, `nWR`, refresh rates, etc., which are not considered in this chapter.

Table 4.2: DDR3 timing parameters that will be evaluated in this chapter

DRAM Timing Parameter	Description
`nCAS`	Column Access Strobe. Internal DRAM delay between a `READ` to an open row and data available at I/O pins.
`nRCD`	RAS to CAS Delay. *Required* delay between an `ACT` command and the first `READ` or `WRITE` to the open row.
`nRP`	RAS Precharge. *Required* delay between a `PRE` command and the next `ACT` command to a different row.
`nRAS`	Row Access Strobe. *Required* row cycle time. Minimum time between successive `ACT` or `PRE` commands in a bank. Must be at least $\texttt{nRCD} + \texttt{nRP}$.

period, as indicated by the prefix "n." Note that the DDRx clock frequency is half the data transfer rate, as data is transferred across the channel on both edges of the clock. Due to the complexity of the memory system, one cannot directly determine the impact of these timings on overall performance of the memory hierarchy and applications running on the CPU. We use X-Mem in our third case study in Sec. 4.6.3 to quantify the effect of different timing parameters for a mix of DDR3 channel frequencies.

4.4 X-Mem: Design and Implementation

We now discuss the important design decisions and implementation details behind X-Mem. These are organized according to the four functional requirements for cloud platforms described in the introduction. The *tool extensibility* aspect is divided among subsections for *access pattern diversity* (Sec. 4.4.1), *platform variability* (Sec. 4.4.2), and *metric flexibility* (Sec. 4.4.3). Fig. 4.3 depicts the high-level software organization that will be referred throughout the section.[4]

4.4.1 Access Pattern Diversity

The diversity of access patterns supported by X-Mem is important for characterizing and designing cloud systems. Even without writing custom extensions, users can often stimulate memory using a

[4]All specific features listed in this section are current as of X-Mem version 2.2.3.

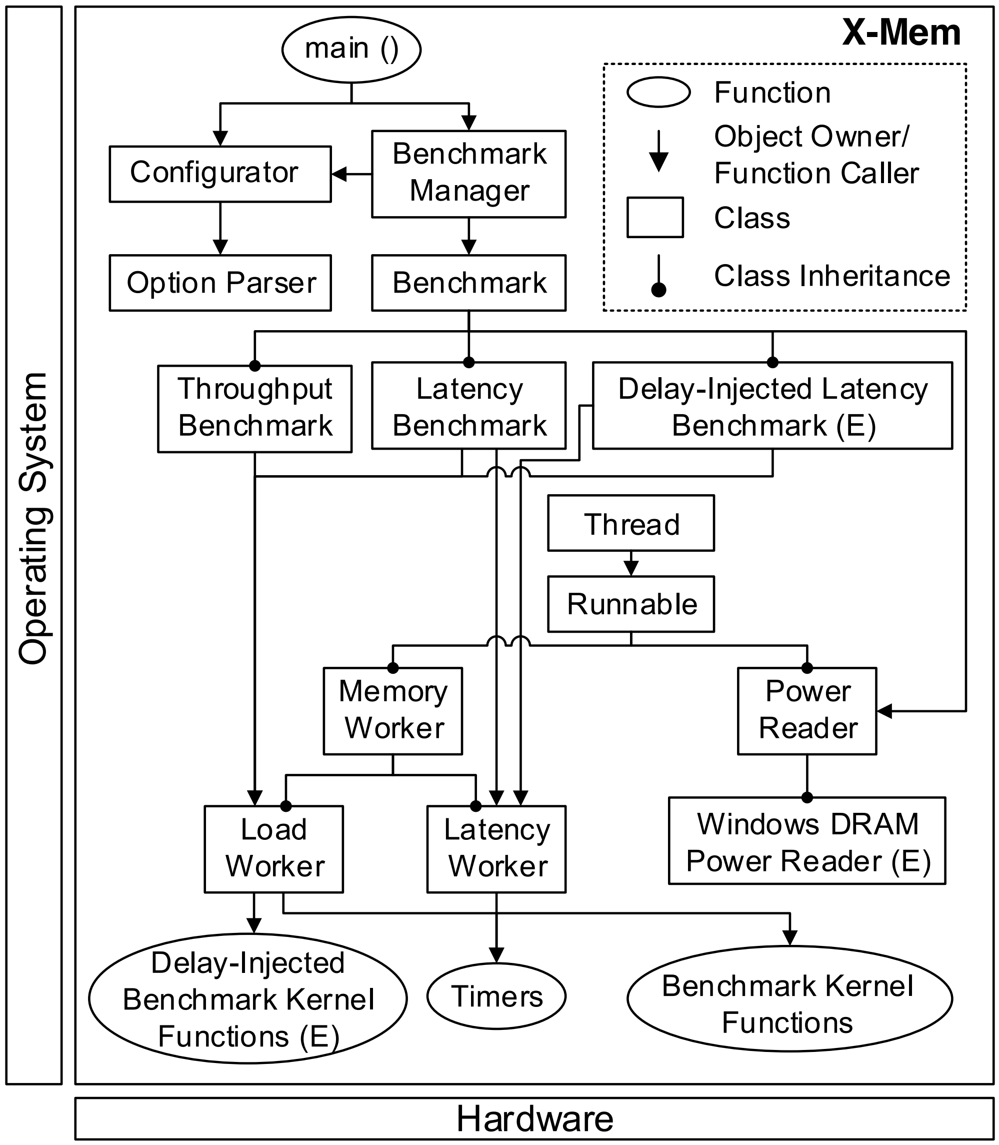


Figure 4.3: The high-level X-Mem software organization facilitates portability through rigid OS-/hardware abstractions. Its modular design enables simple prototyping of extended capabilities *(E)*.

specific access pattern that resembles an important phase of application behavior. This can enable cloud subscribers to better optimize their code for the memory organization of their target platform. Cloud providers can use such functionality to evaluate candidate memory configurations for different classes of applications. Computer architects could even use X-Mem to evaluate memory system optimizations early in the design or prototyping phases without running a full application.

At a high level, the user input causes a set of unique memory `Benchmarks` to be constructed by a global `BenchmarkManager`. The manager object generally allocates a large contiguous array on each NUMA node using a specified page size, and carves up the space as needed for each `Benchmark`. `Benchmarks` are run one at a time, where each is multi-threaded. There are two types of `Benchmark`: `ThroughputBenchmark` and `LatencyBenchmark`. Benchmarks can employ `LoadWorker` threads that measure memory throughput, and `LatencyWorker` threads to measure either loaded or unloaded latency, depending on whether other `LoadWorker` threads are running concurrently. Both workers types are descended from the `MemoryWorker` class.

To ensure consistency of results, each benchmark must be primed before execution. To avoid OS interference, `MemoryWorkers` lock themselves to designated logical CPU cores and elevate their thread scheduling priority. The workers then prime their tests by running them several times before an official timed pass. This helps to accomplish three things: *(i)* the instruction cache is warmed up with the core benchmark code; *(ii)* the data cache(s) are warmed up with (part of) the working set; and *(iii)* the CPU is stressed sufficiently enough that it is likely in a high-performance state when the benchmark begins (e.g., maximum voltage/frequency setting).

Each `Benchmark` gives its `MemoryWorkers` a pointer to an appropriate *benchmark kernel function* and a corresponding *dummy benchmark kernel function*. The dummy is used to quantify the overheads associated with the non-memory access parts of the benchmark kernel function, which may include the function call and sparse branch instructions. During a benchmark, each `MemoryWorker` repeatedly executes its benchmark kernel function until the cumulative elapsed time reaches a target raw duration, T_{raw}, which is configurable at compile-time and defaults to 250 ms. The number of iterations of the benchmark kernel function is recorded; the dummy benchmark kernel function is repeated for same number of times. The execution time for the dummy kernel T_{dummy} is subtracted from T_{raw} to obtain the worker's final adjusted time, T_{adjusted}.

Each benchmark kernel accesses exactly 4 KiB of memory before returning. This allows the function caller to measure the throughput/latency distribution of the memory access pattern over many chained iterations, regardless of the thread's working set size, which might vary from KiBs to GiBs. The decision to use 4 KiB per function call is a compromise between precision, accuracy, flexibility, and overhead. It provides sufficiently fine granularity to benchmark small L1 caches and avoids crossing typical page boundaries. At the same time, it is large enough to keep the function overhead low and to be accurately captured with high resolution timers.

X-Mem's low-level *benchmark kernel functions* include many different memory access patterns. Each of these global kernel functions implements a unique combination of the following: *(i) type*, currently pure-load or pure-store; *(ii) structure*, which currently include *sequential*, *strided*, and purely *random* addressing; and *(iii) chunk size*, which is the access width for a single memory instruction. X-Mem presently supports strides in both forward and reverse directions, with lengths of $\pm\{1,2,4,8,16\}$ chunk multiples. We currently include four chunk sizes in the standard release of X-Mem as of v2.3: 32, 64, 128, and 256 bits wide. Unsupported chunk sizes for each platform are disabled.

The random access benchmark kernel functions (used by the `LatencyWorker` as well as some `LoadWorkers`) were implemented as a *pointer-chasing* scheme that creates a chain of dependent reads to random addresses. This forces only one memory request to be outstanding at a time, ensuring that the average access latency can be accurately measured over many chased pointers. In this work, the chain is constructed by initializing a contiguous array of pointers to all point at themselves and then randomly shuffling the array. An alternative technique is to construct a random Hamiltonian Cycle of pointers. Both techniques are $O(N)$, but the random shuffle approach ran much faster on our machines. However, with the random shuffle method, a series of pointers may occasionally form a small cycle that "traps" the kernel function, effectively shrinking the intended working set size. This can cause incorrect results but can be mitigated by using multiple iterations or by using the Hamiltonian Cycle technique instead. Nevertheless, in most cases, the latency measurements generated using the two methods are indistinguishable.

The number of unique micro-benchmarks is many times greater than the 88 currently-included benchmark kernel functions would suggest. For instance, the user may specify the number of worker threads for each micro-benchmark. Each thread can have a different memory region working set size and kernel benchmark function. These working set regions can be allocated in a NUMA-aware fashion with configurable page size and adjustable alignment. It is also possible for groups of worker threads to use overlapped memory regions.

Although X-Mem currently implements a diverse set of memory access patterns, the tool may see uses beyond the case studies presented in this chapter. Thus, we designed the tool to allow for the addition of new access patterns with minimal modifications by abstracting core memory access functionality in the individual benchmark kernel functions. We believe this is a key requirement for cloud providers that host diverse third-party applications and also for the subscribed application developers. Rich functionality can be added by merely writing a few extra specialized benchmark kernel functions. To leverage these, developers can lightly modify existing derivations of the `Benchmark` and `MemoryWorker` classes to use their new kernels, or write their own derived classes. A simple extension might support wider vector memory instructions such as AVX-512 [243]. This could be done with a new 512-bit chunk size option and copying and modifying the existing 256-bit benchmark kernels to use the wider instructions and registers. In another example, the standard X-Mem release includes a third derived `Benchmark` type: the `DelayInjectedLatencyBenchmark`. This class implements a special version of the multi-threaded loaded `LatencyBenchmark`, where the `LoadWorker` use slightly modified benchmark kernel functions with `nop` instructions interspersed between memory access instructions. This has proven useful for characterizing main memory latency when subjected to a wide range of load traffic.

More radical extensions are also possible with relatively little effort. Specialized access patterns for benchmarking translation-lookaside buffer (TLB) performance or measuring inter-cache communication latency with variable load interference can be built on top of the existing codebase. For security research, a small benchmark kernel can be added that performs Rowhammer-like DRAM attacks [244, 245]. A memory power "virus" might be written to test server power capping techniques. Benchmarks for characterizing data dependence of memory power [2] and performance could be crafted.

4.4.2 Platform Variability

To help cloud subscribers gain insight on various platforms, we designed X-Mem to support different combinations of hardware and software. X-Mem currently runs on many cloud-relevant platforms that span different ISAs, hardware features, and OSes. Currently supported architectures include x86, x86-64 with optional AVX extensions, ARMv7-A with optional NEON extensions, and ARMv8 (64-bit). GNU/Linux and Windows are currently supported on each architecture.[5]

X-Mem abstracts OS and hardware-specific interfaces and semantics wherever possible. Two classic C/C++ language features were used to achieve this: *(i)* ***typedef*** is used to abstract ISA-specific datatypes for vector-based wide memory access, and *(ii) pre-processor macros* that guard OS or architecture-specific code. In addition to language features, X-Mem wraps OS APIs. For example, generic functions are used to pin worker threads to logical cores and to elevate thread priority. However, X-Mem cannot control the semantics of these OS services. Whenever they cannot be controlled, the tool is clear to the user and programmer about possible sources of deviation in reported results.

Each benchmark kernel function and its dummy had to be carefully hand-crafted to stimulate memory in a "correct" manner for characterization on each platform. This helps ensure that measured results can be compared fairly as possible. Whenever possible, the implementations of the benchmark kernel functions use several tricks to defeat compiler optimizations in the important sections of code without resorting to un-portable inline assembly. Two examples include *manual loop unrolling* to control branching overheads, and the use of the `volatile` keyword to keep the compiler from pruning away "meaningless" memory reads and writes that are critical to benchmark correctness.

The execution time for an unrolled loop of benchmark kernels is measured using X-Mem's `start_timer()` and `stop_timer()` functions. Internally, these functions use a high-resolution timer, whose implementation is specified at compile-time as an OS-based or hardware-based timer. OS timers are implemented essentially with POSIX `clock_gettime()` and `CLOCK_MONOTONIC` or Windows' `QueryPerformanceCounter()`. Hardware timers are less portable, even for the same ISA, but they enable finer-grain timing for very short routines. On Intel systems, the Time Stamp Counter (TSC) register is sampled using the special `rdtsc` and `rdtscp` partial serializing instructions along with the `cpuid` fully-serializing instruction [246].[6] Our testing has shown that for the default $T_{\text{raw}} = 250ms$ benchmark duration, there is no measurable difference between hardware

[5]We were able to compile, but not link X-Mem for Windows/ARM combinations due to a lack of library support for desktop ARM applications.

[6]Benchmark workers are always pinned to a single core, and timer values are never directly compared across workers, so users can choose TSC in their Intel-targeted X-Mem builds if needed.

and OS timers in X-Mem. OS timers are used by default to aid portability, although this option and T_{raw} can be easily changed at compile time.

The tool generates results as fairly as possible to allow for "apples-to-apples" comparisons of memory systems. We use Python-based SCons [247] to simplify the build process and maximize portability. On GNU/Linux builds, we verified that the g++ compiler generates the intended code on each platform by disassembling and inspecting the X-Mem executables. On Windows builds, the Visual C++ compiler cannot generate AVX instructions for our variables that were intentionally tagged with the `volatile` keyword. On the other hand, it also does not support inline assembly code for an x86-64 target. Thus, on Windows/x86-64/AVX-specific builds, we were forced to implement all SIMD-based benchmark kernels by hand in the assembler. Nevertheless, compiled code and our manual implementations were nearly identical. We also verified the equivalence of benchmark results experimentally.

Ports to other OSes and architectures are possible with relatively straightforward extensions to X-Mem's source code and build toolchain, thanks to its heavy use of abstractions. Many platform-specific features can be enabled or disabled at compile time through the use of included preprocessor switches.

4.4.3 Metric Flexibility

X-Mem can measure performance and power of the memory hierarchy, where a number of statistics can be recorded for each of X-Mem's diverse access patterns. X-Mem currently reports on several application-visible performance categories such as unloaded latency (no background traffic), loaded latency (variable controlled background traffic) and aggregate throughput. It can also sample memory power during stimulated performance benchmarking. Measurements can be made for each level of the memory hierarchy, from CPU caches all the way to main memory.

X-Mem's metric flexibility is useful to both cloud subscribers and providers in quantifying subtle hardware characteristics. For example, a programmer working on a search application could find that the distribution of DRAM loaded latency is strongly correlated with the distribution of query latency. Such an insight would not be possible to achieve with only the arithmetic mean of memory latency. Specifically for cloud providers, average power can be used for optimizing performance per Watt, and peak power can be used for power provisioning purposes [210].

With regard to performance benchmarking, X-Mem actively stimulates memory and measures the real behavior of the hierarchy as could be seen by an application running on the CPU. The metrics capture the overall impact of the underlying platform architecture and associated configuration settings on performance, but low-level secondary effects are not disaggregated. This is distinct from a passive performance counter-based approach, which is better suited to breaking down individual performance components, but often cannot make end-to-end measurements. We believe the active stimulation method used by X-Mem is a more useful measurement approach for the cloud usage model, which is concerned primarily about ground truth memory hierarchy performance from the application's point of view. It also has the benefit of being much more flexible and portable than approaches that rely on model-specific performance counters.

Each user-visible *benchmark iteration* is composed of many *passes*. For each iteration, X-Mem maintains the arithmetic mean of the relevant metrics. If the benchmark is run for more than one iteration, these extra samples track the metric's distribution over time. We consider this a useful feature for evaluating interference effects, as concurrent execution of other applications on the platform can influence the measurement of unloaded latency. In the absence of interference, by the central limit theorem, we expect the per-iteration results to approach a normal distribution. However, if there time-varying interference, the distribution can shift. For example, a second application can begin accessing DRAM heavily halfway through an X-Mem benchmark, which might add noise to X-Mem's active measurement of DRAM latency. The transient effects of this interference can be captured up to the resolution of a single benchmark iteration. This is on the order of 10s to 100s of milliseconds. The tool can be easily modified to trade off sampling accuracy for higher iteration sampling rates by adjusting T_{raw} at compile time.

Performance metrics are captured as follows. Memory throughput is reported in *MiB/s* by accumulating the results from all `LoadWorker` instances that execute concurrently. The unloaded latency metric is reported in *ns/access* without any other `MemoryWorkers` executing. For loaded latency, results are reported in ns/access from the `LatencyWorker` given a concurrent load stimulus driven by `LoadWorkers` and reported in MiB/s.

With regard to power metrics, average and peak numbers are indirectly measured for each benchmark iteration (sample). To achieve this, X-Mem provides the virtual `PowerReader` class as an interface that needs to be implemented for each a specific system. `PowerReader` executes a low-overhead background thread that regularly polls the power consumption of the memory during benchmark execution at a fixed sampling rate. The implementation of the `PowerReader` interface is left as an extended feature, as power instrumentation varies widely between systems and end-user needs often differ. By default, X-Mem includes the `WindowsDRAMPowerReader` extension, which leverages a generic software power meter exposed by the OS. On our *Server* platform evaluated later in the chapter, this meter relies on architecture-dependent Intel RAPL features to expose total DRAM power per socket. However, the underlying implementation of the OS meter is independent of X-Mem's `PowerReader` functionality. One could also implement `PowerReader` by using a dedicated hardware multimeter for each DIMM [1,2], improving measurement accuracy, precision, and granularity.

X-Mem can be easily extended to add new metrics of interest. For example, STT-RAM can have data-dependent energy consumption. It might be characterized in a novel way by using new data-based benchmark kernels along with data-aware power book-keeping. Systems with PCM could have their endurance and wear-leveling mechanisms [248] tested with a specialized load generator. Active data integrity testing may require new reliability metric extensions as well. Thus, our tool is flexible enough to suit specific needs of cloud providers and subscribers.

4.4.4 UI and Benchmark Management

In this section, we describe X-Mem's command-line user interface (UI) to demonstrate how the various described features integrate together at run-time.

Upon starting the process, X-Mem prints basic build information to command-line output, such as the version number, build date, target OS, and target architecture. Using OS-specific APIs, it then queries some necessary information from the run-time environment, such as the page size(s) and the number of physical and logical cores. The `BenchmarkManager` works together with a `Configurator` object to translate user inputs into a set of unique `Benchmark` objects to run in

Table 4.3: X-Mem UI options for v2.2.3. † indicates a repeatable option with argument. * indicates a default case.

Option	Args.	Description
`-a` or `---all`		Use all benchmark kernels.
`-c` or `---chunk_size`†	32, 64*, 128, 256	Memory access width for load kernels.
`-e` or `---extension`†	0, 1, ...	Run extended mode with given index.
`-f` or `---output_file`	FILE.csv	Dump benchmark results to a CSV file.
`-h` or `---help`		Print X-Mem usage and exit.
`-i` or `---base_test_index`	0*, 1, ...	Base of enumeration for benchmarks.
`-j` or `---num_worker_threads`	1*, 2, ...	Total number of threads to use.
`-l` or `---latency*`		Run (un)loaded latency benchmarks.
`-n` or `---iterations`	1*, 2, ...	Number of iterations per benchmark.
`-r` or `---random_access`		Use random-access load kernels.
`-s` or `---sequential_access`		Use seq. or strided-access load kernels.
`-t` or `---throughput*`		Run throughput benchmarks.
`-u` or `---ignore_numa`		Only test NUMA node 0.
`-v` or `---verbose`		Enable verbose standard output.
`-w` or `---working_set_size`	4*, 8, ...	Per-thread array size in KiB.
`-L` or `---large_pages`		Use OS-defined large pages.
`-R` or `---reads*`		Use read accesses for load kernels.
`-W` or `---writes*`		Use write accesses for load kernels.
`-S` or `---stride_size`†	±1*, 2, 4, 8, 16	Stride length in multiples of chunk size. Only applies to load kernels with `-s`.

sequence. A brief synopsis of each available option as of X-Mem v2.2.3 is shown in Table 4.3. For options with arguments, they may be specified in any of the following forms (using three iterations per benchmark as an example): `-n3`, `-n 3`, or `---iterations=3`. If the `---verbose` output mode is set, extra build and run-time information is printed to standard output along with other information such as the compile-time options (some of which are OS or architecture-specific). The user's interpreted options are also displayed.

Based on the selected user options, X-Mem's `BenchmarkManager` object builds potentially many `Benchmark` objects, each of which represents a single measurement scenario with a set of unique parameters. For example, for the "`-t -R -W -r -s -S1 -S-4 -c64 -c256`" command-line options, X-Mem would build and run a separate `ThroughputBenchmark` object for all possible combinations of CPU NUMA node, memory NUMA node, read and write-based kernels, random-access patterns, forward and reverse access in strides of 1 and -4, and access granularities (chunk sizes) of 64 and 256 bits.

For all constructed `Benchmarks`, the number of threads, working set size per thread, and page size are fixed based on the command-line parameters. In the above example, for each `Benchmark`, four identical load-generating threads (`-j4`) would each stress 64 MiB (`-w65536`) of unshared contiguous memory, allocated as large pages, using a single combination of the other parameters. On most systems, this would measure aggregate main memory throughput performance. Thus, to characterize the entire memory hierarchy with maximum degrees of freedom, X-Mem would need to be executed several times in sequence, sweeping the input parameters for the number of worker threads (`-j`), working set size per thread (`-w`), and page size (`-L`).

After each `Benchmark` completes, it displays the final results to standard output (with extra detail if the `-v` option was used) and writes its results to a new row in the optional CSV file. If the `Benchmark` results have any obvious inaccuracies, then a warning will be printed.

X-Mem's `Configurator` parses user inputs via the helpful third-party and open-source `OptionParser` class [249]. The `OptionParser` is ideally designed for rapid extensions to X-Mem's command-line interface, facilitating radical modifications to our tool without disrupting much of the existing functionality. We also provide by default the `-e` option to allow users to run more specialized modes.

4.5 Experimental Platforms and Tool Validation

We describe the experimental platforms used in the rest of the chapter and validate our tool. We picked seven systems to highlight the applicability of X-Mem to various platforms that may be used in the cloud.

The details of each system are shown in the top half of Table 4.4. The systems span different ISAs (*x86-64* and *ARMv7*), OSes (*Windows* and *GNU/Linux*), power budgets (*wimpy* and *brawny* systems), and virtualization (*bare metal* and *VM*). These platforms are: a *Desktop* workstation; a many-core rack-mountable cloud *Server*; a low-power x86 *Microserver*; an ARM *PandaBoard* ES [250]; an Azure cloud VM (*AzureVM*) [206]; an Amazon EC2 cloud VM (*AmazonVM*) [207]; and a Scaleway bare metal ARM cloud microserver (*ARMServer*) [208]. On Intel-based platforms, HyperThreading (SMT) and TurboBoost (DVFS) were disabled in the BIOS to obtain consistent results across multiple runs of X-Mem. We also tested the unloaded and loaded latency of the *Server* with main memory ECC disabled. This had no observable effect on performance, suggesting that the hardware ECC module is not actually bypassed in the memory controller when disabled. In the rest of the chapter, the presented *Server* results all have ECC enabled.

Table 4.4: Top: platforms used for X-Mem validation and case studies. Bottom: main memory configurations for the *Desktop* and *Server* platforms, where * indicates our default setting.

System Name	ISA	CPU	No. Cores	CPU Freq.	L1 Cache ($)	L2$	L3$	$ Blk.	Process	OS	NUMA	ECC
Desktop	x86-64 w/ AVX	Intel Core i7-3820 (Sandy Bridge-E)	4	3.6 GHz*, 1.2 GHz	split, private, 32 KiB, 8-way	private, 256 KiB, 8-way	shared, 10 MiB, 20-way	64 B	32 nm	Linux		
Server	x86-64 w/ AVX2	Dual Intel Xeon E5-2600 v3 series (Haswell-EP)	12 per CPU	2.4 GHz	split, private, 32 KiB, 8-way	private, 256 KiB, 8-way	shared, 30 MiB, 20-way	64 B	22 nm	Win.	✓	✓
Microserver	x86-64	Intel Atom S1240 (Centerton)	2	1.6 GHz	split, private, 24 KiB 6-way data, 32 KiB 8-way inst.	private, 512 KiB, 8-way	-	64 B	32nm	Win.		✓
PandaBoard (ES)	ARMv7-A w/ NEON	TI OMAP 4460 (ARM Cortex-A9)	2	1.2 GHz	split, private, 32 KiB, 4-way	shared, 1 MiB	-	32 B	45 nm	Linux		
AzureVM	x86-64	AMD Opteron 4171 HE	4	2.1 GHz	split, private, 64 KiB, 2-way	private, 512 KiB, 16-way	shared, 6 MiB, 48-way	64 B	45 nm	Linux		✓
AmazonVM	x86-64 w/ AVX2	Intel Xeon E5-2666 v3 (Haswell-EP)	4	2.9 GHz	split, private, 32 KiB, 8-way	private, 256 KiB, 8-way	shared, 25 MiB, 20-way	64 B	22 nm	Linux		✓
ARMServer	ARMv7-A	Marvell Armada 370 (ARM Cortex-A9)	4	1.2 GHz	split, private, 32 KiB, 4/8-way (I/D)	private, 256 KiB, 4-way	-	32 B	unk.	Linux		unk.

System Name	Config. Name	Memory Type	No. Channels	DPC	RPD	DIMM Capacity	Chan. MT/s	`nCAS` - clk (`tCAS` - ns)	`nRCD` - clk (`tRCD` - ns)	`nRP` - clk (`tRP` - ns)	`nRAS` - clk (`tRAS` - ns)
*Desktop**	*1333 MT/s, Nominal Timings 4C*	DDR3 U	4	2	2	2 GiB	1333	9 (13.5 ns)	9 (13.5 ns)	11 (16.5 ns)	24 (36.0 ns)
Desktop	*1333 MT/s, ≈33% Slower Timings 4C*	DDR3 U	4	2	2	2 GiB	1333	12 (18.0 ns)	12 (18.0 ns)	15 (22.5 ns)	32 (48.0 ns)
Desktop	*800 MT/s, Nominal Timings 4C*	DDR3 U	4	2	2	2 GiB	800	7 (17.5 ns)	7 (17.5 ns)	8 (20.0 ns)	16 (40.0 ns)
Desktop	*800 MT/s, ≈33% Slower Timings 4C*	DDR3 U	4	2	2	2 GiB	800	10 (25.0 ns)	10 (25.0 ns)	11 (27.5 ns)	22 (55.0 ns)
Desktop	*1333 MT/s, Nominal Timings 1C*	DDR3 U	1	2	2	2 GiB	1333	9 (13.5 ns)	9 (13.5 ns)	11 (16.5 ns)	24 (36.0 ns)
Desktop	*1333 MT/s, ≈33% Slower Timings 1C*	DDR3 U	1	2	2	2 GiB	1333	12 (18.0 ns)	12 (18.0 ns)	15 (22.5 ns)	32 (48.0 ns)
Desktop	*800 MT/s, Nominal Timings 1C*	DDR3 U	1	2	2	2 GiB	800	7 (17.5 ns)	7 (17.5 ns)	8 (20.0 ns)	16 (40.0 ns)
Desktop	*800 MT/s, ≈33% Slower Timings 1C*	DDR3 U	1	2	2	2 GiB	800	10 (25.0 ns)	10 (25.0 ns)	11 (27.5 ns)	22 (55.0 ns)
*Server**	*1333 MT/s, Nominal Timings*	DDR3 R	4 per CPU	1	2	16 GiB	1333	9 (13.5 ns)	9 (13.5 ns)	9 (13.5 ns)	24 (36.0 ns)
Server	*1333 MT/s, ≈33% Slower Timings*	DDR3 R	4 per CPU	1	2	16 GiB	1333	12 (18.0 ns)	12 (18.0 ns)	12 (18.0 ns)	32 (48.0 ns)
Server	*1600 MT/s, Nominal Timings*	DDR3 R	4 per CPU	1	2	16 GiB	1600	11 (13.75 ns)	11 (13.75 ns)	11 (13.75 ns)	29 (36.25 ns)
Server	*1600 MT/s, ≈33% Slower Timings*	DDR3 R	4 per CPU	1	2	16 GiB	1600	15 (18.75 ns)	15 (18.75 ns)	15 (18.75 ns)	38 (47.5 ns)
Server	*1867 MT/s, Nominal Timings*	DDR3 R	4 per CPU	1	2	16 GiB	1867	13 (13.92 ns)	13 (13.92 ns)	13 (13.92 ns)	34 (36.42 ns)
Server	*1867 MT/s, ≈33% Slower Timings*	DDR3 R	4 per CPU	1	2	16 GiB	1867	18 (19.28 ns)	18 (19.28 ns)	18 (19.28 ns)	46 (49.27 ns)

Table 4.5: Tested X-Mem build combinations. A checkmark ✓ indicates full build/test, while a circle ❍ represents incomplete build/test due to lack of platform support (for instance, we built X-Mem for 32-bit x86 and tested it on x86-64 processors due to a lack of native 32-bit Intel processors).

ISA	**OS**	**Built g++ 4.8**	**Tested**	**OS**	**Built VC++ 2013**	**Tested**
x86	Ubuntu 14.04	✓	❍	Win 8.1/Server 2012	✓	❍
x86-64	Ubuntu 14.04	✓	✓	Win 8.1/Server 2012	✓	✓
x86-64 w/ AVX	Ubuntu 14.04	✓	✓	Win 8.1/Server 2012	✓	✓
ARMv7	Ubuntu 14.04	✓	✓	Win 8.1/Server 2012	❍	
ARMv7 w/ NEON	Ubuntu 14.04	✓	❍	Win 8.1/Server 2012	❍	
ARMv8 (64-bit)	Ubuntu 14.04	✓		Win 8.1/Server 2012		

The bottom half of Table 4.4 describes the main memory configurations used for the *Desktop* and *Server* that will be used for Case Study 3. They are used to illustrate the impact of tuning various main memory parameters on performance. The primary memory knobs of interest include the number of (interleaved) channels, the channel data rate, and four primary DDR3 DRAM timing parameters: `tCAS`, `tRCD`, `tRP`, and `tRAS` [25]. In the *Desktop* and *Server*, the physically populated DIMMs were not changed across configurations or experiments. All memory parameters were changed via automated BIOS updates and system reboots.

The X-Mem tool was built and tested with a variety of OSes and hardware platforms, as shown by Table 4.5. These include x86 (32-bit), x86-64 (AMD64) with/without AVX SIMD extensions, and ARMv7-A with/without NEON SIMD extensions using g++ 4.8 on Ubuntu 14.04 LTS. The x86-family targets were also built with Visual C++ 2013 on Windows 8.1. We could compile but not link the ARMv7-A family targets for Windows on ARM due to lack of support for desktop CLI applications. The ARMv8 target was built for Linux, but was not tested due to lack of a suitable platform. On all working builds, X-Mem was tested extensively for each of the platforms listed in Table 4.4.

We validated X-Mem against a variety of tools on several platforms where applicable. For instance, we compare against Intel's Memory Latency Checker (MLC) v2.3 [233] for loaded DRAM latency which is only supported on Intel platforms. We choose MLC because it is a tool that has the most overlap in capabilities (see Table 4.1). To the best of our knowledge, it is the only tool which can perform a similar type of loaded latency measurement as X-Mem. We use our *Desktop* with

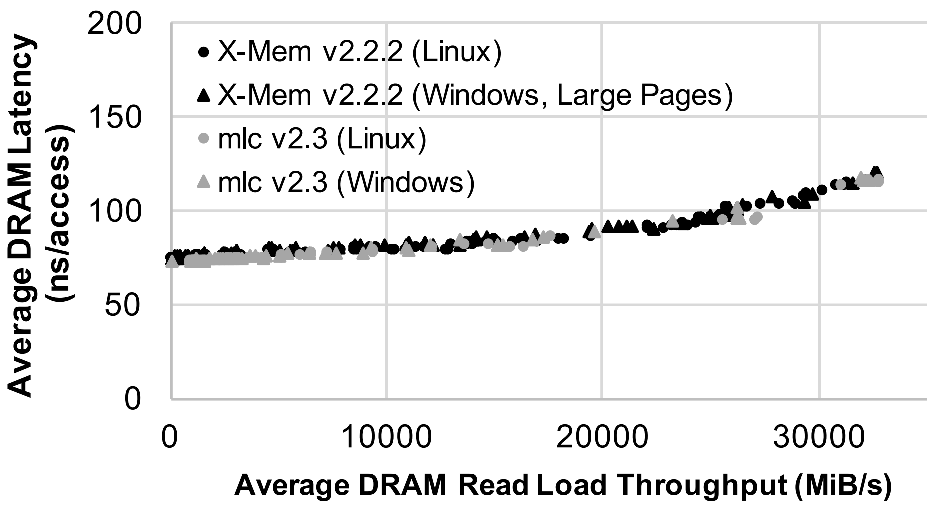


Figure 4.4: Validation of X-Mem vs. MLC [233] shows close agreement for total loaded latency measurements from CPU to DRAM on both Windows and Linux.

configuration *1333 MT/s, Nominal Timings 4C** at 3.6 GHz, running both Linux and Windows.

The MLC validation results are shown in Fig. 4.4, which shows the average total main memory latency versus the aggregate read-based memory load. We find close agreement in these results as well as other closely-matched tests that are not depicted.[7]

However, MLC is less portable, because it relies on Intel's proprietary hardware performance counters that may not be available in other systems. Therefore, as a consequence of relying on performance counters – unlike X-Mem – MLC does not necessarily capture the true *application-visible* performance.

We also validate X-Mem's measurements of aggregate DRAM throughput and loaded latency using Intel's Performance Counter Monitor (PCM) v2.8 [251]. We use our *Desktop* with configuration *1333 MT/s, Nominal Timings 4C** at 3.6 GHz. PCM indirectly calculated the average main memory throughput based on hardware model-specific registers (MSRs). For all tested read-based throughput benchmarks, the results agreed with PCM to within a fraction of a percent. For write-based throughput benchmarks, X-Mem again closely matched PCM's reported raw write throughput. In these cases, while PCM also reported raw read throughput (generated by cache-line fills, prefetches, etc.), X-Mem abstracts these secondary effects by reporting just the application-visible

[7]X-Mem for Windows used large pages as DRAM latency suffered considerably with normal page sizes on all of our Windows platforms.

throughput metrics (as described earlier in Sec. 4.4.3.

4.6 Case Study Evaluations

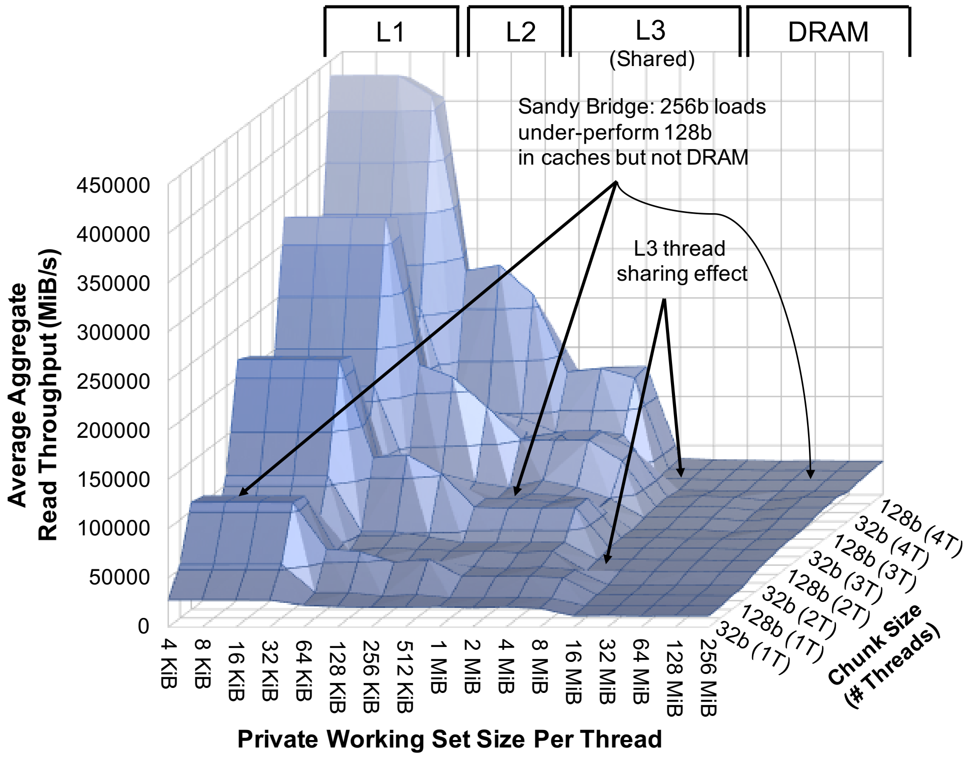
In this section, we leverage X-Mem's four key features – *diverse access patterns*, *platform variability*, *flexible metrics*, and *tool extensibility* – to present a trio of experimental case studies. The first two cover uses of X-Mem for cloud subscribers, while the last case study addresses cloud providers. We conducted an exhaustive sweep of all X-Mem micro-benchmark scenarios for each of our experimental platforms and observed many interesting details. However, for brevity, we present only a few key insights.

We begin in Case Study 1 with a quantitative exposure of cache organization details using the *Desktop*. Then we study the joint effects of page size and NUMA on DRAM performance and evaluate the energy proportionality of DRAM using our *Server* platform. Case Study 2 directly compares the cache hierarchies and main memory performance trends of all seven platforms that we consider. Finally, in Case Study 3, we evaluate the impact of tuning memory parameters on application-visible performance primarily using the *Desktop*.

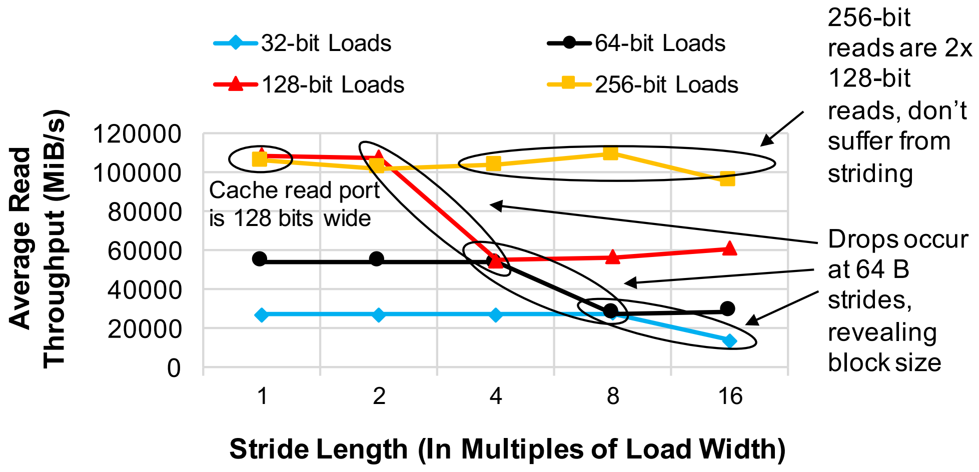
4.6.1 Case Study 1: Characterization of the Memory Hierarchy for Cloud Subscribers

Cloud subscribers would benefit from understanding the memory hierarchy of their platform. X-Mem facilitates this by mapping the memory performance with respect to application parameters such as number of threads, working set size, access patterns/granularity, and OS memory management policies. This procedure can reveal the underlying cache and memory hardware organization, allowing the programmer to exploit it.

We propose an intuitive visualization technique, the *memory landscape*, that depicts the aggregate memory throughput or latency as a surface plot over working set size per thread, number of threads, and chunk size. Fig. 4.5a shows the throughput landscape using a forward sequential read access pattern on the *Desktop* workstation. We make several observations. *(i)* Throughput falls off with increasing working set size (x axis), leaving a clear demarcation of cache/DRAM boundaries



(a) Memory hierarchy landscape



(b) Strided L1D cache behavior with 1 thread

Figure 4.5: Cache organization insights obtained using X-Mem can help cloud subscribers optimize application performance. The results shown are for the Sandy Bridge-E *Desktop* with config. *1333 MT/s, Nominal Timings 4C** at 3.6 GHz.

(labeled at the top of the figure). *(ii)* L1 and L2 throughput scales linearly with the number of threads (y axis). This confirms that the smaller caches are private to each core. In contrast, the sharing of the L3 cache among cores is illustrated by the outline of the "foothills" next to the flat DRAM "plain." *(iii)* DRAM performance scales linearly with number of threads and chunk size (Y-axis). We believe this is due to more efficient miss handling for wide load instructions. *(iv)* Although not depicted for brevity, using X-Mem's unloaded latency measurement capability, we found that both the L1D and L3 caches appear to have fixed access times in clock cycles (4 and 38 cycles, respectively) regardless of CPU frequency (ranging from nominal 3.6 GHz down to 1.2 GHz). In contrast, an L2 cache hit takes a variable number of CPU clock cycles as frequency is scaled. Such visualization enables programmers to reason about the general memory performance of their target cloud platform.

Another important consideration for programmers is the cache configuration, which could be hidden by the provider. We focus on the L1 data cache (L1D) of the Sandy Bridge-based *Desktop* as an example. Using X-Mem with a single thread and a working set size of just 4 KiB, we swept the chunk size and stride length (as a multiple of load chunk size). The results are shown in Fig. 4.5b.

We present three observations from the figure. *(i)* Observing the drops in throughput as a function of chunk and stride reveals the cache block/line size (64 B). The L1 block size can be inferred to be 64 B based on the dramatic drops in throughput at that stride length (red, black, and blue curves). This inference matches the *Desktop*'s known block size from Table 4.4. At this stride length, subsequent loads access different blocks in adjacent cache sets. We believe that the loss of performance is due to bank conflicts between adjacent sets. *(ii)* AVX 256-bit reads using the `vmovdqa` instruction perform no better than the 128-bit version for normal sequential accesses. Unlike the other chunk sizes, the 256-bit accesses maintain their performance as stride length increases. This suggests that a cache port is just 128 bits wide, and 256-bit accesses are simply split into two μ-ops. This observation is confirmed by a report on Sandy Bridge's implementation details [252]. *(iii)* L1D bandwidth can only be saturated using vectorized loads. Thus, for workloads that are already cache-friendly, further performance gains might be achieved through explicit SIMD memory-access optimization. X-Mem enables similar observations in the absence of public

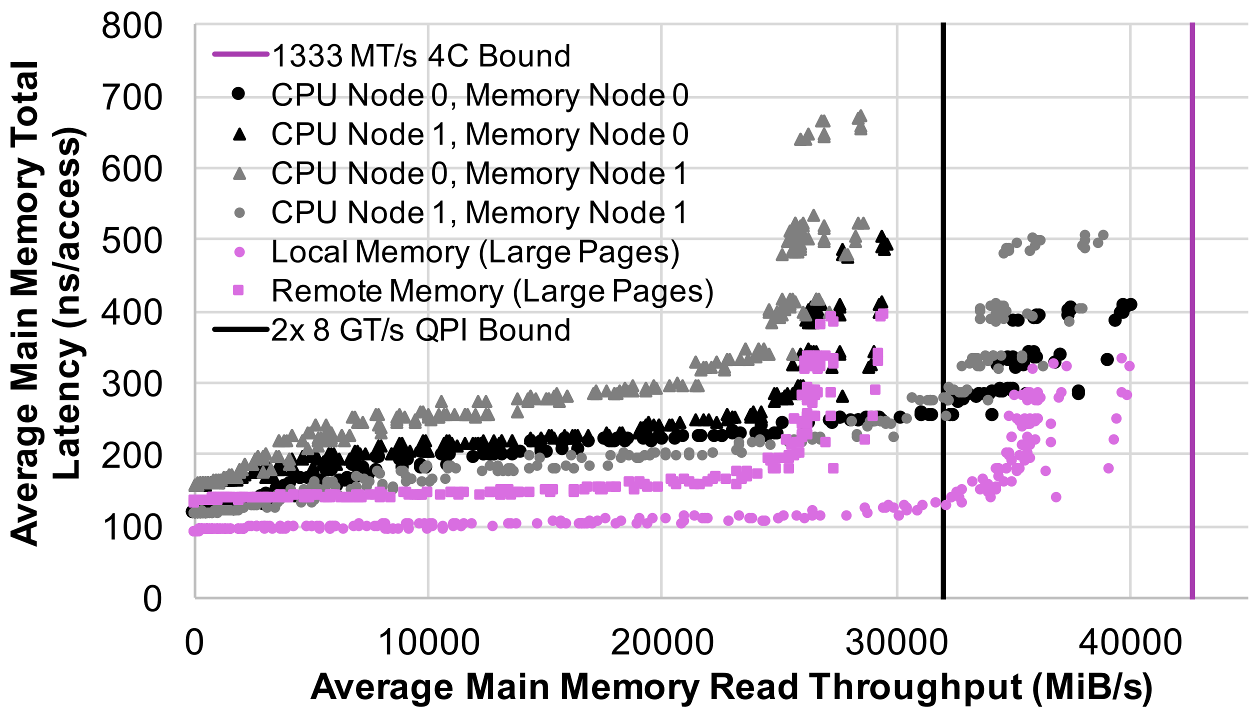


Figure 4.6: X-Mem reveals how significant main memory performance asymmetry may arise from the interaction of NUMA and page size. Results shown are for the Windows *Server* with configuration *1333 MT/s, Nominal Timings**.

information on the micro-architecture.

In addition to the hardware factors described thus far, OS memory management affects performance. To study this, we use a dual-socket NUMA *Server* platform running Windows that is typical of a cutting-edge cloud deployment. Fig. 4.6 shows the interaction of NUMA and page size on the loaded latency trend of main memory.

We present three observations from the figure. *(i)* The cross-socket QPI link forms a performance bottleneck for both memory latency and bandwidth. The latency curves stay flat until the load approaches the peak theoretical bandwidth, where queuing and contention begin to dominate delay, resulting in a *latency wall*. Remote access (triangles) incurs a latency penalty compared to local access (circles) even when there is no bandwidth starvation. This is due to the extra network hops required to reach remote memory. *(ii)* Large 2 MiB pages (pink points) reduce latency overall compared to regular-sized 4 KiB pages (black and gray points) due to reduced thrashing of the TLBs. *(iii)* For regular-sized pages, the two NUMA nodes have asymmetric local memory access

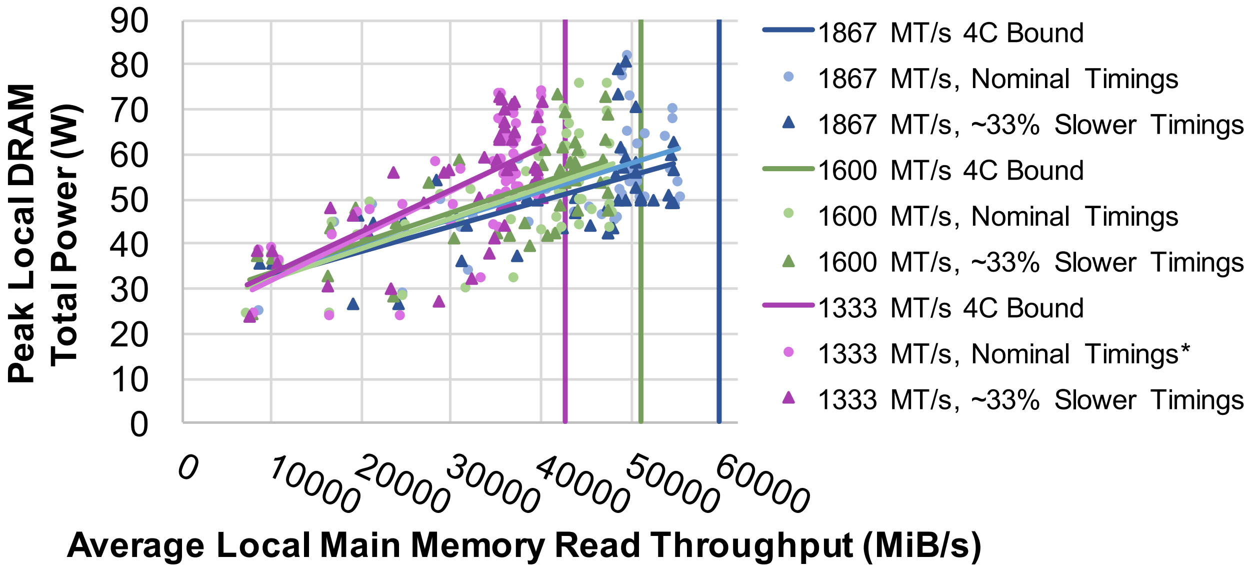


Figure 4.7: X-Mem can allow straightforward measurements of DRAM power and energy proportionality [39]. Results shown are for the Windows *Server* local NUMA node.

latency. CPU node 0 (black points) has better latency than node 1 (gray points). This suggests that page tables are stored on NUMA node 0 in Windows. For applications running on NUMA systems where large pages are not feasible, this could become a significant performance limitation. These three observations support the push for NUMA-aware cloud platforms [253]. Moreover, subscribers should consider using large page sizes for memory-bound applications. If this is not possible, then it may be preferable to use NUMA node 0 to obtain better memory performance (at least under Windows). For cloud providers, we recommend that hypervisors distribute page tables in a NUMA-aware fashion to avoid performance asymmetry becoming a bottleneck.

We now briefly consider DRAM energy proportionality, which is another crucial metric for cloud providers [39, 40, 182–184, 226]. The energy proportionality of *Server* main memory for different DDR3 channel frequencies and DRAM timing parameters is shown in Fig. 4.7.[8]

We make two observations from the figure. *(i)* There appears to be no significant impact from DRAM timing on the power vs. performance relationship. *(ii)* The memory subsystem initially appears to be more proportional at lower channel frequencies. However, the higher channel frequen-

[8]The physical DIMMs were capable of 1867 MT/s. For the lesser frequencies, we simply underclocked the DDR3 channels as listed in Table 4.4 instead of installing physically slower DIMMs. Thus, any difference in results is actually due to the configuration itself, not hardware variability [2, 228], and possibly how Intel's RAPL measures or models power.

cies on average consume less power for the same throughput. Notice that for all configurations, the DRAM power appears to reach the same rough ceiling of 60 W to 70 W. We believe this effect is due to channel contention. As throughput saturates, utilization also increases, and DRAMs cannot be put into low power idle states often.

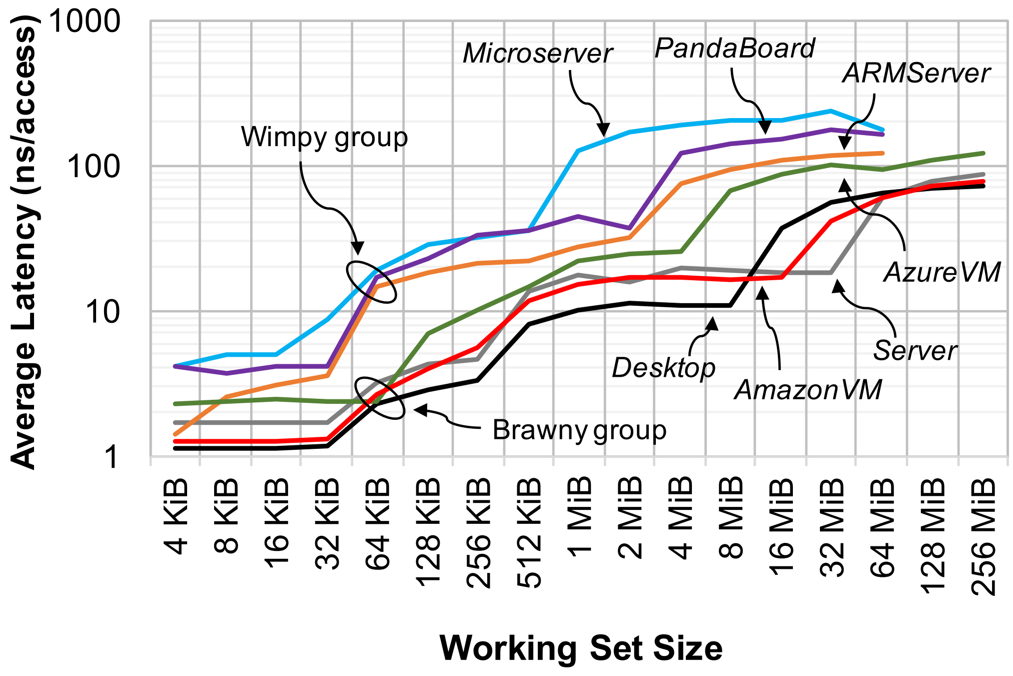
In this case study, X-Mem revealed micro-architectural and OS factors in memory performance for our *Desktop* and *Server*, enabling us to make recommendations for applications written for our particular platforms. In a similar fashion, cloud subscribers can apply X-Mem to their own platforms of interest and derive their own relevant insights to help optimize application performance. Meanwhile, X-Mem's unique ability to capture memory subsystem power alongside key performance metrics will enable cloud providers to identify platforms settings that offer the best performance per Watt for different classes of cloud applications.

4.6.2 Case Study 2: Cross-Platform Insights for Cloud Subscribers

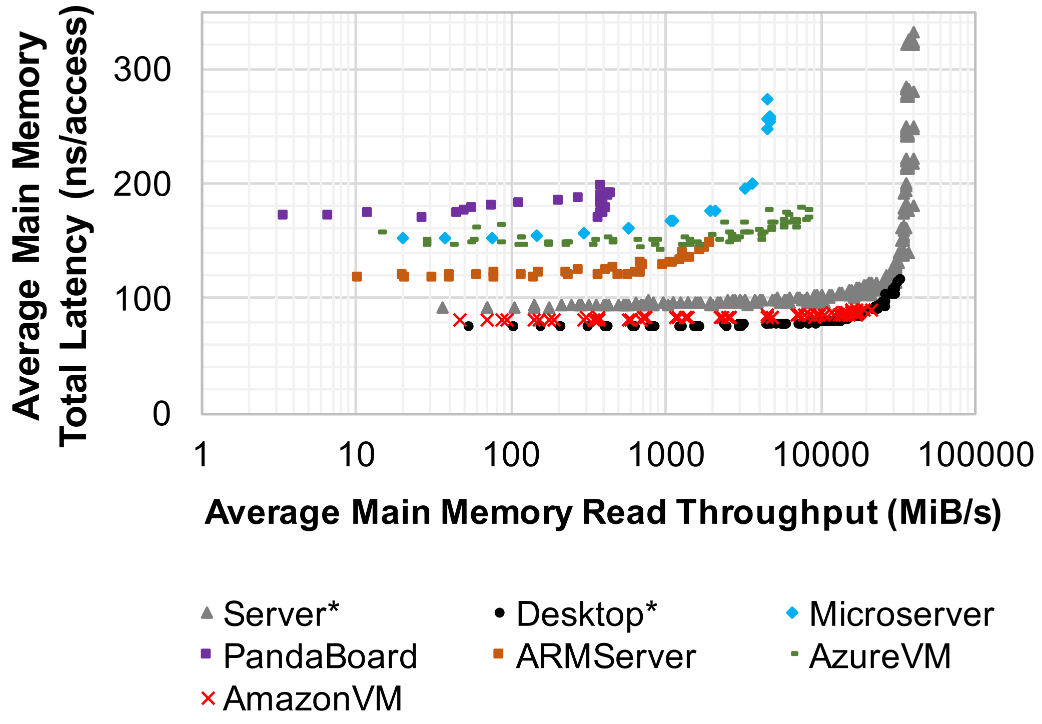
In this case study, we demonstrate X-Mem's ability to characterize the memory hierarchy of diverse platforms with a single tool. This is useful to cloud subscribers, who need to evaluate alternative platform choices as objectively as possible. We compare general performance aspects of caches and main memory across our seven platforms listed in Table 4.4, exposing the differences in *(i)* caches and main memory unloaded latency, *(ii)* main memory loaded latency, and *(iii)* read/write behavior among the systems. In our two public cloud virtual machines (*AzureVM* and *AmazonVM*), we had no way to directly control for interference from other cloud subscribers nor server-to-server heterogeneity in the datacenter; we repeated our experiments several times to ensure that our measurements were consistent over time.

An important step in choosing a suitable cloud platform is to understand the subtleties in memory hierarchy performance, which is heavily influenced by the cache organization (as discussed in Case Study 1). We examine average unloaded latency of each cache level by sweeping the working set size, which works even on virtualized hardware or if the cloud provider deliberately hides the specification.

Fig. 4.8a illustrates the results for all seven platforms. We find that the brawny high-power sys-



(a) Unloaded latency across cache levels in the memory hierarchy



(b) Loaded latency trends for main memory DRAM

Figure 4.8: X-Mem enables general cross-platform comparisons of cache and memory latency to help subscribers choose suitable providers for their applications. Results shown are for our seven diverse example systems.

tems (the *Desktop*, *Server*, *AzureVM*, and *AmazonVM*) and the wimpy low-power systems (the *Microserver*, *PandaBoard*, and *ARMServer*) form latency clusters, with considerable variation within each group. The *Desktop* and *ARMServer* slightly outperform their intra-group rivals at all cache levels. This is because they feature fewer cores and higher clock frequencies than their peers, but their cache sizes are not the largest. With regard to virtualization, the latency of *AmazonVM* does not suffer in comparison to its bare metal *Server* counterpart, which has near-identical CPU hardware (differences arise in clock frequencies, number of cores, and effective L3 cache size). *AzureVM*'s weaker showing is due to its much older hardware; it is possible there are more competitive VM instances in Azure that we did not receive. These hardware insights may be important to a programmer who only needs a few cores for an application that prefers fast cache access over capacity. In addition to helping subscribers choose an appropriate platform, X-Mem can help detect any performance heterogeneity across VM instances from a single provider.

The loaded latency of main memory is especially important in a cloud setting, where interference can play a significant role in application performance. Fig. 4.8b depicts results for our seven example platforms (X-axis in log scale).

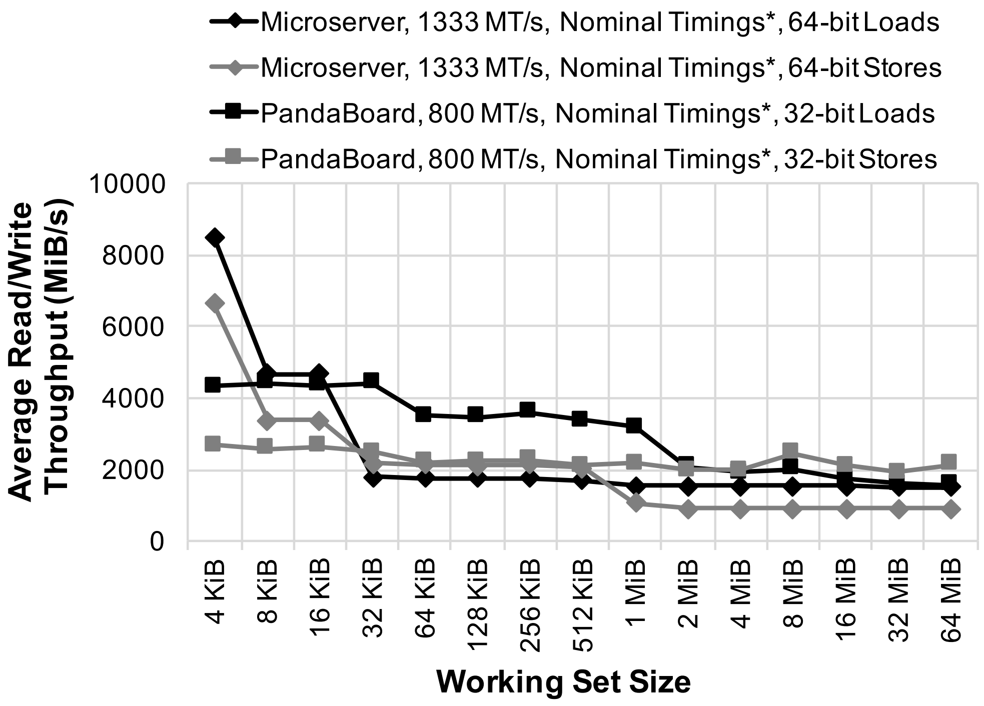
We make several observations from the figure. *(i)* Again, performance falls into the same two clusters: brawny and wimpy. *(ii)* The location of the latency wall varies drastically across platforms. The large latency gap between the low power *Microserver* and the other brawnier Intel systems is primarily due to its low clock frequency and also its in-order core design. Although the *PandaBoard* has better unloaded cache latency than the Atom-based *Microserver*, the former cannot match the latter's DRAM peak throughput or loaded latency curve. *(iii)* The *Desktop* features lower overall latency compared to the *Server* likely due to a lack of ECC and a smaller physical die size. They both have similar DRAM configurations, and their caches both appear to use write-back and write-allocate policies. *(iv)* While the *Server* hits a steep latency wall, the other systems do not saturate as severely. This can be attributed to the balance between CPU performance (e.g., clock frequency and the number of cores) and memory performance (e.g., peak channel bandwidth and rank/bank-level parallelism). For memory-intensive multi-threaded or multi-programmed workloads, the *Server* system would benefit from higher DDR frequencies. *(v)* Although the *PandaBoard* has on-package DRAM, it cannot achieve the same throughput or latency performance as the *Microserver*. Moreover, the *PandaBoard* does not exhibit a clear saturation effect, indicating that its memory bandwidth is over-provisioned.[9] Thus, X-Mem's ability to characterize the latency wall can be useful to cloud subscribers, who should choose platforms with an appropriate balance of performance. They could also use X-Mem's latency measurement capabilities to quantify the extent of cross-VM memory interference that results in performance inconsistency.

Finally, X-Mem can help reveal important performance aspects of cache and memory read and write behavior that vary among platforms. Comparative single-threaded throughput results are illustrated in Fig. 4.9.

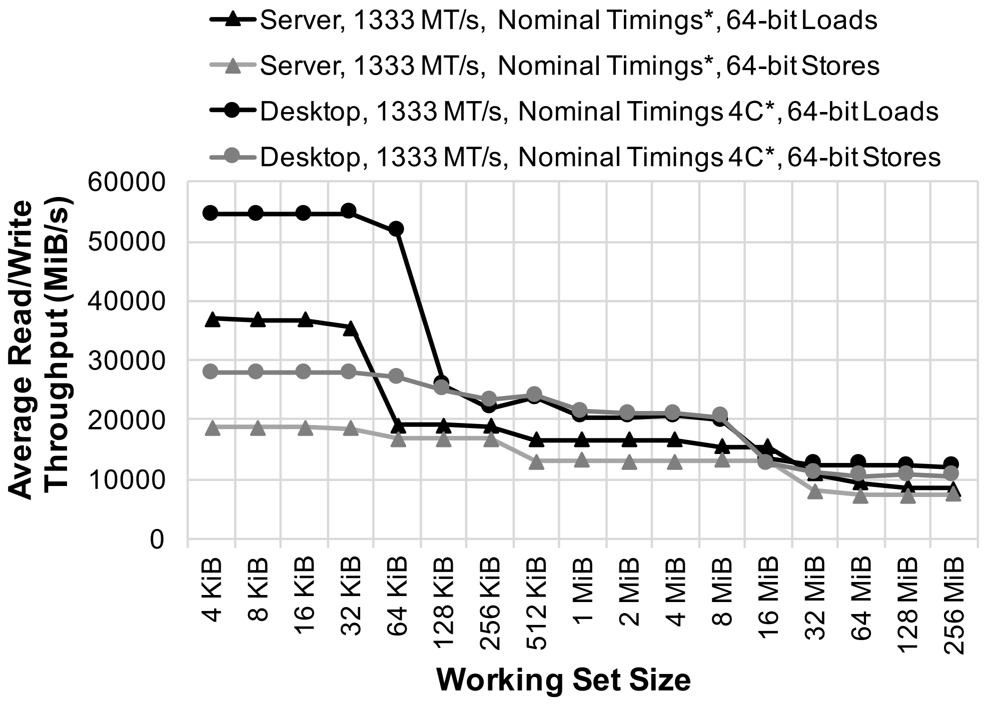
We make three observations. *(i)* From Fig. 4.9a, the *PandaBoard* featured nearly flat write throughput across the memory hierarchy, only outperforming reads for large working sets in DRAM. This indicates a combination of *write-through* and *write-around* cache policies. The other systems did not exhibit this effect. Instead, X-Mem revealed their *write-back* and *write-allocate* cache policies. We also noticed a strange behavior in the Atom-based *Microserver*: for working sets within the 24 KiB limit of the L1D cache, but larger than 4 KiB, throughput dropped by a factor of $2\times$ compared to the 4 KiB working set for both read and write operations. A 4 KiB working set maps to exactly one block in each cache set (the *Microserver L1D is 6-way set associative*), so we conjecture that the cache was somehow operating in direct-mapped mode, with five ways disabled, perhaps due to a system mis-configuration, which may have resulted in a reduced effective capacity. *(ii)* Our Intel systems (Fig. 4.9b exhibited a 2:1 read to write peak throughput ratio throughout the memory hierarchy; this means they have half as many L1 write ports as read ports. Such kinds of observations, enabled by X-Mem, can help cloud subscribers understand the strengths and weaknesses of different memory hierarchies, helping them to choose and configure the right platform for their read/write patterns.

Having a single cross-platform memory characterization tool facilitates the direct comparisons in this case study, aiding cloud subscribers to choose the right provider for their application. Such decisions are not trivial. For instance, memory latency-sensitive and throughput-sensitive applications may be suited to different platforms. We believe X-Mem helps to fulfill this important

[9]In the figure, one core was used for load generation, while the other was used for latency measurement. From our peak throughput measurements, even with two cores loading the DRAM, *PandaBoard* does not saturate the theoretical peak bandwidth of its memory at 3.2 GiB/s.



(a) *Microserver* vs. *PandaBoard*



(b) *Server* vs. *Desktop* (@ 3.6 GHz)

Figure 4.9: Cross-platform comparison of memory hierarchy throughput with a single thread.

role.

4.6.3 Case Study 3: Impact of Tuning Platform Configurations for Cloud Providers and Evaluating the Efficacy of Variation-Aware DRAM Timing Optimizations

The memory system plays an important role in CapEx and OpEx for cloud providers. The system must deliver competitive performance for as many applications as possible without incurring prohibitive provisioning and power delivery costs. At the same time, providers often cannot afford to specialize their hardware at design time for each class of application. In this final case study, we apply X-Mem to examine the efficacy of an alternative approach: *tuning platform configurations* to cater to DRAM main memory performance requirements as needed. *In particular, we are interested in using X-Mem to understand whether DRAM latency variation-aware performance tuning makes sense for a cloud scenario that has millions of modules and applications.*

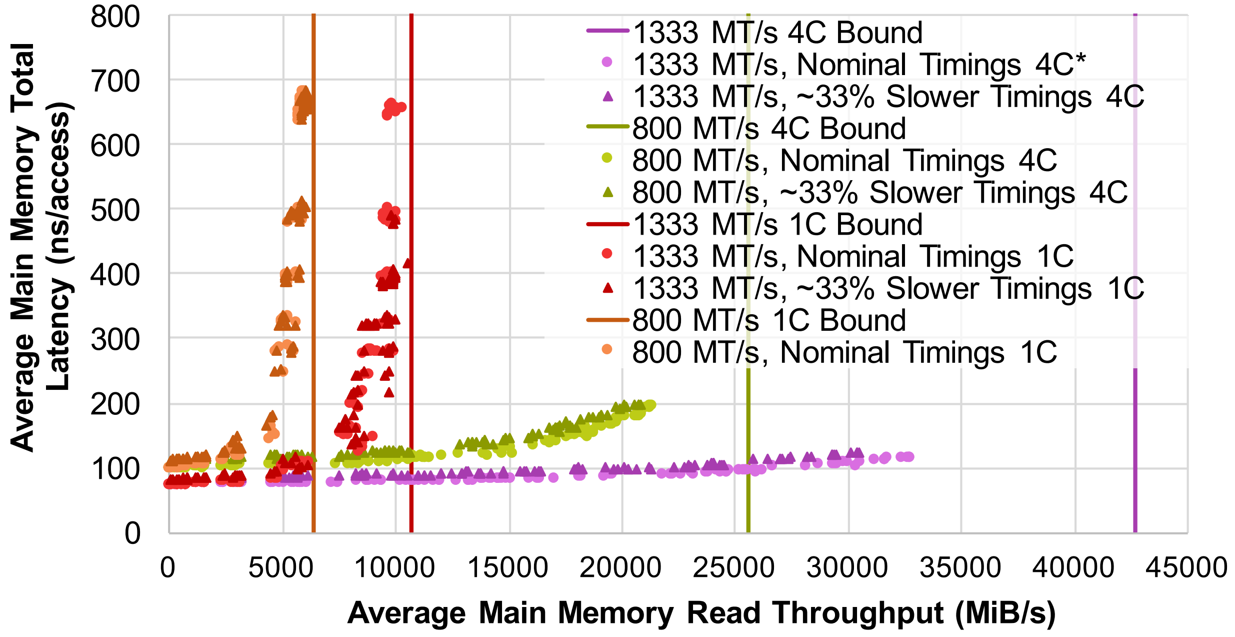
We consider *(i)* unloaded latency and *(ii)* loaded latency trends as functions of various firmware-controlled knobs. These knobs include CPU frequency, number of DRAM channels, channel frequency, and DRAM device timing parameters. To facilitate this study, we use the *Desktop* and *Server* platforms, each with two alternate DRAM timing settings as shown in Table 4.4: *Nominal Timings* and *≈33% Slower Timings*. In the latter case, DDR3 timing parameters `tCAS`, `tRCD`, `tRP`, and `tRAS` [25] were each slowed down on all channels by approximately 33% to imitate a slower (and cheaper) memory module, similar to those reported in [213, 228]. Unlike these two prior works, we did not tune the DRAM timing parameters to be faster than their datasheet values. Our primary goal in this case study concerns the sensitivity of memory performance to DDR3 channel frequencies and DRAM timings, in order to understand the potential benefit of variation-aware DRAM performance tuning.

Table 4.6: Sensitivity of unloaded main memory latency (in ns/access) with respect to various frequencies and timing parameters, enabled by X-Mem. Timing parameters have the greatest effect on unloaded latency when the CPU is fast and the memory bus is slow.

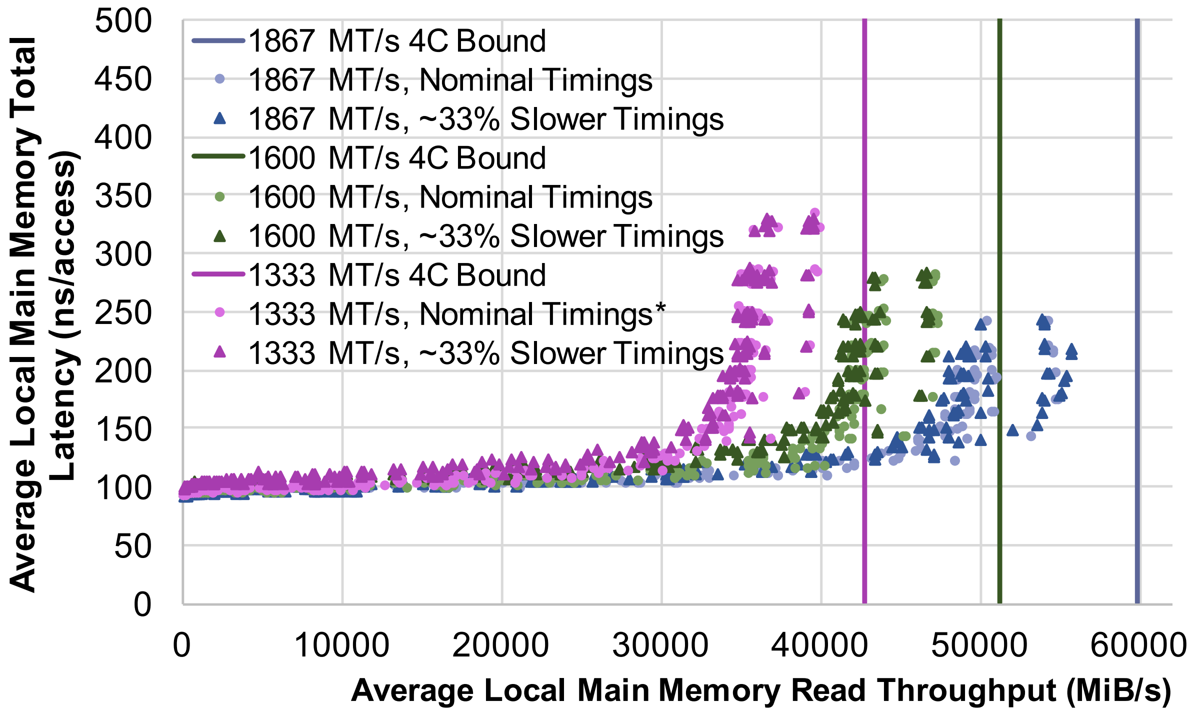
Mem. Channel Frequency →	***1867 MT/s***	***1867 MT/s***	***1600 MT/s***	***1600 MT/s***	***1333 MT/s***	***1333 MT/s***	***800 MT/s***	***800 MT/s***
Platforms ↓ Timings →	***Nom.***	***≈ 33% Slow***	***Nom.***	***≈ 33% Slow***	***Nom.***	***≈ 33% Slow***	***Nom.***	***≈ 33% Slow***
***Server* (NUMA Local, Lrg. Pgs.)**	91.43	91.54	91.66	95.74	91.99*	97.61	-	-
***Server* (NUMA Remote, Lrg. Pgs.)**	126.51	128.54	129.62	139.25	133.59*	141.69	-	-
***Desktop 4C* @ 3.6 GHz**	-	-	-	-	73.33*	81.91	97.21	110.89
***Desktop 1C* @ 3.6 GHz**	-	-	-	-	72.38	80.94	97.36	109.56
***Desktop 4C* @ 1.2 GHz**	-	-	-	-	109.65	118.25	131.86	145.76
***Desktop 1C* @ 1.2 GHz**	-	-	-	-	108.44	117.09	131.85	144.46

We consider the parameters influencing DRAM unloaded latency first. The results are summarized in Table 4.6. We make several observations. *(i)* Using the *Desktop*, CPU frequency has a significant impact: overall latency increases by up to 50% when the clock is scaled down from 3.6 GHz to 1.2 GHz. This is because the chip's "uncore" is slowed down along with the cores, causing the cache levels to consume more time in the critical path of DRAM access. *(ii)* On both systems, slower DDR3 DRAM timing parameters have a moderate effect at the 1333 MT/s baseline channel frequency (up to 12% on the *Desktop*), with generally less sensitivity on the *Server* system (up to 6%). This is because the *Server* has higher baseline cache latencies than the *Desktop* (as shown earlier in Case Study 2). The impact of even an aggressive $\approx$33% slowdown in DRAM timings on the *Server* is significantly less than the penalty of accessing remote NUMA memory. *(iii)* The gap between nominal and slower DRAM timing configurations narrows as DDR3 channel frequency is scaled up. At 1867 MT/s, the *Server*'s memory latency is impacted by as little as 1% for an aggressive $\approx$33% slowdown in DRAM timings. *(iv)* As the CPU frequency is reduced, the overall memory latency becomes less sensitive to DRAM timing (from 12% at 3.6 GHz to 7% at 1.2 GHz on the *Desktop*). *(v)* Finally, reducing the number of channels on the *Desktop* (*4C* to *1C*) has virtually no impact on unloaded memory latency (only $\approx$ 1 ns). This is because interleaving only affects the mapping of linear (physical) addresses to memory locations, and should have no impact when there is only one outstanding memory request at a time (as is done by X-Mem, which measures latency with random pointer chasing). These five observations suggest that cloud providers should carefully consider the platform configuration as a whole when making memory provisioning decisions. For example, there is likely an ideal balance between CPU and main memory frequency that minimizes overall memory latency for a latency-sensitive application under system cost and power constraints.

Next, we discuss the impact of memory channel width and transfer rates on main memory loaded latency. The results are depicted in Fig. 4.10 for the two platforms. Unlike the unloaded latency case, we find that both the number of channels and the channel frequency play significant roles. The number of channels (Fig. 4.10a) is the most important variable for memory performance under heavy loading, as the multiplication of available bandwidth dramatically flattens the latency wall. The quad-channel 1333 MT/s memory configuration is easily over-provisioned for the quad-



(a) *Desktop* @ 3.6 GHz



(b) *Server* with local NUMA memory & large pages

Figure 4.10: X-Mem enables read-loaded main memory latency measurements for various channel frequencies and DRAM timing configurations. Channel width and frequency have a strong effect on loaded latency, unlike the timing parameters.

Table 4.7: Percent slowdown caused by DRAM ≈*33% Slower Timings* for two memory-sensitive PARSEC applications on the *Desktop* system at 3.6 GHz with different application memory intensities (thread count).

Benchmark	**Config.**	**1T**	**2T**	**3T**	**4T**
canneal	*1333 MT/s 4C**	9.74%	9.02%	8.83%	8.89%
canneal	*800 MT/s 1C*	9.90%	9.29%	8.38%	7.83%
streamcluster	*1333 MT/s 4C**	11.14%	11.53%	11.82%	12.24%
streamcluster	*800 MT/s 1C*	8.10%	5.93%	2.63%	1.24%

core *Desktop*, but the same setup is woefully under-provisioned for the 12-core per socket *Server*. The latter requires frequencies of up to 1867 MT/s to mitigate the latency wall.

For the remainder of this case study, we focus on the impact of DRAM timing parameters on memory loaded latency and draw parallels to measured application performance. Our results obtained with X-Mem are shown in Fig. 4.10 (light circles for nominal DRAM timings and dark triangles for slower timings). The results indicate that the impact of DRAM timing parameters is relatively minor for loaded latency, in comparison to the unloaded latency case discussed earlier. This is because when the memory system is loaded, overall delay becomes increasingly dominated by resource contention, and less dependent on the "native" DRAM latency. However, in AL-DRAM [213], the authors found that tuning DRAM timings could significantly improve application performance, especially when memory bandwidth is scarce under loaded conditions. Our memory performance results seem to contradict those of AL-DRAM.

Thus, we decided to study this discrepancy further with two memory-intensive PARSEC benchmarks used in AL-DRAM [213]. We chose the `canneal` and `streamcluster` benchmarks – which are known to be memory intensive – that are used in AL-DRAM. The results are shown in Table 4.7 for two configurations on our *Desktop*. The table shows the percent difference in benchmark runtime, averaged over five runs, for each memory channel configuration (table rows) and number of PARSEC benchmark threads (table columns).

We find that both `canneal` and `streamcluster` are moderately sensitive to DRAM timings when there is sufficient memory bandwidth available (approximately 8% to 12% performance difference for the *1333 MT/s, 4C** cases). However, when the available channel bandwidth is reduced, or more load is placed on the memory, the sensitivity generally decreases (i.e., the *800*

MT/s, 1C cases, or increasing the number of threads). This small study appears to validate our claims made above using X-Mem: tuning DRAM timings should have a greater effect on lightly-loaded systems running latency-sensitive applications, but further investigation may be required. Nevertheless, X-Mem should prove to be an invaluable tool in conducting such investigations.

This case study highlights the ability of X-Mem to help cloud providers provision and configure their platforms to suit different performance requirements. It also could be used to infer third-party "black box" application memory performance characteristics without intrusive instrumentation. These application-level inferences could be useful for cloud providers to properly match subscribers' applications with best-configured available hardware.

4.7 Conclusion

In this chapter, we introduced X-Mem: a new open-source memory characterization tool [234,235]. X-Mem will bring value to both cloud subscribers and providers by helping them characterize the memory hierarchy and study its impact on application-visible performance and power. In contrast with prior tools, X-Mem addresses four key needs of cloud platforms: access pattern diversity, platform variability, metric flexibility, and tool extensibility. Our three case studies showed several examples of how the tool can be used to gain insights. In particular interest to Opportunistic Memory Systems, in Case Study 3 we found that that memory timing parameters should only be a secondary knob for cloud providers to tune system performance. Moreover, we conclude that exploiting memory process variation specifically for *latency* improvement in the cloud is not very useful. However, this assessment could change in the future, with different memory interfaces, protocols, process technologies, and emerging devices all playing significant roles. We leave this issue to future work. We hope that the broader community finds X-Mem useful and will extend it for future research, especially for the exploration of Opportunistic Memory Systems and application/memory performance tuning.

Part II

Opportunistically Coping with Memory Errors

CHAPTER 5

Performability: Measuring and Modeling the Impact of Corrected Memory Errors on Application Performance

Memory reliability is a key factor in the design of warehouse-scale computers. Prior work has focused on the performance overheads of memory fault-tolerance schemes when errors do not occur at all, and when detected but uncorrectable errors occur, which result in machine downtime and loss of availability. We focus on a common third scenario, namely, situations when hard but correctable faults exist in memory; these may cause an "avalanche" of corrected errors to occur on affected hardware. We expose how the hardware/software mechanisms for managing and reporting memory errors can cause severe performance degradation in systems suffering from hardware faults. We inject faults in DRAM on a real cloud server and quantify the single-machine performance degradation for both batch and interactive workloads. We observe that for SPEC CPU2006 benchmarks, memory errors can slow down average execution time by up to 2.5$\times$. For an interactive web-search workload, average query latency degrades by up to 2.3$\times$ for a light traffic load, and up to an extreme 3746$\times$ under peak load. To gain insight, we formulate analytical models that capture the theoretical performance of batch and interactive applications in systems experiencing correctable memory errors. We use these models to predict the effects of page retirement and changes to enable faster error handling. Our analyses of the memory error-reporting stack reveals architecture, firmware, and software opportunities to improve performance consistency by mitigating the worst-case behavior on faulty hardware.

Collaborators:

- Dr. Mohammed Shoaib, Microsoft Research
- Dr. Sriram Govindan, Microsoft

- Dr. Bikash Sharma, Microsoft
- Dr. Di Wang, Microsoft Research
- Prof. Puneet Gupta, UCLA

Source code and data are available at:

- `https://nanocad-lab.github.io/X-Mem/`
- `https://github.com/Microsoft/X-Mem`
- `https://github.com/nanocad-lab/X-Mem`
- `https://github.com/nanocad-lab?&q=performability`
- `http://nanocad.ee.ucla.edu`

5.1 Introduction

In datacenters, particularly clouds, failures of servers and their components are a common occurrence and can impact performance and availability for users [254]. Faults can manifest as correctable errors (CEs) that can degrade performance, detected but uncorrectable errors (DUEs) that can cause machine crashes, and undetected errors that often cause silent data corruption (SDC). At best, errors are a nuisance by increasing maintenance costs; at worst, they cause cascading failures in software micro-services, leading to major end-user service outages [255].

Hard faults in main memory DRAM are one of the biggest culprits behind server failures in the field [256–260], while main memory comprises a significant fraction of datacenter capital and operational expenses [261, 262]. Unfortunately, memory reliability is expected to decrease in future technologies [260] as a result of increasing manufacturing process variability in nanometer nodes [7, 75, 263, 264]. Meanwhile, researchers are actively exploring new approaches to designing memory for the datacenter, such as intentionally using less reliable DRAM chips to reduce provisioning costs [261, 262].

Thus far, the research community has not explored application performance on machines when errors actually occur. In past smaller-scale systems with older DRAM technology, this was not a major consideration because of the rarity of memory errors. In modern warehouse-scale computers (WSCs), however, the worst case for errors is no longer a rare case. While most servers have low error rates, in any given month, there are hundreds of servers in a datacenter that suffer from millions of correctable errors [260]. Understanding and addressing this issue is important: a recent study at Facebook found that correctable memory errors can lead to wildly unpredictable and degraded server performance [260], which is a primary challenge in cloud computing [201].

In this chapter, we experimentally characterize the performance of several applications on a real system that suffers from correctable memory errors. Our contributions are as follows:

- *We identify why and how memory errors can degrade application performance* (Sec. 5.3). When memory errors are corrected, they are reported to the system via hardware interrupts. These can cause high firmware and software performance overheads.

- *We quantify the extent of performance degradation caused by memory errors on a real Intel-based cloud server using a custom hardware fault-injection framework* (Sec. 5.4). Our measurements show that batch-type SPEC CPU2006 benchmarks suffer an average 2.5× degradation in execution time, while an interactive web-search application can experience up to 100× degradation in quality-of-service when just 4 memory errors are corrected per second using "firmware-first" error reporting.

- *We derive analytical models for the theoretical performance of batch (throughput-oriented) and interactive (latency-oriented) applications on systems in the presence of memory errors.* These can provide general insights for situations where empirical testing and fault injection are not feasible.

- *We predict the impact of page retirement, error masking, and firmware/software speedups on application performance using our analytical models.* We find that memory page retirement is a very effective and low-overhead technique for mitigating the impact of hard faults. Error masking can improve performance consistency in presence of errors, but necessarily reduces the sampling quality which is useful for page retirement and field studies on DRAM errors. A significant change to error-handling architectures may be required to enable more gradual and predictable performance degradation in presence of errors. Finally, the models allow engineers to reason about when to service a faulty machine by understanding the performance impact on the workload of interest.

This chapter is organized as follows. Sec. 5.2 introduces a few closely-related studies on memory errors from which we differentiate our work. We discuss background material on memory errors in Sec. 5.3. This motivates our proposition that error reporting can be a major cause of performance loss. We follow with a description of our experimental fault injection setup and analyze our empirical results for application performance in Sec. 5.4. We then derive a series of analytical models that capture theoretical application performance in presence of errors and use them to make projections in Secs. 5.5, 5.6, 5.7, and 5.8. We discuss our major insights and conclude the chapter in Sec. 5.10.

5.2 Related Work

There is a large body of prior work that address reliability in memory systems [265]. Many compelling reliability-aware techniques for energy savings in caches and memory have been proposed [1, 8, 161, 163, 168, 169, 266–268], but none of them have focused on large-scale systems. A recent thread of research studied memory failures in large-scale field studies that characterized broader trends [257–260, 269–271], but none of them has addressed the performance impact of memory errors. A recent work has proposed designing datacenter DRAM with heterogeneous reliability for cost reduction [261], but they did not consider the performance implications from increased error rates. Delgado et al. [272] were the first to experimentally expose the performance implications of Intel's System Management Mode (SMM), which is often used for memory error reporting (and which we discuss in this work). They observed inconsistent Linux kernel performance and reduced quality-of-service (QoS) from SMM on latency-sensitive user applications.

Researchers have generally considered the server and application performance overheads of DRAM fault tolerance schemes only in the case when no errors occur. Prior work, however, has not identified or measured the performance degradation caused by memory errors in systems *with* faulty hardware. This effect is important: datacenter operators have observed performance-degrading memory errors to occur routinely in the field, where they increase maintenance costs and reduce performance consistency. In this chapter, we address these gaps in prior work by describing the mechanisms by which performance degrades and empirically demonstrate that the performance degradation on a real cloud server can be severe.

5.3 Background: DRAM Error Management and Reporting

Error reporting, or logging, in firmware and/or software is required for datacenter operators to detect failures and service them appropriately. They also enable the numerous past [257–260, 269–271] and future field studies of DRAM errors. Page retirement, which has been recently shown to significantly reduce DUE occurrences at Facebook [260], also relies on accurate and precise error logging in order to identify failing pages. We believe that a primary cause of performance degradation from memory errors is the firmware/software stack, not the hardware fault tolerance mechanisms. Thus, we discuss how DRAM errors are made visible to software. Because Intel-based systems are dominant in the cloud, we focus on their Machine Check Architecture (MCA) [246], specifically for Haswell-EP (Xeon E5-v3) processors. Other platforms may have similar mechanisms, but they are beyond the scope of this work.

When the ECC circuits in the memory controller detect or correct an error, they write information about the error into special registers. This includes information like the type/severity of the error (CE or DUE) and possibly the physical address. The hardware then raises an interrupt to either the OS or firmware, but not both simultaneously; this option is specified statically at the system boot time [246, 273].

If the software interrupt mode is selected, the ECC hardware raises either a Corrected Machine Check Interrupt (CMCI) for a CE, or Machine Check Exception (MCE) for a DUE. For an MCE, the typical response in current cloud servers is to cause a kernel panic, which crashes and restarts the entire machine. (Higher-end machines, which are typically not deployed in public clouds, support poisoning, a technique that facilitates recovery attempts from uncorrectable errors instead of crashing.) For a CMCI, the kernel simply logs the DRAM error with a timestamp, which can then be read by user-level software through the system event log. On our system, a CMCI raised by the memory uncore is broadcast to all cores on the socket, but handled by just one thread (which may be statically assigned, as suggested by Intel [246], or as we believe in the case of our platform, dynamically load-balanced within a socket). With CMCI or MCE-based software error reporting, the OS kernel might know the physical address of the error, but generally not the precise location that the error occurred in the DRAM organization. This is because the mapping is complex and dependent on hardware and platform configurations. Thus, using purely software-mode interrupts on supported processors, the OS might use page retirement, although the datacenter operators would generally not know which memory module is failing on a machine that reports many errors. Note that in our platform used for the experiments, the kernel *does not* know the physical address nor the DRAM location of a memory error when using CMCI-based reporting.

Firmware-based error reporting, on the other hand, can determine both the physical address and the precise location of the memory error in DRAM by performing the required platform-specific calculations. This makes it our preferred boot-time option. If the firmware interrupt mode is selected in Intel machines with the Enhanced Machine Check Architecture (EMCA) [273], the ECC hardware raises a maximum-priority System Management Interrupt (SMI). In our platform, an SMI is broadcast to all logical processors across both sockets. All processors that receive SMIs immediately enter the special System Management Mode (SMM) in firmware. SMM raises each processor to the highest-possible machine privilege level. The job of SMM in response to a memory error is to read all relevant registers in the memory controller, compute additional information about the error, and report the error. Because SMM is not re-entrant, whenever a memory error occurs in firmware-first mode, the entire system becomes unresponsive. On our platform, when an SMI is being handled, all operating system and user threads are stalled: no forward progress on any system or application task can be made. Before exiting, SMM constructs an entry with detailed error information in the Advanced Configuration and Power Interface (ACPI) [274] tables and then forwards the error to the OS by raising the appropriate MCE or CMCI interrupt.

5.4 Measuring the Impact of Memory Errors on Performance

We experimentally characterized the impact of correctable memory errors to verify our claim that system-level error management and reporting is a primary cause of degraded application performance. Our hardware fault-injection methodology is described first, before we discuss the empirical results.

5.4.1 Experimental Methods

We measured the performance impact of memory CEs on a real Intel Haswell-EP-based cloud server running Windows Server 2012 R2. We expect the behavior of Linux-based machines to be similar, to what we find in this work, although we were not able to adapt our experimental framework and all workloads of interest to function on both platforms. We did not evaluate the relative performance of the reliability, availability, and serviceability (RAS) features in non-faulty situations because they are well understood and do not explain the denial-of-service effect [260] seen on machines with errors in the field. Instead, we measured the relative performance impact on the system when errors actually occurred, enabling us to quantify the impact of the complete hardware/firmware/software error reporting stack.

DRAM fault injection was physically performed using proprietary hardware and software tools and was controlled by OS and user-level software. The tools have the flexibility to flip specific bit cells with any desired pattern. Before starting the application under test, our framework uses the Windows kernel debugger [275] to perform virtual-to-physical address translation for a special region in the private virtual memory region. The tools are then used to inject a soft single-bit fault into DRAM hardware at a controlled time and location using the translated physical address. Demand and patrol scrubbing were disabled to prevent unintended removal of the injected fault. Note that in production systems, scrubbing is typically enabled; this might contribute additional performance overheads beyond those measured in this chapter in the presence of real hard faults.

Using a lightly-modified version of X-Mem, our open-source and extensible memory characterization and micro-benchmarking tool [11, 235], we sensitized a single faulty DRAM location using one thread at a user-controlled constant rate. To ensure every load to the faulty memory actually reached the DRAM (and not just the caches), we flushed the cache line after every access and used memory barriers to ensure only one access could be outstanding at a time. This fault-sensitizing approach is independent of application access behavior. In general, because the performance penalty incurred by interrupts is not *necessarily* “paid” by the aggressor thread that sensitizes the fault (except in the case of SMIs), the results only depend on the interrupt performance and the number of error-interrupts per second (as we show in Sec. 5.4.2, there was no measurable performance degradation caused by the *hardware* RAS technique in the presence of errors, such as SECDED vs. ChipKill). These facts make our experiments tractable to perform for different applications while yielding correct results for different DRAM fault models.

We deliberately swept a wide range of error-interrupt rates in our experiments to capture different scenarios. A rate in the range of 100-1000 error-causing interrupts per second, while seemingly extreme, might actually be common in production datacenters. This is due to a power-law distribution, where a few machines see many errors in a month [260]. However, existing data from the field does not provide sufficient time-resolution information to determine how bursty errors actually are compared to the relevant timescales of an application. For instance, a server that had one million reported errors in a month [260] might have had them uniformly over time (average 0.38 errors/sec) or as an avalanche during a single hour (average 277 errors/sec). We believe the latter type of scenario is more likely to occur in reality; a hard fault may begin to manifest in a frequently-accessed mechanism, such as a stuck I/O pin in the DDR channel interface. Moreover, there may actually be more errors in practice than those indicated in the logs used by field studies. This is because existing errors that are pending service from firmware/software may block the recording of others.

To validate the accuracy of our fault injection approach, we used a variety of faulty memory modules (DIMMs) with known fault patterns that spanned major DRAM manufacturers. The faulty DIMMs consisted of specimens that failed in a production datacenter setting and were characterized after the fact, and of specimens that had failed post-manufacturing screening tests and were graciously provided by each manufacturer for our research needs. We replicated several of the known failure patterns using our fault injection framework on known-good DIMMs. For both the injected and the ground-truth faulty memories, the system-level response was identical: the expected number of errors reported in the OS, and the performance impacts that we outline in the next subsection were identical. Therefore, all of our reported results use our fault injection framework as a valid substitute for real faulty DIMMs.

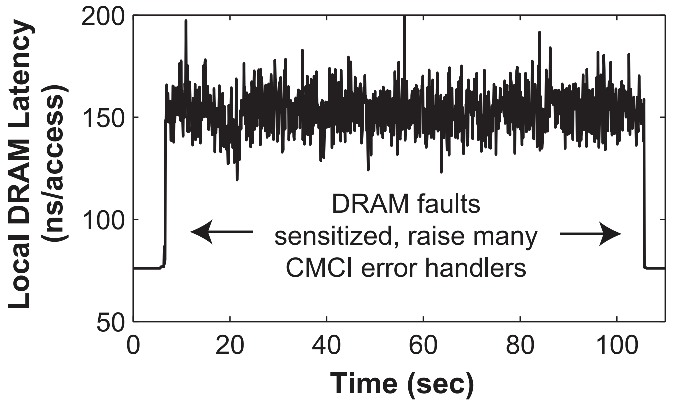
5.4.2 Empirical Results using Fault Injection

We first verified our hypothesis that the error reporting interface is a major culprit behind performance degradation on machines with memory errors. This was done by measuring raw memory performance as well as application-level performance with error interrupts enabled and disabled in the BIOS. When interrupts were disabled, we measured no degradation in performance incurred by sensitizing memory faults – even at very high rates – for each of the available hardware RAS techniques (SECDED, SDDC/ChipKill, rank sparing, channel mirroring, etc.). Conversely, the performance degradation when interrupts were enabled depended only on the fault sensitizing rate – regardless of the RAS scheme. This proved that the firmware/software overhead to report memory errors causes significant performance degradation, and warranted further analysis. (Note that forms of memory scrubbing may also cause additional performance degradation in presence of errors, but these were not evaluated due to experimental limitations in our fault injection framework.) We then characterized the latency of the error-reporting interrupts before examining their interference with batch (throughput-oriented) and interactive (latency-oriented) applications.

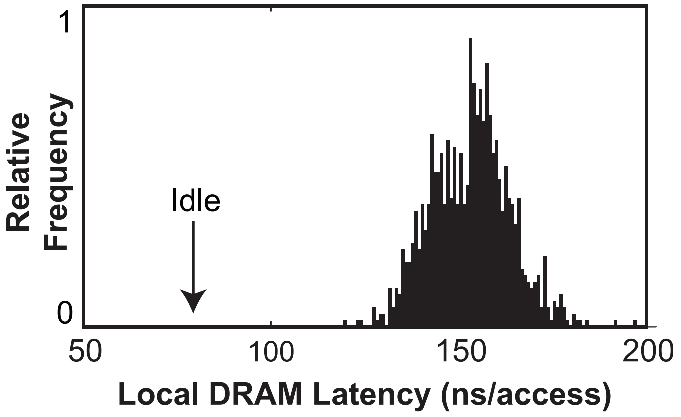
5.4.2.1 Error-Handling Interrupt Latencies

We measured interrupt latencies by accessing the faulty DRAM location – located on the same socket as the sensitizing thread – as fast as possible, causing interrupts that flood the whole socket and constantly pre-empt the sensitizing thread. The fault-sensitizing rate was thus limited by the interrupts, allowing us to measure the handler latency directly based on the completed number of memory accesses per second. We found that the SMI latency (133 ms) is up to 171$\times$ worse than the CMCI latency (775 μs). This is because the SMI invokes SMM, which must read all the relevant registers to reconstruct the error information, populate the ACPI tables, and then raise a CMCI to the OS before exiting and allowing threads to resume. The implementation of SMM impacts performance even more than the indicated latency by blocking all threads from executing. It also executes slowly due to the way it uses memory [246]. In contrast, CMCIs operate like a conventional interrupt, only pre-empting a single logical core and running in the OS context. Both of these error-reporting interrupts exhibit latencies that are high enough to cause significant application interference.

We studied the memory interference caused by CMCIs further. We measured the native DRAM latency over time using the standard version of our X-Mem tool [11,235], while our fault injection framework sensitized a DRAM fault to raise 1000 CMCI/sec. The trace and histogram of memory latency is shown in Fig. 5.1. The flood of CMCI handlers compete for bandwidth in the shared L3 cache and DRAM, adding Gaussian noise to the overall memory latency, which doubled on average. This could interfere with the performance of a victim application, even if it is isolated in a virtualized environment.



(a) Trace

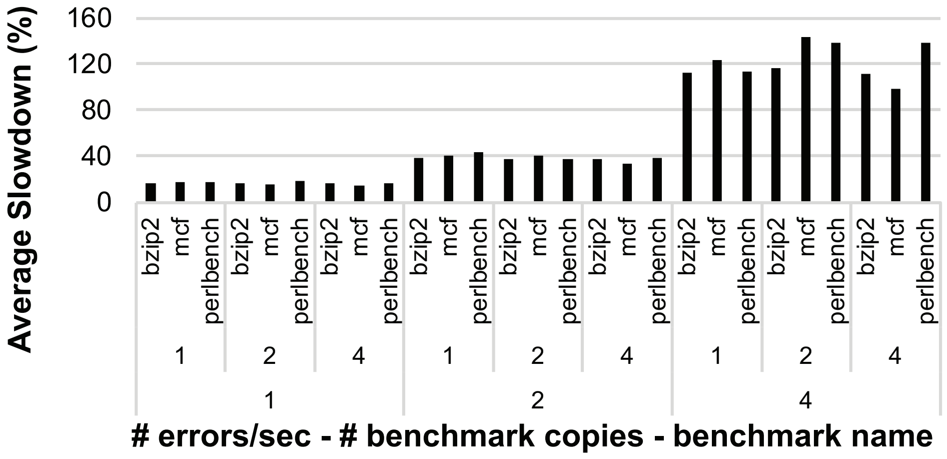


(b) Histogram

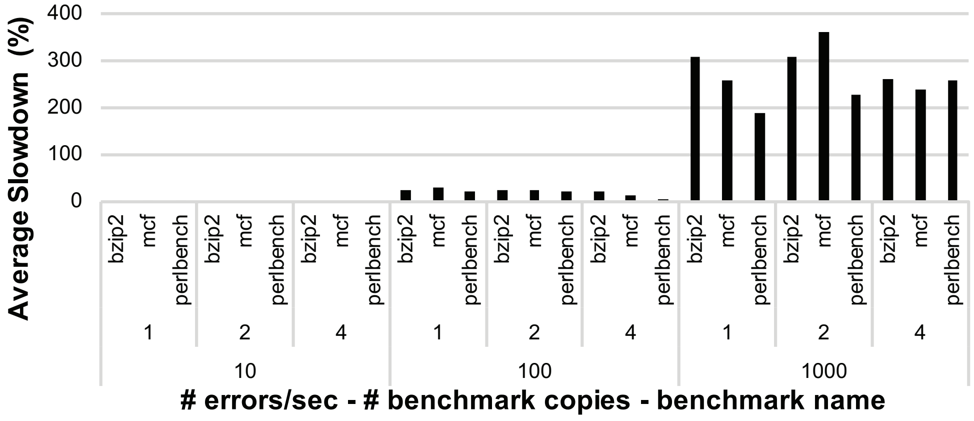
Figure 5.1: CMCI handlers can cause significant memory interference, degrading performance even if the application thread is not directly interrupted. A non-interrupted thread sees an increase of DRAM latency that has additive Gaussian noise. Here, 1000 CMCIs per second worsens average memory latency by about 2X by competing for memory-level parallelism and bandwidth in the L3 cache and DRAM.

5.4.2.2 Impact of Error Handlers on Batch Applications

Given the high interrupt-handling latencies that we measured, we characterized how much performance degradation memory errors can actually cause on real applications. First, we considered the performance of batch applications using three benchmarks from the SPEC CPU2006 suite: `bzip2`, `mcf`, and `perlbench`. The results are shown in Fig. 5.2a when the server is configured to report memory errors using the SMI interrupt, and Fig. 5.2b for the CMCI interrupt. For each benchmark, we varied the number of error-interrupts per second (outer axis labels) and the number of identical



(a) SMI Reporting



(b) CMCI Reporting

Figure 5.2: Empirical results for three different SPEC CPU2006 "batch" applications, show significant performance degradation in the presence of errors. Workloads were run to completion, and run-to-run variation was negligible.

copies for each benchmark that were run simultaneously on different cores (inner axis labels).

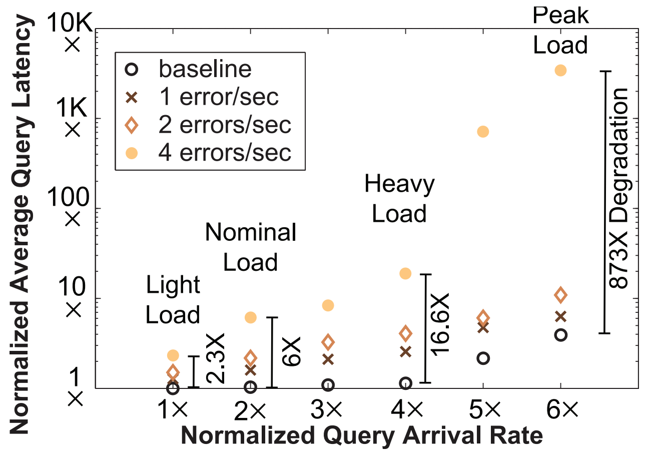
Performance penalties were significant for both types of interrupt. When using SMIs, the average slowdown was roughly 16% for all three benchmarks with just a single error-interrupt per second. With two SMIs per second, the penalty rose to approximately 36%, and with four SMIs per second, the average penalty was almost 115%. For the SMI cases, the applications behave as though they were duty-cycled at a rate of one minus the average utilization consumed by error handlers. For CMCI reporting, performance degradation is negligible for ten error-interrupts per second. At 1000 error-interrupts/sec with CMCIs, our batch applications perform approximately

200% to 350% worse.

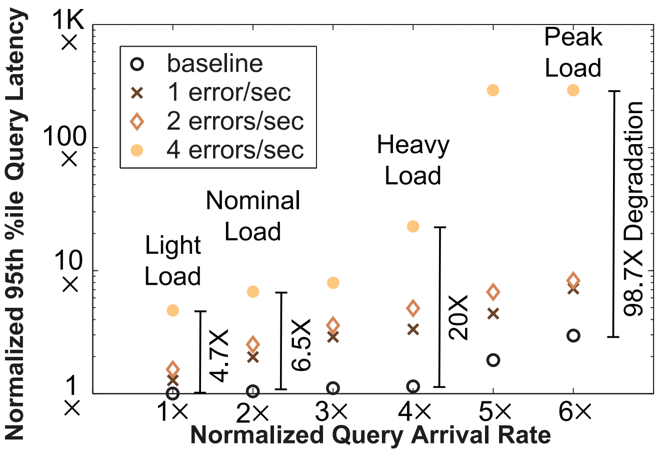
As the error rate increases, performance varies considerably within and across the three applications for both types of interrupts. We believe this is caused by two factors.

- As we noted earlier, high error rates cause increased memory interference; this affects each application differently. For example, in the SMI case, `bzip2` has very consistent performance no matter how many copies run, while `mcf` shows more variation because it is a memory-heavy workload. Given the frequent task pre-emption and possible cache pollution and/or flushes that are caused by SMM, multiple copies of `mcf` are more likely to interfere in main memory, causing additional performance degradation.
- In the case of CMCIs, each processor core may not receive a fair share of interrupts. Without knowing the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, only 80% of the interrupts that a thread running on another core receives.

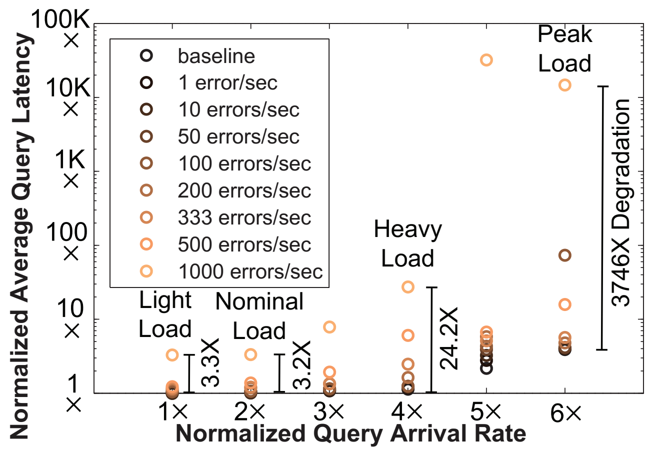
Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.



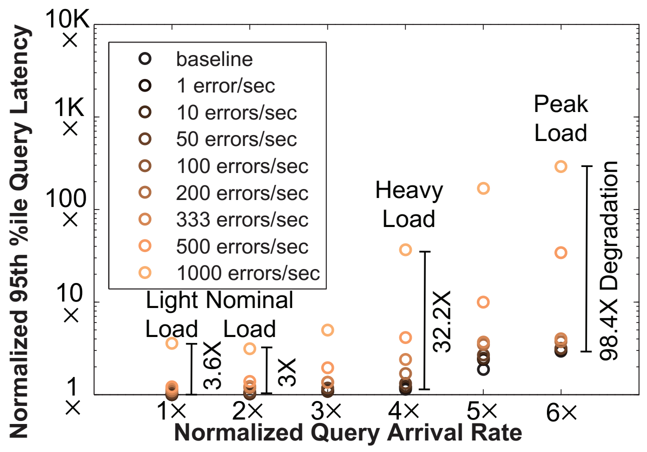
(a) SMI, Average



(b) SMI, 95th Percentile



(c) CMCI, Average



(d) CMCI, 95th Percentile

Figure 5.3: An industrial web-search application running on a state-of-the-art cloud server experiences severe performance degradation in the presence of memory errors. SMI interrupts (133 ms) degrade performance much faster than CMCIs (775 μs) because of their higher handling latencies.

5.4.2.3 Impact of Error Handlers on Interactive Web-Search Application

Finally, we consider an interactive web-search workload, which was developed internally. It emulates the index-searching component of a major production search engine using real web-search traces. Fig. 5.3 depicts the normalized average and 95^{th} percentile query latency as a function of the normalized query arrival rate and the number of DRAM error-interrupts per second. We find that even for light loads, error-interrupt rates of four SMIs per second can cause $2.3\times$ higher average query latency. Under peak search traffic, the introduction of just a few memory errors to the system causes the average query latency to increase by up to a staggering $873\times$ (Fig. 5.3a). We see similar trends in the tail latency (Fig. 5.3b). In contrast, CMCI-based error reporting does not result in significant performance penalties for up to 50 error-interrupts per second. CMCIs can still deny service completely, however, just like SMIs: up to $3746\times$ degradation can occur in average search latency under peak load with 1000 error-interrupts per second (Figs. 5.3c and 5.3d). Note that the relative degradation in tail latency is usually greater than the average for the lightly loaded cases, but less than the average for the heavily-loaded cases. This implies that error-interrupts can *completely* deny service in the worst case, which will cause the average latency to exceed the tail latency.

The interactive application is much more sensitive to errors than the batch applications because its performance metric (tail latency) is dependent on the worst-case interference of interrupt handlers with user event handlers, i.e., the timing of events. Conversely, the batch applications' performance metric (overall execution time) is linearly dependent on the number of interrupts and their total handling duration, but not the arrival times of the errors.

These results highlight the need for further modeling in order to develop compelling solutions to the problem of memory errors.

5.5 Analytical Modeling Assumptions

A set of analytical models are derived for the performance of batch and interactive applications under the impact of memory errors. Our formulated models are based on fundamental concepts from queuing and probability theories and our background knowledge described earlier. Although our experiments in the previous section used X-Mem as a small aggressor application to control the error rate seen by the victim workloads of interest, the overall effects of errors impacting applications does not depend on which task sensitizes memory faults. This is because the resulting SMI or CMCI interrupts can affect all applications (both aggressor and victim) equally. We rely on two major assumptions.

Assumption 1. *Tasks are considered as a simple binary state at any point in time: either making forward progress or not.* Background sub-tasks that execute asynchronously with respect to the CPU are assumed to pause while the error handler runs. This view abstracts details such as CPU utilization, instructions per cycle, etc., which is valid as long as the workload is not I/O-bound.

Assumption 2. *For a given interval of time when errors can occur, all errors arrive randomly and independently of each other.* Thus, the time t between occurrences is given by the exponential probability distribution with fixed average error rate λ [276].[1] We justify this assumption by considering a system that is concurrently executing many threads. The probability of any particular thread triggering a memory fault (leading to either a memory CE or DUE) at a particular instant is very small. For a system with enough cores and threads, error arrivals can be modeled as independent from each other because of the Poisson limit theorem, also known as the "law of rare events" [277], which is the same logic that is applied commonly in queuing theory [278]. This assumption may be less accurate in uniprocessor scenarios where only a single thread executes at a time, but without further information, we use the exponential distribution to make modeling tractable. Moreover, in a system with a failing coarse-grain memory component (e.g., rank failure), the faults are likely to be triggered randomly and often by any running thread.

We consider four basic scenarios in increasing levels of complexity: (Sec. 5.7.1) a uniprocessor system running a batch application, (5.7.2) multiprocessor/batch, (5.8.1) uniprocessor/interactive, and (5.8.2) multiprocessor/interactive. There are many other more complex scenarios such as mixed batch/interactive applications and heavily multi-programmed systems. For simplicity, we

[1]The exponential distribution is the only continuous-time probability distribution with fixed average error rate that is memoryless (independent inter-arrival times) [276]. That is, $p(t;\lambda) = \lambda e^{-\lambda t} U(t)$, where $U(t)$ is the unit step function.

do not currently consider these, which are part of our ongoing work. Each of the four scenarios relies on the notion of application thread servicing-time; we discuss this first in Sec. 5.6.

5.6 Model for Application Thread-Servicing Time (ATST)

A core concept in all of our queuing-theoretic models is the *application thread-servicing time* (ATST), which is a random variable that represents the total "wall-clock" execution time for an *application event handler* once it is issued to a processor. Once an application event has been issued, it will execute to completion without being rescheduled, except for error arrivals that preempt its execution. Application events must issue and complete in FCFS order. We derive a complete probability distribution for ATST before using the model to predict the theoretical impact of page retirement.

5.6.1 Model Derivation

Let t_{service} represent the ATST that is a function of the *characteristic parameters of the application event* $\vec{y}$ and the *random* number of faults K that occur while the event is being handled. Let $\gamma = \gamma(\vec{y})$ represent the *nominal event handling time* in the system without any faults. Each fault i, $1 \leq i \leq K$, has a corresponding tuple $\vec{x}_i$ that represents the *characteristic parameters of the fault event* that arrives at time t_i. Let $\tau_i = \tau(\vec{x}_i)$ be the *fault handling time for fault i*. The ATST is simply the nominal event handling time plus any extra time consumed by memory error handlers that preempt the application's processor usage by high-priority system interrupts. Then $t_{\text{service}}(K)$ is a function of the random number of errors that interfere K and is given by:

$$\boxed{t_{\text{service}}(K) := \gamma + \sum_{i=1}^{K} \tau_i.} \tag{5.1}$$

The probability distribution of $t_{\text{service}}(K)$ is identical to the distribution of K. Let $P_K(k) := \Pr\{K = k\}$ be the probability mass function (PMF) for K. K is not a Poisson random variable even though the underlying error arrival times follow the exponential distribution. This is because the window of time in which we count K errors is not fixed; instead, the window of time is actually $t_{\text{service}}(K)$. The number of errors that interfere with the application event depends on the total event latency, which in turn depends on the number of errors that occur because they increase overall latency via interference. This seemingly circular logic can be expressed using recursion and solved using dynamic programming.

To derive a formula for P_K, we break down the problem into pieces that are resolved using existing probability theory. $t_{\text{service}}(K)$ consists of one term for the nominal runtime γ and K terms for the error handling times τ_i. Each of these components has a known and fixed duration for which we can write down the PMF for the number of errors that may occur in that time. We build the formulation inductively as follows.

Consider the case where errors can only arrive when the application event handler is executing, i.e., an error is discarded if it arrives while the processor is idle or a previous error is already being handled (effectively $\lambda = 0$ for a short time). Because the nominal event handling time is known to be γ, we can normalize the average error arrival rate λ to be the *number of error arrivals in a time window of length* γ. Let this normalized error arrival rate be $\beta_0 := \lambda\gamma$. Let K_0 be the number of errors that arrive during a time-window of length γ. This is governed by the Poisson distribution with parameter β_0:

$$P_{K_0}(k) := \Pr\{K_0 = k\} = \frac{\beta_0^k e^{-\beta_0}}{k!}. \tag{5.2}$$

Next, consider a time window of length τ_1. Like before, we can normalize λ to be the *number of error arrivals in a time-window of length* τ_1, and call this β_1, i.e., $\beta_1 := \lambda\tau_1$. (Generally, we have $\beta_i := \lambda\tau_i$.) Then the number of errors K_1 that arrive during a window of size τ_1 is governed by the Poisson distribution with parameter β_1:

$$P_{K_1}(k) := \Pr\{K_1 = k\} = \frac{\beta_1^k e^{-\beta_1}}{k!}. \tag{5.3}$$

We can continue inductively for $i \geq 1$:

$$P_{K_i}(k) := \Pr\{K_i = k\} = \frac{\beta_i^k e^{-\beta_i}}{k!}. \tag{5.4}$$

These expressions for P_{K_i} give us enough information to derive the overall PMF P_K, where K errors arrive in the variable time-window of length $t_{\text{service}}(K)$. We again derive the expression inductively. Consider the case where no errors arrive during the application event handler; the probability is found through Eqn. 5.2 for $k = 0$ because we only need to consider the time-window of length γ:

$$P_K(0) = P_{K_0}(0) = e^{-\beta_0}. \tag{5.5}$$

Now consider the case when exactly *one error arrives* during the overall application event. The total event handler latency is $t_{\text{service}}(1) = \gamma + \tau_1$. There is only one way this can happen: a single error arrives while the application is running on the processor, i.e., a time-window of length γ, and no error arrives while the resulting error handler executes, i.e., a time-window of length τ_1. The probability of this happening is:

$$P_K(1) = P_{K_0}(1) \cdot P_{K_1}(0) = e^{-(\beta_0+\beta_1)}\beta_0. \tag{5.6}$$

More possibilities arise for $K = 2$, the case where *exactly two errors* interfere with the application. The ATST would be $t_{\text{service}}(2) = \gamma + \tau_1 + \tau_2$. There are exactly two ways this can happen. *(i)* One error arrives during the application runtime γ, a second error arrives during the first error handler τ_1, and no error arrives during the second error handler τ_2. *(ii)* Two errors arrive during γ, and no errors arrive during τ_1 or τ_2. All of these possibilities are described by the following equation:

$$\begin{aligned} P_K(2) &= P_{K_0}(1) \cdot P_{K_1}(1) \cdot P_{K_2}(0) + P_{K_0}(2) \cdot P_{K_1}(0) \cdot P_{K_2}(0) \\ &= e^{-(\beta_0+\beta_1+\beta_2)}(\beta_0\beta_1 + \frac{\beta_0^2}{2}). \end{aligned} \tag{5.7}$$

Matters get more complex for the case where $K = 3$, where *exactly three errors* interfere with the application. The event handler latency would be $t_{\text{service}}(3) = \gamma + \tau_1 + \tau_2 + \tau_3$. There are exactly five ways this can happen.

- Two errors arrive during γ, one error during τ_1, and no errors during τ_2 and τ_3.

- One error arrives during γ, two errors during τ_1, and no errors during τ_2 or τ_3.
- One error arrives during each of γ, τ_1, and τ_2. No errors arrive during τ_3.
- Three errors arrive during γ, and none during τ_1, τ_2, and τ_3.
- Two errors arrive during γ, none during τ_1, one during τ_2, and none during τ_3.

The resulting equation captures all of these possibilities:

$$\begin{aligned}P_K(3) =&P_{K_0}(2)\cdot P_{K_1}(1)\cdot P_{K_2}(0)\cdot P_{K_3}(0)\\ &+P_{K_0}(1)\cdot P_{K_1}(2)\cdot P_{K_2}(0)\cdot P_{K_3}(0)\\ &+P_{K_0}(1)\cdot P_{K_1}(1)\cdot P_{K_2}(1)\cdot P_{K_3}(0)\\ &+P_{K_0}(3)\cdot P_{K_1}(0)\cdot P_{K_2}(0)\cdot P_{K_3}(0)\\ &+P_{K_0}(2)\cdot P_{K_1}(0)\cdot P_{K_2}(1)\cdot P_{K_3}(0)\\ =&e^{-(\beta_0+\beta_1+\beta_2+\beta_3)}\cdot(\frac{\beta_0^2\beta_1}{2!}+\frac{\beta_0\beta_1^2}{2!}+\beta_0\beta_1\beta_2+\frac{\beta_0^3}{3!}+\frac{\beta_0^2\beta_2}{2!}).\end{aligned} \tag{5.8}$$

We can extend the pattern for arbitrary K. The resulting expression is:

$$\begin{aligned}&P_K(k) = e^{-\left(\sum_{n=0}^{k}\beta_n\right)}\\ &\cdot[\sum_{i_k=0}^{0}\frac{\beta_k^0}{0!}\cdot[\sum_{i_{k-1}=0}^{(1-i_k)}\frac{\beta_{k-1}^{(i_{k-1})}}{i_{k-1}!}\cdot[\sum_{i_{k-2}=0}^{(2-i_{k-1}-i_k)}\frac{\beta_{k-2}^{(i_{k-2})}}{i_{k-2}!}\\ &\cdots\left[\sum_{i_0=0}^{\left(k-\sum_{m=1}^{k}i_m\right)}\frac{\beta_0^{(i_0)}}{i_0!}\right]\cdots]]],\end{aligned} \tag{5.9}$$

for $k \geq 0$.

The above equation is obviously unwieldy for large k. We re-write it more compactly in the following recursive form:

$$\boxed{\begin{aligned} P_K(k) &= e^{-\left(\sum_{n=0}^{k} \beta_n\right)} \cdot f(k,0) \\ f(x,y) &= \begin{cases} \sum_{i=0}^{y} \left(\frac{\beta_x^i}{i!} \cdot f(x-1, y+1-i) \right), \text{ for } x \geq 0 \\ 1, \text{ for } x < 0 \text{ and } y \leq -x \\ 0, \text{ for } x < 0 \text{ and } y > -x. \end{cases} \end{aligned}} \tag{5.10}$$

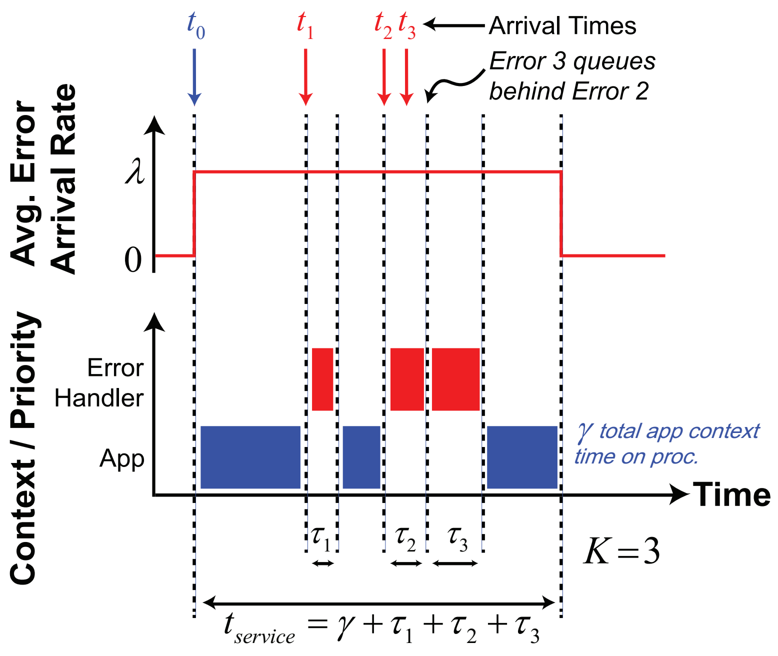
One can combine Eqns. 5.1 and 5.10 to obtain the complete probability distribution for the ATST.

Fig. 5.4 shows an example of how high-priority memory errors (red) can interfere with the execution of an application event (blue) for $K = 3$. There are two basic scenarios supported by Eqn. 5.1: either errors are allowed to arrive and queue while a previous error is being handled (Fig. 5.4a), or they are dropped (Fig. 5.4b. In the latter case, all $\beta_i = 0$ for $i \geq 1$, while the error handling times τ_i are unaffected; then Eqn. 5.10 reduces to the simple Poisson distribution ($P_K(k) = P_{K_0}(k)$). In either case, the total ATST (t_{service}) depends only on the total number of faults K that arrive while the application is executing, not on their arrival times or ordering.

5.6.2 Numeric Computation of ATST

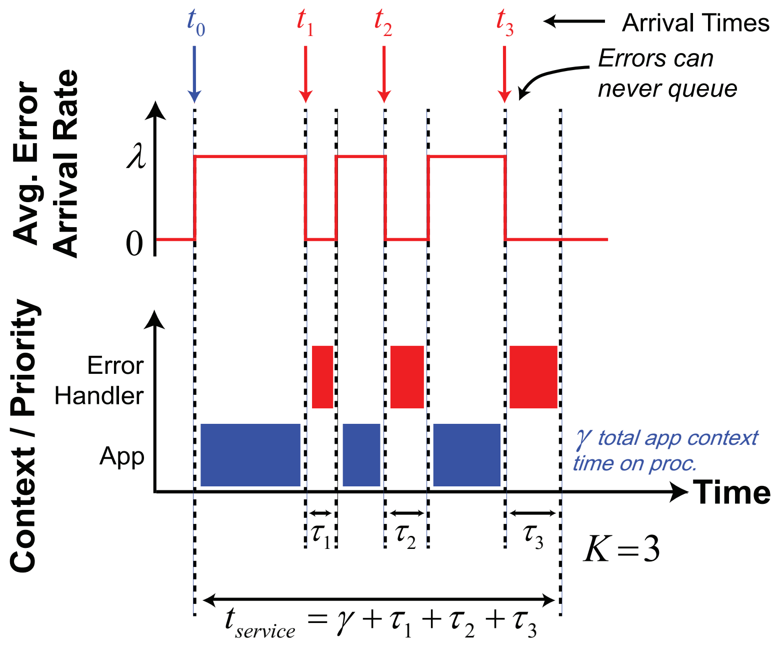
The recursive representation for ATST in Eqn. 5.10 is computationally intractable for large k. Moreover, $P_K(k)$ will asymoptotically approach 0 if the exponent term does not converge, i.e. the mean value of β_n for $1 \leq n \leq \infty$ is ≥ 1. In such a case, the expected value of the event latency ($t_{\text{service,avg}}$) does not exist, and on average, the machine will process error handlers forever: the event will never complete.

Fortunately, we can compute Eqn. 5.10 numerically using a dynamic programming approach for a finite maximum k of interest k_{max}. Construct a matrix $F \in \Re^{k_{max} \times k_{max}}$ containing scalar elements $f_{x,y}$, where each will correspond to $f(x,y)$ from Eqn. 5.10. We compute $f_{x,y}$ elements as



t_0 t_1 t_2 t_3
Arrival Times
Error 3 queues behind Error 2
Avg. Error Arrival Rate
λ
0
Context / Priority
Error Handler
App
γ total app context time on proc.
Time
τ_1 τ_2 τ_3
$K=3$
$t_{service} = \gamma + \tau_1 + \tau_2 + \tau_3$

(a) Errors are allowed to arrive and queue during error handlers



(b) Error arrivals during error handlers are discarded

Figure 5.4: Error handlers (SMI or CMCI) that interfere with an application executing on a processor change the application thread-servicing time (ATST), represented by t_{service}.

Algorithm 4 Dynamic programming approach to compute $f(x,y)$ from Eqn. 5.10.

```
//Input: $k_{max}$, $\beta_i$ for $0 \le i \le k_{max}$
//Output: Matrix $F \in \Re^{(k_{max}+1)\times(k_{max}+1)}$, where elements $f_{x,y}$ correspond to $f(x,y)$ in Eqn. 5.10
for $0 \le x,y \le k_{max}$
Initialize $F$ with all elements set to NaN
for $x = 0 : k_{max}$ do
    for $y = 0 : k_{max} - x$ do
        for $i = 0 : y$ do $u \leftarrow \beta_x^i / i!$
            if $x < 1$ then
                if $y + 1 - i \le -(x-1)$ then
                    $v \leftarrow 1$
                else
                    $v \leftarrow 0$
                end if
            else
                $v \leftarrow f_{x-1,y+1-i}$
            end if
            $w_i \leftarrow u \cdot v$
        end for
        $f_{x,y} \leftarrow \sum_{i=0}^{y} w_i$
    end for
end for
```

follows. Compute rows one at a time, starting from row 0 ($x = 0$). For the current row x, compute $f_{x,y}$ in parallel for $0 \le y \le k_{max} - x$. These parallel computations will require data only from elements $0 \le i \le y$ in row $x - 1$. When completed, find P_K by substituting $f_{x,y}$ for $f(x,y)$ in Eqn. 5.10. This procedure is specified formally in Alg. 4.

If we assume that the fault handling time is constant, i.e., $\tau_1 = \tau_2 = ... = \tau_k = \tau$, then also $\beta_1 = \beta_2 = ... = \beta_k = \beta$, because we defined $\beta_i := \lambda\tau_i$. In this special case, Eqn. 5.10 can be greatly simplified to the following expression:

$$\begin{aligned}\Pr\{K = k\} &= P_K(k) \\ &= e^{-(\beta_0 + k\beta)} \left(\sum_{i=1}^{k} \binom{k-1}{i-1} \frac{\beta_0^i \beta^{(k-i)}}{i!(k-i)!} \right) \text{ for } k \ge 0. \qquad (5.11)\end{aligned}$$

5.6.3 Average ATST

The *expected (mean) event handler latency* $t_{\text{service,avg}}$ is given by:

$$\boxed{\begin{aligned} t_{\text{service,avg}} &:= E[t_{\text{service}}(K)] \\ &= \sum_{k=0}^{\infty} P_K(k) \cdot t_{\text{service}}(k) \\ &= \sum_{k=0}^{\infty} P_K(k) \cdot \left(\gamma + \sum_{i=1}^{k} \tau_i \right). \end{aligned}} \tag{5.12}$$

We can also write $t_{\text{service,avg}}$ as follows. Let N be the expected number of errors that arrive during the event handler with time-window of length $t_{\text{service}}(K)$. It must be that $N := \lambda t_{\text{service,avg}}$. Then

$$\begin{aligned} t_{\text{service,avg}} &= \gamma + \sum_{i=1}^{N} \tau_i \\ &= \gamma + \sum_{i=1}^{\lambda t_{\text{service,avg}}} \tau_i. \end{aligned} \tag{5.13}$$

In general, we cannot solve this equation for $t_{\text{service,avg}}$. However, if we know that the error handling time is constant, i.e., $\tau_i = \tau$ for $1 \leq i \leq \infty$, then Eqn. 5.13 reduces to:

$$t_{\text{service,avg}} = \gamma + \lambda \tau \cdot t_{\text{service,avg}}. \tag{5.14}$$

Solving for $t_{\text{service,avg}}$, we get for constant τ:

$$\boxed{t_{\text{service,avg}} = \begin{cases} \frac{\gamma}{1-\lambda\tau}, \text{ for } \lambda\tau < 1, \\ \infty, \text{ for } \lambda\tau \geq 1. \end{cases}} \tag{5.15}$$

5.6.4 ATST Slowdown

We now define *absolute event slowdown* as a random variable that captures the difference between the actual event handling time and the nominal event handler time:

$$\mathrm{Slowdown}_{\mathrm{absolute}}(K) := t_{\mathrm{service}}(K) - \gamma \tag{5.16}$$

$$\begin{aligned}\mathrm{Slowdown}_{\mathrm{percent}}(K) &:= \frac{\mathrm{Slowdown}_{\mathrm{absolute}}(K)}{\gamma} \cdot 100\% \\ &= \frac{t_{\mathrm{service}}(K) - \gamma}{\gamma} \cdot 100\%.\end{aligned} \tag{5.17}$$

Then the *expected (mean) absolute event slowdown* is given by:

$$\begin{aligned}\mathrm{Slowdown}_{\mathrm{absolute,avg}} &:= E[t_{\mathrm{service}}(K) - \gamma] \\ &= E[t_{\mathrm{service}}(K)] - \gamma \\ &= t_{\mathrm{service,avg}} - \gamma\end{aligned} \tag{5.18}$$

$$\begin{aligned}\mathrm{Slowdown}_{\mathrm{percent,avg}} &:= E[\frac{t_{\mathrm{service}(K)} - \gamma}{\gamma} \cdot 100\%] \\ &= \frac{E[t_{\mathrm{service}}(K)] - \gamma}{\gamma} \cdot 100\% \\ &= \frac{t_{\mathrm{service,avg}} - \gamma}{\gamma} \cdot 100\%.\end{aligned} \tag{5.19}$$

If τ_i are all the constant τ, we have

$$\mathrm{Slowdown}_{\mathrm{absolute,avg}} = \begin{cases} \frac{\gamma}{1-\lambda\tau}, & \text{for } \lambda\tau < 1, \\ \infty, & \text{for } \lambda\tau \geq 1. \end{cases} \tag{5.20}$$

$$\begin{aligned}\mathrm{Slowdown}_{\mathrm{percent,avg}} &= \frac{\frac{\gamma}{1-\lambda\tau} - \gamma}{\gamma} \cdot 100\% \\ &= \begin{cases} (\frac{1}{1-\lambda\tau} - 1) \cdot 100\%, & \text{for } \lambda\tau < 1, \\ \infty, & \text{for } \lambda\tau \geq 1. \end{cases}\end{aligned} \tag{5.21}$$

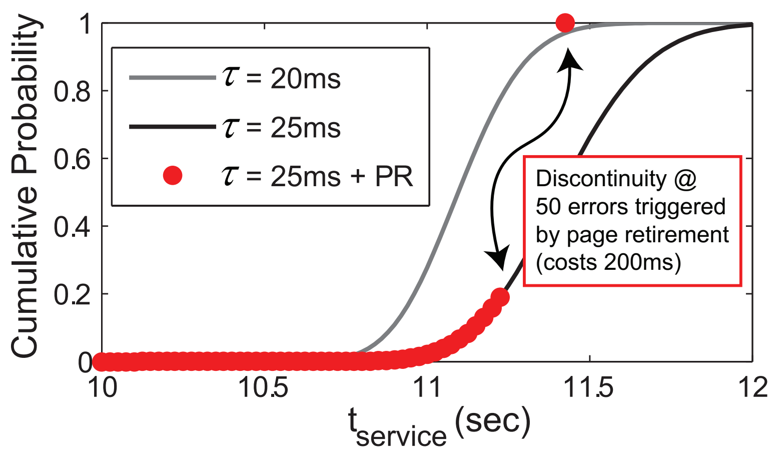


Figure 5.5: Page retirement can bound the worst-case ATST in presence of errors. Here, we assume error-handling times are all constant, $\lambda = 5$/sec, $\gamma = 10$ sec, and the page retirement operation, when invoked, stalls the application for 200 ms.

5.6.5 Using the ATST Model to Predict the Impact of Page Retirement on Application Performance

A convenient feature of our general ATST model (Eqns. 5.1 and 5.10) is the ability to predict the impact of dynamic/variable error-handling latencies, such as page retirement policies. We use the derived ATST model to consider the theoretical impact of page retirement on the performance of a hypothetical application task that is being serviced by a processor.

Fig. 5.5 depicts the cumulative probability distribution (CDF) of the ATST (t_{service}) when the average error rate $\lambda = 5$ per second, the application event handler nominally runs for $\gamma = 10$ seconds, and page retirement, when invoked, stalls the application for 200 ms.

If the error-handling time is constant $\tau = 25$ ms, then retiring the faulty page after exactly 50 errors are corrected will bound the ATST to just under 11.5 seconds, because no further faults will be sensitized after retirement. This results in a discontinuity in the depicted CDF. The expected t_{service} value with page retirement is reduced. Page retirement roughly bounds the worst-case ATST to the case where $\tau = 20$ ms (e.g., equivalent to a 20% speedup in the error-handling routine).

If the latency of page retirement is similar to or less than that of the error handler, it should always be done as soon as possible (e.g., when the high-overhead SMI-based reporting is used). If this is not the case, for very short batch applications or event handlers we believe that page retirement should be done in a “lazy” fashion by waiting until the processor is idle; we leave this consideration for future work. Conversely, for long-running batch applications or long-latency interactive event handlers, the sooner page retirement is used, the tighter the bound on ATST will be. In most cases, the impact to overall memory capacity from page retirement is likely negligible, and the probability of hard CEs morphing into DUEs should be greatly reduced.

5.7 ATST-based Analytical Models for Batch Applications

We derive analytical models for the performance of batch applications on both uniprocessors and multiprocessors for both SMI and CMCI-type error-reporting interrupts. The type of interrupt (SMI or CMCI) used to indicate memory errors is important because their pre-emptive behaviors are different. For the uniprocessor scenario (Sec. 5.7.1), the model is agnostic to which type of interrupt is used to report memory errors because both SMI and CMCI will pre-empt the single running application task with high priority. The multiprocessor scenario (Sec. 5.7.2), however, is sub-divided by interrupt type: SMIs pre-empt all processors equally while CMCIs only need to run on a single processor.

5.7.1 Uniprocessor System

A single task is issued to a uniprocessor system at time t_0 with input parameters $\vec{y}_0$. The goal is to predict the total execution time $t_{\text{runtime,uni}}$ required to complete the task in the presence of computational interference caused by a random number of K memory errors. The *nominal runtime* is a deterministic function of the input parameters $\gamma = \gamma(\vec{y}_0)$. We assume that there are no other applications running on the system except a very small memory fault-triggering *aggressor task* that consumes virtually no processor resources.

The aggressor triggers memory errors randomly according to the exponential distribution. The memory errors raise interrupts to the system, which causes the application task to become a *victim* of pre-emption by the error handlers. The ith memory error arrives at time t_i and has parameters $\vec{x}_i$ and deterministic handling time $\tau_i = \tau(\vec{x}_i)$ for $i \geq 1$. Because we care only about the total execution

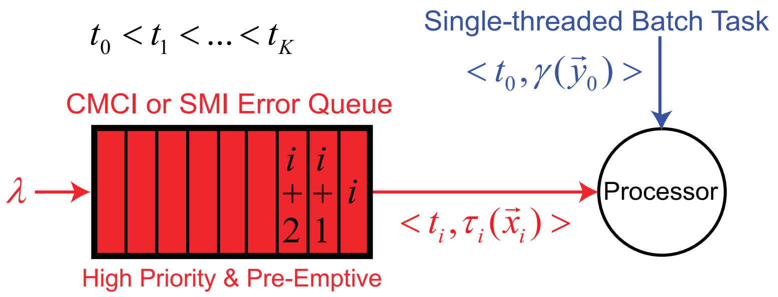

$t_0 < t_1 < ... < t_K$
Single-threaded Batch Task
$< t_0, \gamma(\vec{y}_0) >$
CMCI or SMI Error Queue
λ
$i+2$
$i+1$
i
$< t_i, \tau_i(\vec{x}_i) >$
Processor
High Priority & Pre-Emptive

Figure 5.6: Queuing model for batch applications on uniprocessors, with either CMCI or SMI error interrupts. The batch task (blue) begins executing before errors arrive and pre-empt it (red), increasing its overall execution time.

time for the application task, errors are assumed to never arrive before the task begins executing at time t_0, i.e., $t_0 < t_1 < ... < t_K$. If an error arrives while another is already being handled, it is queued and issued in first-come first-served (FCFS) order. The application victim task only makes progress when there are no errors currently being handled or in the queue.

The queuing model that captures this scenario is depicted in Fig. 5.6. Note that we are not concerned about the queuing statistics of the errors (red), but rather the performance of the application (blue). The overall runtime and the average metric are given by the following.

$$\boxed{\begin{aligned} t_{\text{runtime,uni}}(K) &:= t_{\text{service}}(K) \quad (5.22)\\ t_{\text{runtime,uni,avg}} &:= t_{\text{service,avg}}. \quad (5.23)\end{aligned}}$$

5.7.2 Multiprocessor System

In this scenario, a single task is issued to a multicore system with S processors at time t_0 with input parameters $\vec{y}_0$. The task has $1 \leq M \leq S$ worker threads that perfectly partition the workload at all times and begin executing at the same time. There are two derivative scenarios depending on whether memory errors are exposed through SMI or CMCI interrupts.

SMI Error Interrupts. This case is essentially the same as the uniprocessor/batch case described in the previous subsection. SMI-based error handlers pre-empt all tasks equally and occupy all processor resources uniformly, so the number of application threads and processor cores is not important for estimating the increase in runtime (we saw this effect empirically in Fig. 5.2a). The

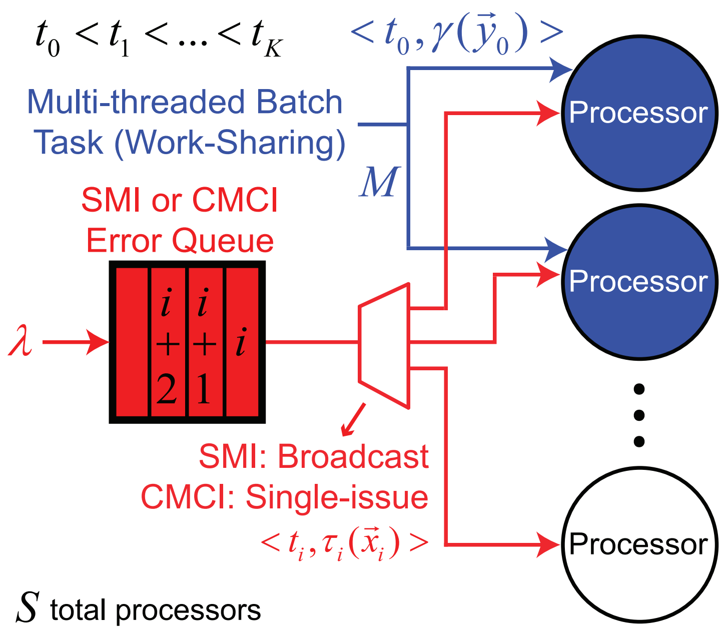


Figure 5.7: Queuing model for batch applications on multiprocessors, with either CMCI or SMI error interrupts. The batch task (blue) begins executing before errors arrive and pre-empt it (red), increasing its overall execution time.

queuing model is depicted in Fig. 5.7. Again, we are not concerned with the queuing statistics of the errors (red), just the performance of the application (blue). The performance metrics are

$$\boxed{\begin{aligned} t_{\text{runtime,multi,SMI}}(K) &:= t_{\text{service}}(K) \quad (5.24) \\ t_{\text{runtime,multi,SMI,avg}} &:= t_{\text{service,avg}}. \quad (5.25) \end{aligned}}$$

CMCI Error Interrupts. Only one core needs to run each CMCI handler invoked by a DRAM error. Unlike the SMI case, the arrival of an error does not block all forward progress of the application. To accurately model this situation, we would need to know a dispatch policy for CMCIs as well as the application inter-thread dependencies and communication patterns. Unfortunately, this situation is complex enough to require detailed system simulations, which defeats the purpose of our modeling exercise (especially considering that we already collected some empirical data through an un-scalable fault injection approach). However, if we only concern ourselves with the average-case behavior, we can analytically model this scenario.

Assume that the error-handling times τ_i converge to an average value τ, i.e., $\lim_{K\to\infty} 1/K \cdot$

$\Sigma_{i=1}^{K}\tau_i = \tau$. Then the *average utilization by errors in isolation* (without any application workload) is given by $\rho = \lambda\tau$, where $0 \leq \rho \leq S$ must hold. Let U be the *average utilization of core-level performance by the application in isolation*, where $0 \leq U \leq 1$ also must hold. Then

$$\boxed{t_{\text{runtime,batch,multi,CMCI,avg}} = \gamma \cdot \frac{UM}{UM - I}.} \tag{5.26}$$

Let I represent the *interference utilization*. If CMCI-based error handlers are issued using an idle core-first policy, then we have

$$\boxed{I_{\text{idle-first}} = \begin{cases} 0, & \text{for } \lambda\tau \leq S - UM, \\ \lambda\tau - (S - UM), & \text{otherwise.} \end{cases}} \tag{5.27}$$

If CMCI-based error handlers are issued using a uniform-random policy, then we have

$$\boxed{I_{\text{random}} := \frac{UM\lambda\tau}{S}.} \tag{5.28}$$

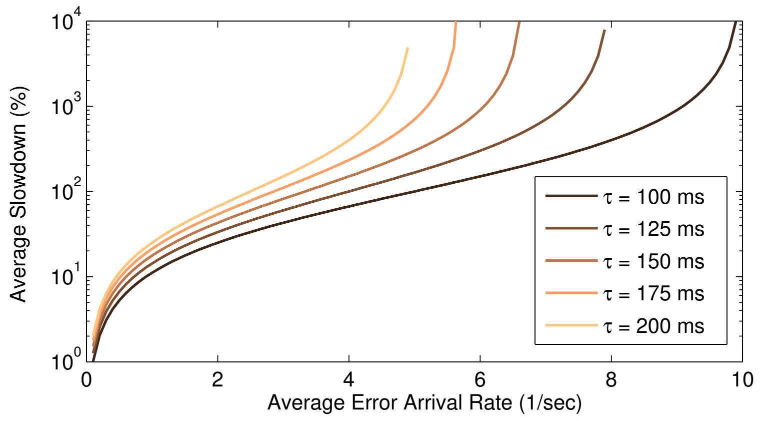
This means that for uniform-random CMCI errors, Eqn. 5.26 reduces to a similar result to the uniprocessor/batch case in the previous subsection, except where the error arrival rate as seen by any core running an application thread is reduced by the factor S because the whole multiprocessor shares the error-handling burden. We have

$$\boxed{t_{\text{runtime,batch,multi,CMCI,random,avg}} = \frac{\gamma}{1 - \lambda\tau/S}.} \tag{5.29}$$

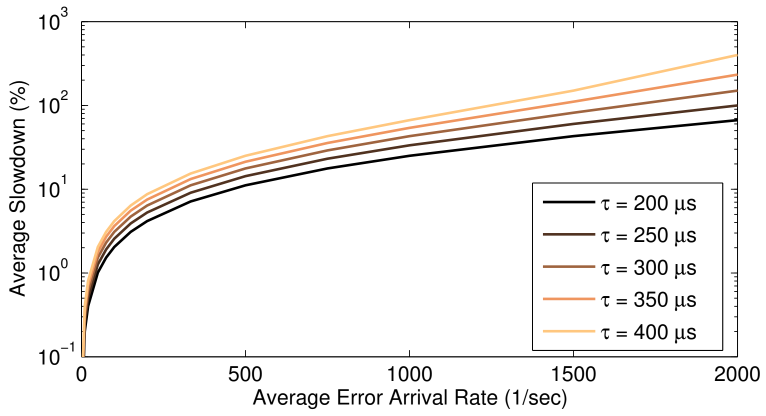
The queuing model for which we have formulated the average-case behavior is depicted in Fig. 5.7.

5.7.3 Predicting the Performance of Batch Applications

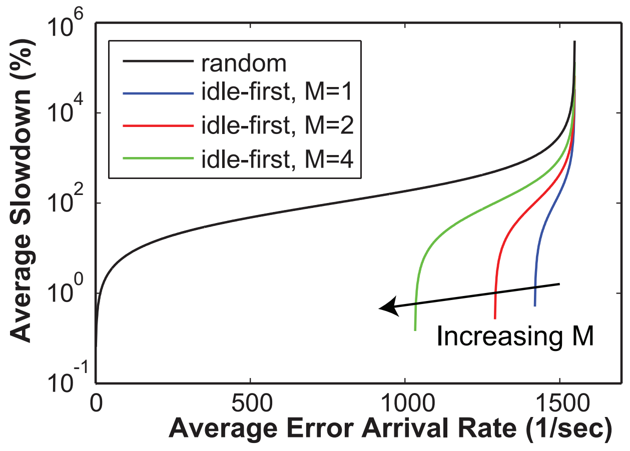
We use our derived analytical models to predict the performance of batch applications for different system scenarios in presence of memory errors. The results are shown in Fig. 5.8. One can see in Figs. 5.8a and 5.8b that for equivalent application slowdown on a uniprocessor system (say, 10%),



(a) Batch, Uniprocessor or Multiprocessor, SMI



(b) Batch/Uniprocessor/CMCI



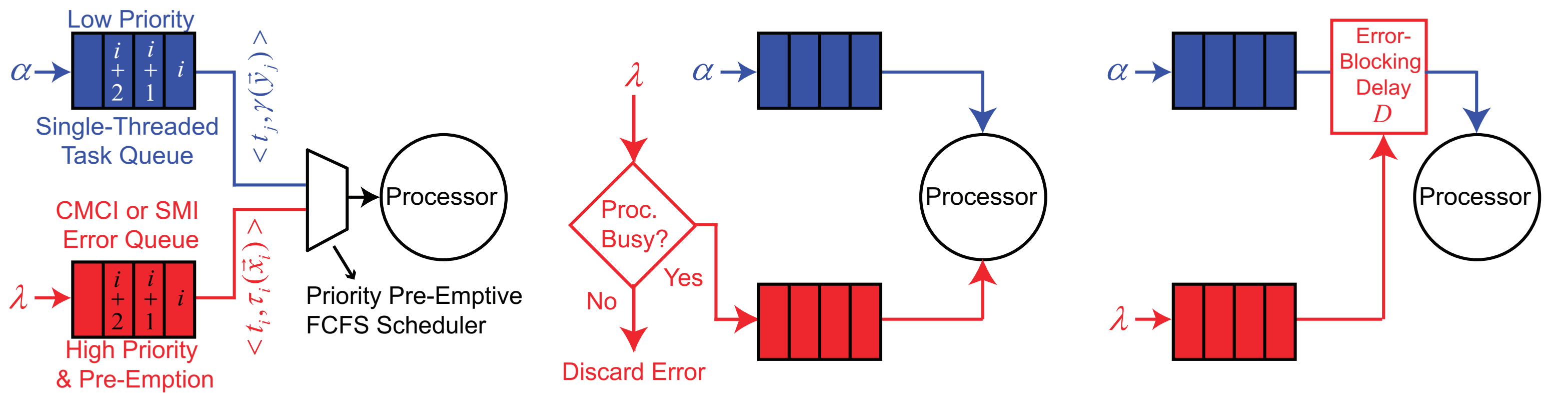
(c) Batch/Multiprocessor/CMCI with $\tau = 775\mu s$

Figure 5.8: Predicted performance of batch applications executing in presence of memory errors.

the system can tolerate hundreds of errors more per second using CMCI instead of SMI reporting if we assume SMIs are around 200 ms, while CMCIs are around 200 μs. For sufficiently high error rates, the system will saturate with errors, causing application slowdown to approach infinity. Fig. 5.8b illustrates the effect of CMCI issuing policy on a multiprocessor system. As expected, the performance degradation is worst when interrupts are distributed evenly across cores, compared with idle-first policies, which only disturb the application after idle cores are used to handle errors first. The results for the batch/multiprocessor SMI (Fig. 5.8a with $\tau = 133$ ms) and CMCI cases (Fig. 5.8b with random-issue CMCIs) shown above agree closely with our empirical measurements on SPEC CPU2006 benchmarks that were shown earlier by Fig. 5.2 in Sec. 5.4.

5.8 ATST-based Analytical Models for Interactive Applications

We derive approximate analytical models for the performance of interactive applications on uniprocessors for both SMI and CMCI-type error-reporting interrupts. Then, we describe the open problem of interactive/multiprocessor systems before briefly presenting some theoretical results for the uniprocessor case.



(a) Accurate (b) Approximation 1 (High App. Util.) (c) Approximation 2 (High Err. Util.)

Figure 5.9: Queuing models for an interactive application running on a uniprocessor with either CMCI or SMI error interrupts. In Approximation 1 (b), $\alpha\gamma >> \lambda\tau$, and in Approximation 2 (c), $\alpha\gamma << \lambda\tau$.

5.8.1 Uniprocessor System

In this scenario, many application events (tasks) arrive to a single-core system at times t_j with input parameters $\vec{y}_j$. They occur according to the exponential probability distribution with average arrival rate α. Upon arrival, if the processor is busy, the event enters a FCFS queue. At the same time, memory errors are still arriving at the system with average rate λ using either SMI or CMCI interrupts. If the processor is idle, the error handler executes immediately. If the processor is handling an application event, the error handler immediately pre-empts the processor. Finally, if the processor is busy handling a previous error, the newly-arrived error enters a separate high-priority FCFS queue. The scenario is depicted in Fig. 5.9(a). The goal is to predict the distribution of event-handling latency $t_{latency,uni}$ in the presence of memory error-handling interference.

This queuing model is complex. The processor is given work according to a priority pre-emptive FCFS scheduler, so the precise arrival times of events t_j and errors t_i can have a significant impact. The precise execution times of events $\gamma_j = \gamma(\vec{y}_j)$ and error handlers $\tau_i = \tau(\vec{x}_i)$ also have an effect. Thus, we analyze the system with two different approximations that simplify matters considerably: *(i) high application utilization*, and *(ii) high error utilization.*

5.8.1.1 Approximation 1: High Application Utilization

Here, we assume that at any given moment, the processor is far more likely to be handling an application event than it is to be handling an error. Let the average nominal event handling time be γ and the average error handling time be τ. Let $\phi := \alpha\gamma$ be the average utilization of the processor by application event handlers in isolation (no errors ever arrive). Similarly, let $\rho := \lambda\tau$ be the average utilization of the processor by error handlers in isolation (no application events ever arrive). Then the assumption here is that $\phi >> \rho$, where $0 \leq \phi, \rho \leq 1$.

In such a case, we approximate the queuing model – shown in Fig. 5.9(b) – by discarding any errors that arrive while the processor is idle. If the processor is busy handling an application event, the error is immediately handled via pre-emption of the processor. If the processor is busy with a previous error, the newly-arrived error enters its high-priority FCFS queue. We expect that this approximation would not introduce significant error on the distribution of $t_{\text{latency,uni}}$ because the probability of the processor being idle and an error discarded is small.

The benefit of this approximation is that we have obtained an *M/G/1* queuing model for the application events, which has some known properties. Errors now only take effect through the ATST, for which we already have derived the complete probability distribution. In theory, we could obtain the complete waiting time distribution for application events [279], where $\psi = \alpha t_{\text{service,avg}}$ is the average overall processor utilization, and $g(s)$ is the Laplace transform of $t_{\text{service}}(K)$:

$$W^*(s) = \frac{(1-\psi)sg(s)}{s-\alpha(1-g(s))}. \tag{5.30}$$

But $t_{\text{service}}(K)$ is a discrete random variable unless $\gamma(\vec{y})$ were a continuous random variable (but we assumed it was deterministic in our formulation of ATST). Then since $\gamma(\vec{y})$ is deterministic, $g(s)$ is not defined. Therefore, instead of deriving the complete waiting time distribution, Instead, we express the average-case behavior as follows [280]:

$$\boxed{\begin{aligned} t_{\text{wait,uni,approx1,avg}} &= \frac{\psi + \frac{\alpha}{t_{\text{service,avg}}}\text{Var}(t_{\text{service}}(K))}{2(\frac{1}{t_{\text{service,avg}}} - \alpha)} && (5.31)\\ t_{\text{lat,uni,approx1,avg}} &= t_{\text{wait,uni,approx1,avg}} + t_{\text{service,avg}}. && (5.32)\end{aligned}}$$

5.8.1.2 Approximation 2: High Error Utilization

Now suppose the opposite of Approximation 1, i.e., the average processor utilization from errors in isolation is much greater than the utilization caused by application events in isolation ($\phi << \rho$). This implies that the probability of any application event arriving to an idle system or one already handling a prior event is close to zero. Instead, it is highly likely that when an application event arrives to the system in Fig. 5.9(a), no other events are present, and the processor is handling errors when it would otherwise have been idle and ready to process the application event. Because errors enjoy higher execution priority than application events, this would force the event to experience a *blocking delay* until the error queue is depleted.

We approximate this scenario using the queuing model shown in Fig. 5.9(c). Let D represent the blocking delay experienced by a newly-arrived application event that is caused by a *busy period* consisting of prior errors that arrived to the system when it was idle. Assume that all error handlers have a constant processing latency, i.e., $\tau_i = \tau$. Let C be a random variable that represents the total number of errors in a given busy error-handling period. Then the blocking delay is a random variable as well and is given by $D = C\tau$.

Because errors are likely to arrive at an idle system or one already handling prior errors, we can model error behavior as the well-known *M/D/1* queue. Then C is governed by the Borel distribution [281]. Let $\rho = \lambda\tau$ once again be the average processor utilization caused by error arrivals in isolation ($0 \leq \rho \leq 1$). The PMF for D or C is thus given by:

$$\boxed{P_D = P_C := \Pr\{C = c\} = \frac{e^{-\rho c}(\rho c)^{c-1}}{c!}.} \tag{5.33}$$

The total latency for a given application event is simply the sum of the blocking delay and the servicing delay (ATST):

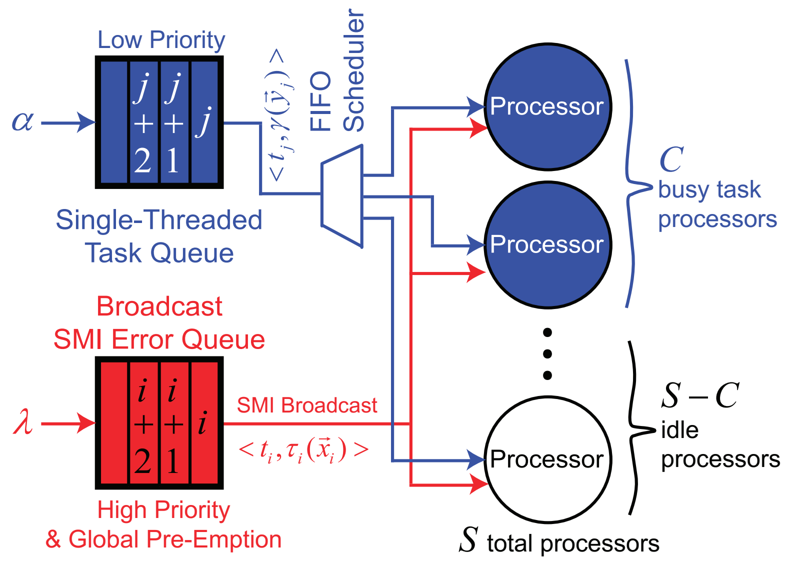
$$\boxed{\begin{aligned} t_{\text{lat,uni,approx2}}(D,K) &:= D + t_{\text{service}}(K) \\ &= C\tau + \gamma + K\tau \\ &= \gamma + \tau(C+K). \end{aligned}} \tag{5.34}$$

The PMF for $t_{\text{lat,uni,approx2}}$ is simply the convolution of PMFs for C and K:

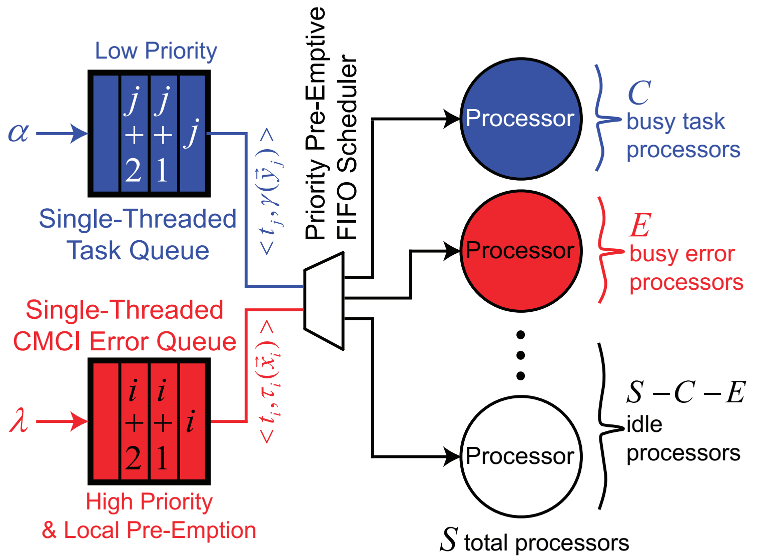
$$\boxed{\begin{aligned} P_{t_{\text{lat,uni,approx2}}} &:= \Pr\{t_{\text{lat,uni,approx2}} = t\} \\ &= \Pr\{C = c \text{ and } K = k\} = P_C * P_K. \end{aligned}} \tag{5.35}$$

5.8.2 Multiprocessor System

For this final scenario, we describe an interactive application running on a multiprocessor system, which remains an open and unsolved problem. Similar to the uniprocessor/interactive case, events in Fig. 5.10 arrive according to the exponential probability distribution with average arrival rate α and servicing times given by $t_{\text{service}}(K)$. Upon arrival, if any processor core is idle, the event is issued to that core; otherwise, the event enters a FCFS queue. The application events form an *M/G/k* queue – because the event servicing time $t_{\text{service}}(K)$ is a non-Markovian random variable,



(a) SMI error interrupts



Low Priority
α
$j+2$
$j+1$
j
$<t_j, \gamma(\vec{y}_j)>$
Priority Pre-Emptive FIFO Scheduler
Single-Threaded Task Queue
Processor
C busy task processors
Processor
E busy error processors
Single-Threaded CMCI Error Queue
λ
$i+2$
$i+1$
i
$<t_i, \tau_i(\vec{x}_i)>$
Processor
$S-C-E$ idle processors
High Priority & Local Pre-Emption
S total processors

(b) CMCI error interrupts

Figure 5.10: Queuing models for interactive applications on multiprocessors. Finding the full distributions remain unsolved problems in queuing theory.

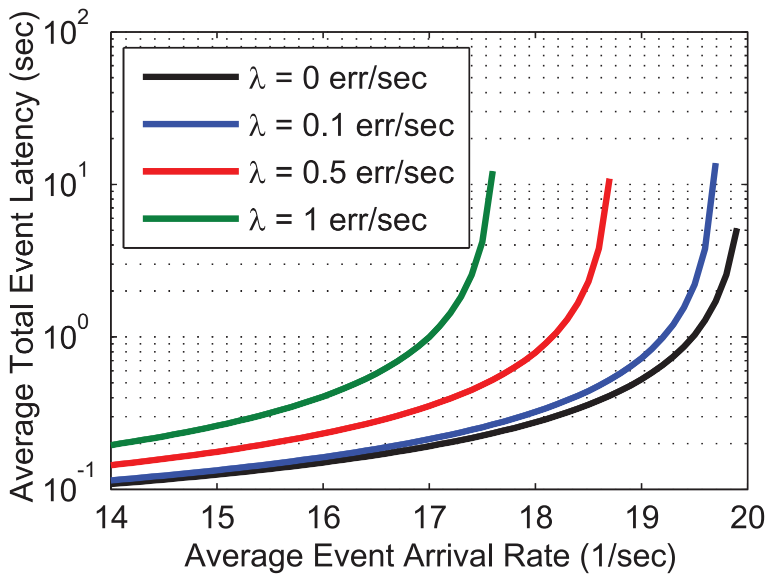


Figure 5.11: Approximation 1 of average total event latency on an application-dominated uniprocessor system for $\gamma = 50$ ms, $\tau = 133$ ms.

and there are multiple processors – which remains unsolved in queuing theory [282].

Complicating matters further, memory errors also arrive at the system randomly with average arrival rate λ using either SMI or CMCI interrupts and have handling (servicing) times of τ_i. If they arrive to an idle system, the errors also cause blocking delays, similar to Approximation 2 in our uniprocessor/interactive model described earlier. The resulting complete system resembles what we call a "$M^2/G^2/k$" queue with priority pre-emptive scheduling.

The complexity of this model makes a complete analytical solution appear intractable. Future work may seek to find hard bounds and approximate solutions to the behavior of the system. This would eventually allow one to predict some behavioral aspects of our *web search* application that we had characterized in Sec. 5.4.2.3.

5.8.3 Predicting the Performance of Interactive Applications on Uniprocessors

As we saw with our empirical *web search* results in Sec. 5.4.2.3, interactive applications tend to suffer more than batch applications from memory errors, particularly during periods of high traffic. Fig. 5.11 depicts the average total event-handling latency of a hypothetical application running on a uniprocessor system for $\alpha\gamma >> \lambda\tau$ (average error utilization is relatively low). This corresponds to Approximation 1 with SMIs ($\tau = 133$ ms). We find that the average latency can increase by $10\times$ for 1 error per second (from 100 ms at 17 event arrivals/second to 1 sec at the same event load). Conversely, for a fixed event latency, the introduction of 1 error per second can reduce event-handling throughput by 25%. Validating the model against a uniprocessor system is left to future work.

5.9 Possible Solutions

There are a number of possible ways to mitigate the problem of memory errors on application performance.

- Dramatically speed up interrupt handlers through firmware/software optimization. This opportunity may be limited.
- Change SMM architecture to allow just a single processor to execute in firmware mode concurrently with the other processors in kernel or user mode, improving performance of SMI-based error handling. However, this may have implications for firmware complexity as well as platform security.
- Expose the complete and static physical-to-DRAM organization mapping at boot time from firmware to OS through a new ACPI structure that is cached in the kernel, improving utility of CMCI-based error handling. This would add complexity to both firmware and system software.
- Leverage aggressive page retirement to remove the source of faults that are sensitized and degrading performance. This could potentially cost capacity and memory-level parallelism.
- Revert to the older error polling-style method, which bounds the time spent handling errors, but potentially reduces the fidelity of error logs for use by the OS, datacenter operators, and field study researchers. It may also reduce the common-case performance of all machines in a WSC, which may affect the tail latency of interactive applications.

- Leverage application-level load-balancing in the datacenter to mitigate the denied service of machines with memory errors. This might have limited use, however, for applications that are composed of many microservices, like those at Google [47].

We leave these solutions, as well as studying the impact of memory errors on datacenter-level performance consistency and availability, to future work.

5.10 Conclusion

Memory reliability will continue to be an important consideration in the design of WSCs for the foreseeable future. As researchers explore new directions in reliability-aware design, variation-tolerant systems, and approximate computing, we advocate for increased awareness of the more subtle effects of reliability on performance, energy, and cost. Our empirical measurements confirmed our hypothesis: a primary cause of performance degradation witnessed on machines with faulty memory is not the underlying hardware itself; rather, it is the extreme software overheads that are required to manage and report the error occurrences in the system.

We believe this is not necessarily a consequence of poor firmware or software engineering, but is driven by fundamental design decisions in Intel's current Machine Check Architecture (MCA). Because the memory controller cannot know the physical address of a memory error, firmware intervention is required via SMIs, but this comes at a significant performance cost. Moreover, one cannot simply use the lower-overhead CMCI-only reporting, because even the OS cannot easily leverage the required registers; without knowledge of the location of the DRAM error, preventative maintenance such as page retirement cannot be used, nor can datacenter technicians easily diagnose failed memory components.

Our derived analytical models allowed us to predict the performance of applications in the presence of errors; this is useful for gaining insight without resorting to un-scalable fault injection experiments. We also used them to study the theoretical benefits of page retirement and faster error handling. Page retirement is an effective technique for mitigating hard faults in memory and bounding the performance degradation caused by correctable errors. It also likely reduces the probability of detected but uncorrectable errors from occurring, which can crash or corrupt a system. We believe that page retirement should be deployed widely, and is preferable to simply masking errors at the hardware level, where the latter could reduce the quality of field studies and data-driven insights based on error patterns. Faster firmware-based error handling should bring substantial worst-case performance benefits without compromising on insights. However, this would likely require substantial changes to Intel's MCA and System Management Mode, possibly including additional hardware structures.

Compelling directions for future work include the design of improved error-handling architectures, characterizing the impact of memory errors on performance of distributed and microservice-based applications across multiple machines, and accounting for the performance degradation in machine servicing and datacenter total cost-of-ownership models.

CHAPTER 6

SDECC: Recovering from Detected-but-Uncorrectable Memory Errors using Software-Defined Error-Correcting Codes

We propose the novel idea of Software-Defined Error-Correcting Codes (SDECC) that opportunistically recover from detected-but-uncorrectable errors (DUEs) in memory. It is based on two key ideas: *(i)* for a given DUE, we compute a small list of candidate codewords, one of which is guaranteed to be correct; and *(ii)* side information about data in memory guides an entropy-based recovery policy that chooses the best candidate. A lightweight cacheline-level hash can optionally be added on top of the ECC construction to prune the list of candidates when a DUE occurs.

We demonstrate the feasibility of SDECC for linear (t)-symbol-correcting, $(t+1)$-symbol-detecting codes and analyze existing SECDED, DECTED, and SSCDSD ChipKill-Correct constructions. SDECC requires minimal architectural support in the memory controller and adds no performance, energy, or parity storage overheads during normal memory access without DUEs.

Evaluation is done using both randomized DUE error injection on data from memory traces of SPEC CPU2006 benchmarks and online during runs of the AxBench suite. We find that up to 99.9999% of double-chip DUEs in the ChipKill code with a hash can be successfully recovered. Recovering double-bit DUEs for a conventional SECDED code with approximation-tolerant applications produces unacceptable output quality in just 0.1% of cases.

Collaborators:

- Clayton Schoeny, UCLA
- Prof. Lara Dolecek, UCLA
- Prof. Puneet Gupta, UCLA

Source code and data are available at:

- `https://github.com/nanocad-lab?&q=sdecc`
- `http://nanocad.ee.ucla.edu`

6.1 Introduction

Hardware reliability is now a central issue in computing. Memory systems are a limiting factor in system reliability [143] because they are primarily designed to maximize bit storage density; this makes them particularly sensitive to manufacturing process variation, environmental operating conditions, and aging-induced wearout [263, 264]. At one extreme, embedded systems used in safety-critical applications must be dependable while satisfying stringent cost and energy constraints. At another extreme, warehouse-scale computers built for the cloud and high-performance computing contain so many components that even the most subtle of reliability problems can be exacerbated [265,283]. DRAM errors are common in warehouse-scale computers: Google has observed 70000 FIT/Mb in commodity memory, with 8% of modules affected per year [143], while Facebook has found that 2.5% of their servers have experienced memory errors per month [260]. Advances in memory resiliency can also help improve system energy efficiency [19].

Error-correcting codes (ECCs) are a classic way to build resilient memories by adding redundant parity bits or symbols. A code maps each information message to a unique codeword that allows a limited number of errors to be detected and/or corrected. Errors in the context of ECC can be broadly categorized as *corrected errors* (CEs), *detected-but-uncorrectable errors* (DUEs), *mis-corrected errors* (MCEs), and *undetected errors* (UDEs). CEs are harmless but are typically reported to system software anyway [12] as they may indicate the possibility of future DUEs. UDEs may result in *silent data corruption* (SDC) of software state, while MCEs may cause *non-silent data corruption* (NSDC)[1]; neither are desirable.

When a DUE occurs, the entire system usually panics, or in the case of a supercomputer, rolls back to a checkpoint to avoid data corruption. Both outcomes harm system availability and can cause some state to be lost. In this chapter, we consider the problem of memory DUEs, because they are more common than MCEs and UDEs and remain a key challenge to the reliability and availability of extreme-scale systems. For instance, even with state-of-the-art strong memory protection using a ChipKill-Correct ECC, the Blue Waters supercomputer suffers from a memory DUE rate of 15.98 FIT/GB [284]. This rate is high enough that whole-system checkpoints would

[1]Any data corruption that might occur from an MCE is *not* silent, because the system still knows when and where the memory error occurred. Hence, we deliberately use the term "NSDC" instead of "SDC."

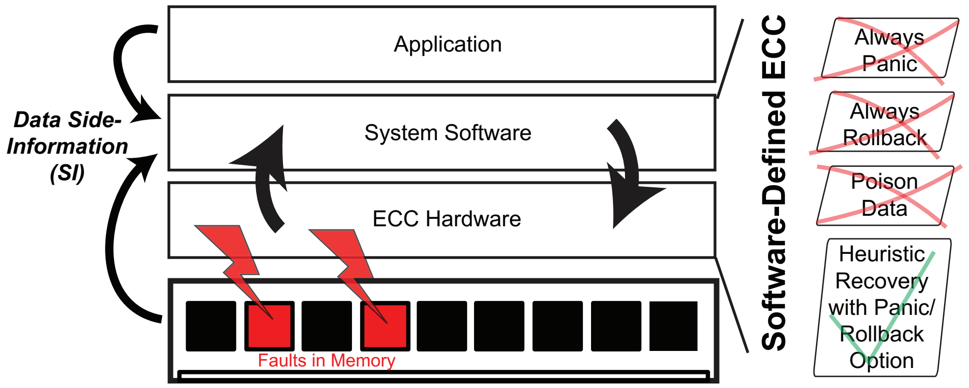


Figure 6.1: Software-Defined ECC (SDECC) concept for recovering from detected-but-uncorrectable errors (DUEs) in memory.

likely be required every few hours, and would add a significant performance and energy overhead to HPC applications [285]. For industry-standard SECDED codes that perform better and use less energy than ChipKill, DUEs are at least an order of magnitude more frequent [284] and compound the reliability/availability problem further.

The theoretical development of ECCs have – thus far – implicitly assumed that every message/information bit pattern is equally likely to occur. This essential assumption has enabled the derivation of many analytical results in the coding theory literature ever since Hamming's seminal work in 1950 [286]. In general-purpose memory systems, however, this assumption does not hold true. For instance, applications exhibit unique characteristics in control flow and data that arise naturally from the algorithm, inputs, OS, ISA, and micro-architecture. Building upon our preliminary ideas first reported in [13, 14], we demonstrate how to exploit some of these characteristics to enhance the capabilities of existing ECCs in order to recover from a large fraction of otherwise-harmful DUEs.

We propose the concept of Software-Defined ECC (SDECC), a general class of techniques spanning hardware, software, and coding theory that improves the overall resilience of systems by enabling heuristic best-effort recovery from memory DUEs. The high-level concept is depicted in Fig. 6.1. The key idea is to add software support to the hardware ECC code so that most memory DUEs can be recovered. SDECC is best suited for approximation-tolerant applications because of its best-effort recovery approach.

Our approach is summarized as follows. When a memory DUE occurs, hardware stores information about the error in a small set of configuration-space registers that we call the *Penalty Box* and raises an error-reporting interrupt to system software. System software then reads the Penalty Box, derives additional context about the error – and using basic coding theory and knowledge of the ECC implementation – quickly computes a list of all possible *candidate messages*, one of which is guaranteed to match the original information that was corrupted by the DUE. If available, an optional lightweight hash is proposed to prune the list of candidates. A software-defined data recovery policy heuristically recovers the DUE in a best-effort manner by choosing the most likely remaining candidate based on available *side information* (SI) from the corresponding un-corrupted cacheline contents; if confidence is low, the policy instead forces a panic to minimize the risk of accidentally-induced MCEs resulting in intolerable NSDC. Finally, system software writes back the *recovery target* message to the Penalty Box, which allows hardware to complete the afflicted memory read operation.

SDECC does not degrade memory performance or energy in the common cases when either no errors or purely hardware-correctable errors occur. Yet it can significantly improve resilience in the critical case when DUEs actually do occur. The contributions of this chapter are described in the following sections.

- We derive the theoretical basis for software to compute a short list of candidate messages/-codewords – one of which is guaranteed to be correct – for any (t)-symbol-correcting, $(t+1)$-symbol-detecting code upon receipt of a $(t+1)$-symbol memory DUE.
- We analyze new properties of commonly-used ECCs that demonstrate the feasibility of SDECC for real systems. We also propose optional support for a second-tier hash that can improve SDECC by pruning the list of candidate codewords before recovery.
- We describe architectural support for SDECC with main memory. In general, SDECC requires minimal changes to existing DRAM controllers and no changes to the ECC design. There is also no performance or energy penalty in the common case, i.e., in the vast majority of memory accesses when DUEs do not occur. Our optional hash costs a small amount

of extra parity bit storage. SDECC incurs no storage overheads when used with purely-conventional ECC constructions.

- We propose and evaluate a cacheline entropy-based recovery policy (called *Entropy-Z*) that utilizes SI about patterns of application data in memory to guide successful recovery from DUEs; when SI is weak, it instead aborts recovery by panicking.
- We evaluate the efficacy of SDECC with a comprehensive DUE injection campaign that uses representative SPEC CPU2006 traces. We compare our policy with alternatives and consider the impact of SI quality on recovery rates.
- We evaluate the impact of SDECC on approximate applications by using online DUE injections on the AxBench suite, and by tracking the effect of resulting MCEs on output quality (benign, tolerable NSDC, and intolerable NSDC).

On average using a conventional Hsiao SECDED code [287], SDECC can successfully recover from 71.6% of *double-bit DUEs* while causing MCEs just 4.7% of the time. Moreover, for approximate applications, we find that out of these induced MCEs, intolerable NSDCs only occur 1.5% to 10.7% of the time. For applications that are not approximation-tolerant or require higher availability, we demonstrate that with a SSCDSD ChipKill-Correct code [288, 289] SDECC can recover on average from 85.7% of *double-chip DUEs*, with an MCE rate of just 1.5%.

Finally, with the proposed 8-bit (16-bit) lightweight hash layered on top of ChipKill, SDECC can recover from 99.940% (99.9999%) of double-chip DUEs while causing MCEs just 0.002% ($< 0.00001\%$) of the time. In the hypothetical case of the Blue Waters supercomputer [284], we estimate that SDECC with ChipKill and 8-bit hash could grant up to a 18.1% application speedup by reducing the required cluster checkpoint frequency without a significant risk of accidentally-induced MCEs and possible NSDC.

This chapter is organized as follows. We begin by covering the necessary basics of ECC in Sec. 6.2 in order order to understand our theoretical contributions that immediately follow in Sec. 6.3. We then apply the derived theory to analyze the properties of existing SECDED, DECTED, and SSCDSD ChipKill-Correct codes in Sec. 6.4. The SDECC architecture is presented in Sec. 6.5, and the *Entropy-Z* data recovery policy is covered in Sec. 6.6. We have two evaluation sections: Sec. 6.7 covers reliability aspects of SDECC, while Sec. 6.8 analyzes the impact on output quality for approximate applications. We discuss system-level benefits and alternatives to SDECC in Sec. 6.9. Related work is presented in Sec. 6.10 before we conclude the chapter with directions for future work in Sec. 6.11.

6.2 ECC Background

We introduce fundamental ECC concepts that are necessary to understand the theory and analysis of SDECC. Table 6.1 summarizes the terms introduced in this section as well the introduction. Throughout this chapter, we will refer to the most important code parameters using the shorthand notation $[n,k,d_{min}]_q$ (not to be confused with the citations).

6.2.1 Key Concepts

A *linear block error-correcting code* $\mathscr{C}$ is a linear subspace of all possible row vectors of length n. The elements of $\mathscr{C}$ are called *codewords*, which are each made up of *symbols*. We refer to symbols as *q-ary*, which means they can take on q values where q is a power of 2. A symbol equivalently consists of $b = \log_2 q$ bits. For example, if $q = 2$, each symbol is a bit, yielding a binary code; if $q = 4$, we have a quaternary code where each symbol consists of two bits. Therefore, for binary codes, whenever we use the term "symbol," it is equivalent to "bit."

The code can also be thought of as an injective mapping of a given q-ary row vector *message* $\vec{m}$ of length k symbols into a codeword $\vec{c}$ of length n symbols. Because the code is linear, any two codewords $\vec{c}, \vec{c'} \in \mathscr{C}$ sum to a codeword $\vec{c''} \in \mathscr{C}$. Thus, there are $r = n - k$ redundant symbols in each codeword. A linear block code can be fully described by either its q-ary $(k \times n)$ *generator matrix* $\mathbf{G}$, or equivalently, by its q-ary $(r \times n)$ *parity-check matrix* $\mathbf{H}$. Each row of $\mathbf{H}$ is a parity-check equation that all codewords must satisfy: $\vec{c}^{\mathrm{T}} \in \mathrm{Null}(\mathbf{H})$ where T is the transpose. There are usually many ways of *constructing* a particular $\mathbf{G}$ and $\mathbf{H}$ pair for a code with prescribed k and n parameters.

Table 6.1: Important ECC Notation

Term	**Description**
CE	corrected error
DUE	detected-but-uncorrectable error
MCE	mis-corrected error
UDE	undetected error
SDC	silent data corruption
NSDC	non-silent data corruption
symbol	logical group of bits
systematic form	1:1 message symbols in codeword, parity symbols at end
construction	particular implementation of an ECC
SECDED	single-error-correcting, double-error-detecting
DECTED	double-error-correcting, triple-error-detecting
SSCDSD	single-symbol-correcting, double-symbol-detecting
ChipKill-correct	ECC construction and mem. organization that either corrects up to 1 DRAM chip failure or detects 2 chip failures
$\mathscr{C}$	linear block error-correcting code
$\mathbf{G}$	generator matrix
$\mathbf{H}$	parity-check matrix
n	codeword length in symbols
k	message length in symbols
r	parity length in symbols
b	bits per symbol
q	symbol alphabet size
t	max. guaranteed correctable symbols in codeword
$\Delta_q(\vec{u},\vec{v})$	q-ary Hamming distance between $\vec{u}$ and $\vec{v}$
d_{min}	minimum symbol distance of code
$wt_q(\cdot)$	q-ary Hamming weight
$\vec{m}$	original/intended message
$\vec{c}$	original/intended codeword
$\vec{e}$	error that corrupts original codeword
$\vec{x}$	received string with error (corrupted codeword)
$\vec{s}$	parity-check syndrome
$[n,k,d_{min}]_q$	shorthand for crucial ECC parameters
(t)SC$(t+1)$SD	(t)-symbol-correcting, $(t+1)$-symbol-detecting

To protect stored message data, one first encodes the message by multiplying it with the generator matrix: $\vec{m}\mathbf{G} = \vec{c}$. One then writes the resulting codeword $\vec{c}$ to memory. When the system reads the memory address of interest, the ECC decoder hardware obtains the *received string* $\vec{x} = \vec{c} + \vec{e}$. Here, $\vec{e}$ is a q-ary error-vector of length n that represents where memory faults, if any, have resulted in changed symbols in the codeword. The decoder calculates the *syndrome*: $\vec{s} = \mathbf{H}\vec{x}^{\mathrm{T}}$. There are no declared CEs or DUEs if and only if $\vec{s} = \vec{0}$, or equivalently, $\vec{x}$ is actually some codeword $\vec{c'} \in \mathscr{C}$. Note that even if $\vec{x} = \vec{c'} \in \mathscr{C}$, there is no guarantee that errors did not actually happen. For instance, if the error is itself a codeword ($\vec{e} \in \mathscr{C}$), then there is a UDE due to the linearity of the code: $\vec{c} + \vec{e} = \vec{c'} \in \mathscr{C}$.

6.2.2 Minimum Distance and Error Correction Guarantees

The *minimum distance d_{min}* of a linear code is defined as

$$d_{min} = \min_{\substack{\vec{u},\vec{v} \in \mathscr{C};\\ \vec{u} \neq \vec{v}}} [\Delta_q(\vec{u},\vec{v})] = \min_{\substack{\vec{c} \in \mathscr{C};\\ \vec{c} \neq \vec{0}}} [wt_q(\vec{c})] \tag{6.1}$$

where $\Delta_q(\vec{u},\vec{v})$ is the q-ary Hamming distance between vectors $\vec{u}$ and $\vec{v}$, and $wt_q(\vec{u})$ is the q-ary Hamming weight of a vector $\vec{u}$. Notice that the minimum distance is equal to the minimum Hamming weight of all non-$\vec{0}$ codewords (because $\vec{0}$ is always a codeword in a linear code).

The maximum number of symbol-wise errors in a codeword that the code is guaranteed to correct is given by $t = \lfloor \frac{1}{2}(d_{min} - 1) \rfloor$. Thus, we often refer to codes with even-valued d_{min} as (t)*-symbol-correcting,* $(t+1)$*-symbol-detecting*, or (t)SC$(t+1)$SD.

6.2.3 ECC Decoder Assumptions

The typical decoding method for ECC hardware is to choose the maximum-likelihood codeword [290] under two implicitly statistical assumptions.

- Assumption 1: all symbols in a codeword are equally likely to be afflicted by faults (the symmetric channel model).

Table 6.2: Important SDECC-Specific Notation

Term	**Description**
N	Number of ways to have a DUE
$W_q(d_{min})$	no. min. weight codewords
$\Psi(\vec{x})$ or $\lvert\Psi(\vec{x})\rvert$	list (or no.) of candidate codewords for received string $\vec{x}$
μ	mean no. of candidate codewords $\forall$ possible DUEs
P_G	prob. of choosing correct codeword for a given DUE
$\overline{P_G}$	avg. prob. of choosing correct codeword $\forall$ possible DUEs
h	Second-tier hash size in bits
`linesz`	Total cacheline size in symbols (message content)

- Assumption 2: all messages are equally likely to occur.

Under these assumptions, the maximum-likelihood *decode target* is simply the minimum-distance codeword from the received string. Under maximum-likelihood decoding, any error $\vec{e}$ with $wt_q(\vec{e}) > t$ is guaranteed to cause either a DUE, MCE, or a UDE.

The common and fastest case in ECC decoding is that there are no errors ($\vec{s} = \vec{0}$). If a construction is in *systematic form*, then the r parity bits are placed at the end of the codeword, and the first k bits are simply an identity mapping of the message $\vec{m}$. This makes it easy to obtain $\vec{m}$ when $\vec{s} = \vec{0}$. Any linear block code can be put in systematic form using elementary matrix operations.

6.3 SDECC Theory

Important terms and notation introduced in this section are summarized in Table 6.2.

SDECC is based on the fundamental observation that when a $(t+1)$-symbol DUE occurs in a (t)SC$(t+1)$SD code, there remains significant information in the received string $\vec{x}$. This information can be used to recover the original message $\vec{m}$ with reasonable certainty. It is not the case that the original message was completely lost, i.e., one need not naïvely choose from all q^k possible messages. In fact, there are exactly

$$N = \binom{n}{t+1}(q-1)^{(t+1)} \tag{6.2}$$

ways that the DUE could have corrupted the original codeword, which is less than q^k. But guessing correctly out of N possibilities is still difficult. *In practice, there are just a handful of possibilities: we call them* candidate codewords *(or candidate messages).*

If the hardware ECC decoder registers a DUE, there can be several equidistant candidate codewords at the q-ary Hamming distance of exactly $(t+1)$ from the received string $\vec{x}$. We denote the set of candidates by $\Psi(\vec{x}) \subseteq \mathscr{C}$.

Without any *side information* (SI) about message probabilities, under conventional principles, each candidate codeword is assumed to be equally likely. In other words, there is a candidate codeword more likely than the others if and only if it is uniquely closest to $\vec{x}$; in such a case, a CE could have been registered instead of a DUE (depending on the implementation of the ECC decoder).

We retain Assumption 1 from Sec. 6.2.3, which means we assume all DUEs are equally likely to occur. However, in the specific case of DUEs, *we drop Assumption 2*: this allows us to leverage SI about memory contents to help choose the right candidate codeword in the event of a given DUE.

The size of the candidate codeword list $|\Psi(\vec{x})|$ is independent of the original codeword; it depends only on the error vector $\vec{e}$ due to linearity of the code $\mathscr{C}$. We express this fact in the following lemma.

Lemma 1. *Let $|\Psi(\vec{x})|$ be the number of equidistant candidate codewords for a received string $\vec{x} = \vec{c} + \vec{e}$ that is a $(t+1)$-symbol DUE. Then*

$$|\Psi(\vec{x})| = |\Psi(\vec{c} + \vec{e})| = |\Psi(\vec{e})|.$$

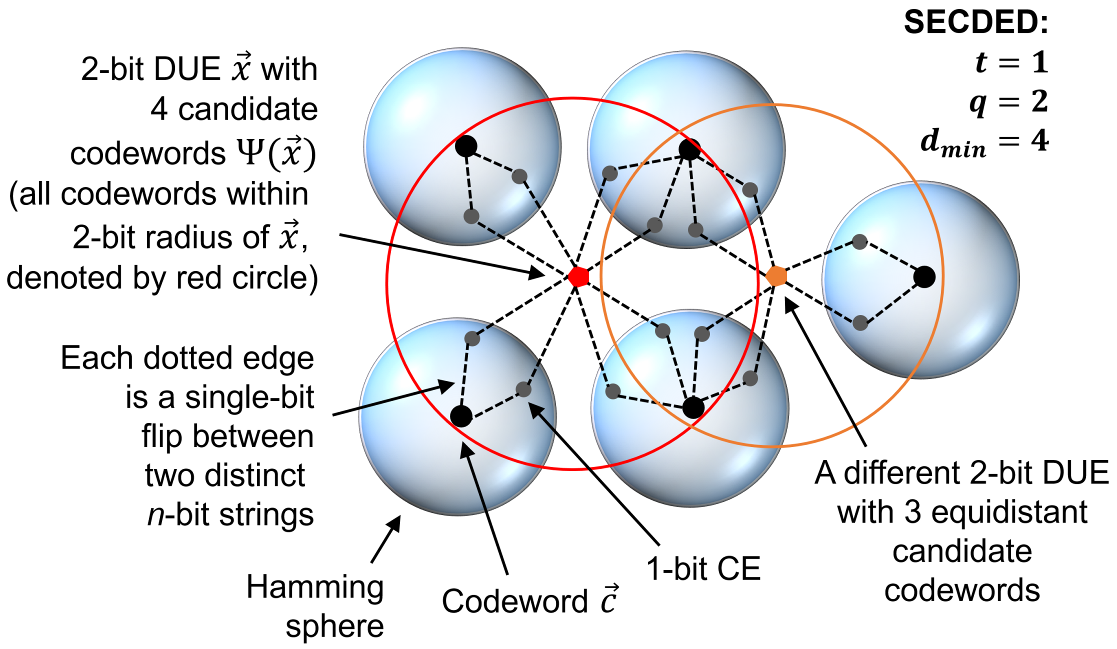

SECDED:
$t = 1$
$q = 2$
$d_{min} = 4$
2-bit DUE $\vec{x}$ with 4 candidate codewords $\Psi(\vec{x})$ (all codewords within 2-bit radius of $\vec{x}$, denoted by red circle)
Each dotted edge is a single-bit flip between two distinct n-bit strings
Hamming sphere
Codeword $\vec{c}$
1-bit CE
A different 2-bit DUE with 3 equidistant candidate codewords

Figure 6.2: Illustration of candidate codewords for 2-bit DUEs in the imaginary 2D-represented Hamming space of a binary SECDED code. The actual Hamming space has n dimensions.

Proof.

$$
\begin{aligned}
|\Psi(\vec{x})| &= |\Psi(\vec{c}+\vec{e})| \\
&= |\{\vec{c'} \in \mathscr{C} : \vec{c'} + \vec{e'} = \vec{c} + \vec{e};\ wt(\vec{e'}) = t+1\}| \\
&= |\{\vec{c''} \in \mathscr{C} : \vec{c''} + \vec{e'} = \vec{e};\ wt(\vec{e'}) = t+1\}| \\
&\text{(by linearity, } \vec{c'} - \vec{c} = \vec{c''} \in \mathscr{C}\text{)} \\
&= |\Psi(\vec{e})|.
\end{aligned}
$$

□

Notice that the *actual set* of candidate codewords $\Psi(\vec{x})$ still depends on both the error-vector $\vec{e}$ and the original codeword $\vec{c}$ (because $\vec{x} = \vec{c} + \vec{e}$).

One can better understand candidate codewords by visualizing the Hamming space of a code. Consider Fig. 6.2, which depicts the relationships between codewords, CEs, DUEs, and candidate codewords for individual DUEs for a SECDED code ($q = 2$ and $d_{min} = 4$). Here, the red-point received string $\vec{x}$ (a DUE) has four candidate codewords, marked within the red circle; these are each the minimum distance of 2 bit flips away from $\vec{x}$. Similarly, the orange-point received string has three candidate codewords, indicated by the orange circle.

We derive bounds on the number of candidate codewords, show how to compute a list of candidates for a given DUE, and explain how to prune a list of candidates using a small cacheline-level second-tier checksum.

6.3.1 Number of Candidate Codewords

The number of candidate codewords $|\Psi(\vec{e})|$ for any given $(t+1)$ DUE $\vec{e}$ has a linear upper bound that makes DUE recovery tractable to implement in practice. For instance, for any linear $[39,32,4]_2$ SECDED code, we find that there can never be more than 19 candidates for any double-bit DUE; in codes of interest, the number is usually smaller. This upper bound is given by the following lemma.

Lemma 2. *For any error $\vec{e}$ with $wt_q(\vec{e}) = (t+1)$ in a $(t)SC(t+1)SD$ linear q-ary code $\mathscr{C}$ of length n,*

$$|\Psi(\vec{e})| \leq \left\lfloor \frac{n(q-1)}{t+1} \right\rfloor .$$

Proof. The received string $\vec{x}$ is exactly q-ary distance 1 from the t-boundary of the nearest Hamming sphere(s). Thus, there are at most $n(q-1)$ single-element perturbations $\vec{p}$ such that $\vec{y} = \vec{x} + \vec{p}$ is a CE inside a Hamming sphere of a codeword. For each perturbation that results in a CE, there must be exactly t more single-element perturbations to fully arrive at a candidate codeword $\vec{c'}$. Because we cannot perturb the same elements more than once to arrive at a given $\vec{c'}$, there cannot ever be more than $\lfloor (n(q-1))/(t+1) \rfloor$ candidate codewords. □

The probability of correctly guessing the original codeword – without the use of any side information – for a specific error $\vec{e}$ is simply the reciprocal of the number of candidate codewords: $P_G(\vec{e}) = 1/|\Psi(\vec{e})|$. Let $\overline{P_G}$ be the average probability of guessing the correct codeword over all possible $(t+1)$-symbol DUEs. Also let $\sum_{\vec{e}}$ represent the summation over all possible $(t+1)$

symbol-wise error vectors $\vec{e}$. Then we have

$$\overline{P_G} = \frac{1}{N} \sum_{\vec{e}} \frac{1}{|\Psi(\vec{e})|}. \tag{6.3}$$

6.3.2 Average Number of Candidates

For a particular construction of a given code, we define $W_q(w)$ as the total number of codewords that have q-ary Hamming weight w. Then $W_q(d_{min})$ refers to the total number of minimum weight non-$\vec{0}$ codewords; its value depends on the exact constructions of **G** and **H** for given $[n,k,d_{min}]_q$ parameters. The average number of candidate codewords over all possible $(t+1)$-symbol DUEs is denoted as μ.

Lemma 3. *For a linear q-ary (t)SC$(t+1)$SD code $\mathscr{C}$ of length n and with given $W_q(d_{min} = 2t+2)$, the average number of candidate codewords μ over all possible $(t+1)$-symbol DUEs is*

$$\mu(n,t,q) = \frac{\binom{2t+2}{t+1} W_q(2t+2)}{\binom{n}{t+1}(q-1)^{(t+1)}} + 1.$$

Proof. In order to find the average number of candidate codewords, we must sum the number of candidate codewords for each unique $(t+1)$ q-ary error $\vec{e}_E$ where $E = i_1, i_2, \cdots, i_{(t+1)}$, and $i_1 \neq i_2 \neq \cdots \neq i_{(t+1)}$. We then divide that sum by the number of error-vectors (n choose $(t+1)$). By linearity and without loss of generality, assume $\vec{c} = \vec{0}$. We know that the only codewords $\vec{c'} \in \mathscr{C}$ that can satisfy $\Delta(\vec{c} - \vec{c'}, \vec{e}) = (t+1)$ have weight $W_q(d_{min})$. Each such $\vec{c'}$ that has $\vec{c}_{i_1} = \vec{e}_{i_1}$, $\vec{c}_{i_2} = \vec{e}_{i_2}$, etc., then has ($d_{min}$ choose $(t+1)$) distinct error-vectors $\vec{e}_E$. Thus summing over all error-vectors, each codeword $\vec{c'}$ with $wt(\vec{c'}) = d_{min}$ contributes to (d_{min} choose $(t+1)$) candidate codewords. To average, we divide [(d_{min} choose $(t+1)$)] $\times W_q(d_{min})$ by (n choose $(t+1)$). We also divide by $(q-1)^{(t+1)}$ because each non-zero element of the error vector $\vec{e}_E$ can take values from 1 to $q-1$. Finally, we add 1 to the expression since the original codeword $\vec{c}$ is a candidate codeword for every possible error-vector, and was not already counted in $W_q(d_{min})$. □

We find that μ is often easier to compute than $\overline{P_G}$ for long symbol-based codes; this is useful because $1/\mu$ is a lower bound on $\overline{P_G}$ according to the well-known Arithmetic-Mean Harmonic-

Mean inequality. We express this fact in the following lemma.

Lemma 4. *Over all possible ways N to have a $(t+1)$-symbol DUE $\vec{e}$, let μ be the average number of candidate codewords and $\overline{P_G}$ be the average probability of recovery. Then*

$$\frac{1}{\mu} \leq \overline{P_G}.$$

Proof.

$$\begin{aligned}\frac{1}{\mu} &= \frac{N}{\sum_{\vec{e}} |\Psi(\vec{e})|} \\ &\leq \frac{\sum_{\vec{e}} \frac{1}{|\Psi(\vec{e})|}}{N} \text{ (Arithmetic-Mean Harmonic-Mean inequality)} \\ &= \overline{P_G}.\end{aligned}$$

□

6.3.3 Computing the List of Candidates

So far we have bounded the number of candidate codewords for any $(t+1)$-symbol DUE; we now show how to find these candidates. The candidate codewords $\Psi(\vec{x})$ for any $(t+1)$-symbol DUE received string $\vec{x}$ is simply the set of equidistant codewords that are exactly $(t+1)$ symbols away from $\vec{x}$. More formally, let subscripts be used to index symbols in a vector, starting from the most significant position. Then we have the following lemma.

Lemma 5. *Let $\Psi(\vec{x})$ be the list of candidate codewords for a given $(t+1)$-symbol DUE $\vec{x} = \vec{c} + \vec{e}$. Then*

$$\begin{aligned}\Psi(\vec{x}) = \vec{c} \cup \{&\vec{c'} \in \mathscr{C} : \\ &\Delta_q(\vec{c'} - \vec{c}) = d_{min}, \vec{c'}_i = \vec{x}_i \; \forall i \text{ where } \vec{e}_i \neq 0\}.\end{aligned} \tag{6.4}$$

Proof. For a codeword $\vec{c'} \in \mathscr{C}$ to be a candidate codeword, due to linearity, it must be that $\Delta(\vec{c'}, \vec{x}) = \Delta(\vec{c'}, \vec{c} + \vec{e}) = \Delta(\vec{c'} - \vec{c}, \vec{e}) = t + 1$. Since $d_{min} = 2t + 2$, the only codewords $\vec{c'}$ other than $\vec{c}$ that satisfy

Algorithm 5 Compute list of candidate codewords $\Psi(\vec{x})$ for a $(t+1)$-symbol DUE $\vec{x}$ in a linear (t)SC$(t+1)$SD code with parameters $[n,k,d_{min}]_q$. For error vectors, subscripts indicate the symbol positions of errors, but not their q-ary values. For example, $\vec{e}_3$ corresponds to $[00100\ldots0]$.

```
for $i = 1 : n$ do
    for $j = 1 : q-1$ do
        $\vec{p} \leftarrow j * \vec{e}_i$ //(symbol $i$ in $p$ gets $q$-ary value $j$, all others 0)
        $\vec{y} \leftarrow \vec{x} + \vec{p}$
        if Decoder($\vec{y}$) not DUE then
            $\vec{c'} \leftarrow$ Decoder($\vec{y}$) //Compute candidate codeword
            if $\vec{c'} \notin \Psi(\vec{x})$ then //If candidate not already in list
                $\Psi(\vec{x}) \leftarrow \Psi(\vec{x}) \cup \vec{c'}$ //Add candidate to list
            end if
        end if
    end for
end for
```

the previous equation also satisfy $wt(\vec{c'} - \vec{c}) = d_{min}$ and $wt(\vec{c'} - \vec{e}) = d_{min}/2$. Thus a minimum weight codeword $\vec{c'}$ is a candidate codeword if and only if $t+1$ of the non-zero elements in $\vec{c'} - \vec{c}$ are in the same location as the $t+1$ elements in $\vec{e}$. □

Notice that this equation depends on the error $\vec{e}$ and original codeword $\vec{c}$, but we only know the received string $\vec{x}$; moreover, the equation does not say how to actually calculate the list $\Psi(\vec{x})$.

Fortunately, there is a simple and intuitive algorithm (shown in Alg. 5) to find the list of candidate codewords $\Psi(\vec{x})$ with runtime complexity $O(nq/t)$. The essential idea is to try every possible single symbol *perturbation* $\vec{p}$ on the received string. Each *perturbed string* $\vec{y} = \vec{x} + \vec{p}$ is run through a simple software implementation of the ECC decoder, which only requires knowledge of the parity-check matrix $\mathbf{H}$ ($O(rn\log q)$ bits of storage). Any $\vec{y}$ characterized as a CE produces a candidate codeword from the decoder output.

6.3.4 Pruning Candidates using a Lightweight Hash

In systems that require high reliability and availability, we propose to optionally prune the list of candidate codewords using a lightweight second-tier hash of several codewords grouped together in a cacheline. This would increase the success rate of SDECC while dramatically reducing the risk of MCEs. Several previous works [291–294] have also proposed the use of RAID-like multitier codes, but using traditional checksums instead than hashes. These constructions typically achieve stronger overall correction capabilities while reducing parity storage overheads for the expected types of faults. Unfortunately, these schemes tend to impact common-case performance and energy due to the required non-locality of the parity layout.

We observe that second-tier hashes can also be used to prune a list of candidate codewords for a DUE. For instance, we can compute a h-bit *original hash* of the `linesz`$\times b$ total message bits of a cacheline when it is written to memory. Then when a DUE occurs, after the candidate messages are found using Alg. 5, we can compute in software the *candidate hashes* for each *candidate cacheline* and compare them with the original that is read from memory. On average for a universal hash function, the number of *incorrect* candidates $|\Psi(\vec{x})| - 1$ will be reduced by a factor of 2^h. In most cases, only one candidate will match the original hash and we can fully correct the DUE; there is a chance of hash collision, in which case the number of candidates is still reduced but not down to one. Thus, we should not need long hashes to achieve a significant improvement in DUE recovery rate.

Errors in the original hash can cause candidate pruning to fail. However, this is a concern only when there is simultaneously both a $(t+1)$-symbol DUE $\Psi(\vec{x})$ in one of the cacheline codewords and an error in the hash. Although we consider this situation to be very unlikely, there are two possible outcomes for a universal hash.

- Outcome 1. The hash cannot prune the list of candidates because no candidates' computed hashes match. This case is fairly benign: SDECC just falls back to the full list of candidates. The probability of this lower bounded by

$$\Pr[\text{no cand. hash match} \mid \text{orig. hash corrupt}] \geq (2^h - |\Psi(\vec{x})|)/(2^h - 1). \tag{6.5}$$

- Outcome 2. The corrupted hash collides with the computed hash of an incorrect candidate. Unfortunately, this case is not benign: it causes an MCE because the original message is mistakenly pruned along with other incorrect candidates. However, for all but the smallest hashes, the probability is much less than Outcome 1 and is upper bounded by

$$\Pr[\text{wrong match} \mid \text{orig. hash corrupt}] \le (|\Psi(\vec{x})| - 1)/(2^h - 1). \tag{6.6}$$

We assume there is no more than one DUE per cacheline. Accordingly, we also assume the hash is not corrupted in order to maintain a consistent fault model. Future work can explore SDECC in the context of detailed fault models where these assumptions can be revisited.

Table 6.3: Summary of Code Properties – $\overline{P_G}$ is Most Important for SDECC

Class of Code	**Code Params.**	**Type of Code**	**Class of DUE**	**# Min. Wt. Codew.**	**# DUEs**	**Avg. # Cand. Codew.**	**LBnd. Prob. Rcov.**	**Prob. Rcov.**
	$[n,k,d_{min}]_q$		$(t+1)$	$W_q(d_{min})$	N	μ and min/max range	$1/\mu$	$\overline{P_G}$
32-bit SECDED	$[39,32,4]_2$	Hsiao [287]	2-bit	1363	741	12.04 (8 to 15)	8.31%	8.50%
32-bit SECDED	$[39,32,4]_2$	Davydov [295]	2-bit	1071	741	9.67 (7 to 19)	10.34%	11.70%
64-bit SECDED	$[72,64,4]_2$	Hsiao [287]	2-bit	8404	2556	20.73 (8 to 27)	4.82%	4.97%
64-bit SECDED	$[72,64,4]_2$	Davydov [295]	2-bit	6654	2556	16.62 (12 to 33)	6.02%	6.85%
32-bit DECTED	$[45,32,6]_2$	–	3-bit	2215	14190	4.12 (1 to 9)	24.27%	28.20%
64-bit DECTED	$[79,64,6]_2$	–	3-bit	17404	79079	5.40 (1 to 12)	18.52%	20.53%
128-bit SSCDSD	$[36,32,4]_{16}$	Kaneda [288]	2-sym.	56310	141750	3.38 (1 to 9)	29.67%	39.88%

6.4 SDECC Analysis of Existing ECCs

Code constructions exhibit structural properties that affect the number of candidate codewords $|\Psi(\vec{e})|$. Certain combinations of error positions produce fewer candidate codewords than others. This favors recovery of certain errors even if one simply guesses from the corresponding list of candidate codewords. In fact, distinct code constructions with the same $[n,k,d_{min}]_q$ parameters can have different values of μ and distributions of $|\Psi(\vec{e})|$.

We apply the SDECC theory to seven code constructions of interest in this chapter: SECDED, DECTED, and SSCDSD (ChipKill-Correct) constructions with typical message lengths of 32, 64, and 128 bits. Table 6.3 lists properties that we have derived for each of them. Most importantly, the final column lists $\overline{P_G}$ — the *random* baseline probability of successful recovery without SI.

These probabilities are far higher than the naïve approaches of guessing randomly from q^k possible messages or from the N possible ways to have a DUE. Thus, our approach can handle DUEs in a more optimistic way than conventional ECC approaches.

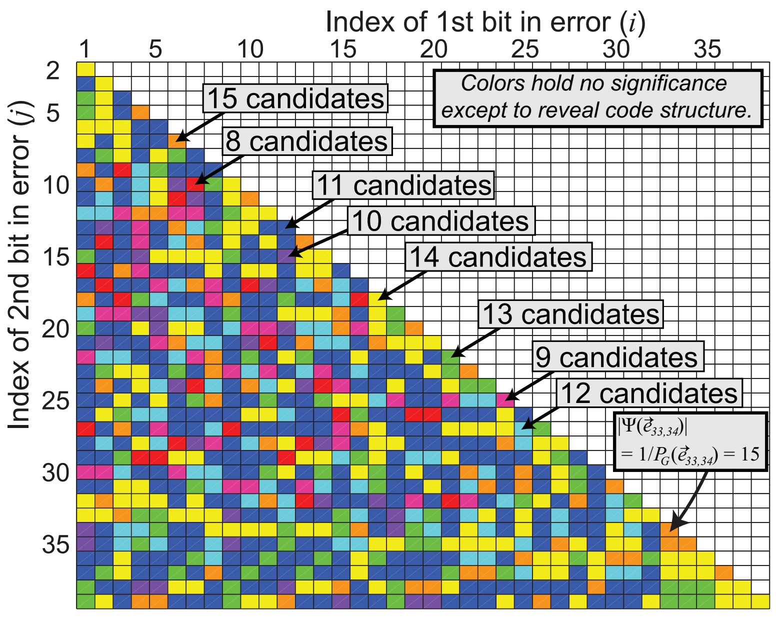
6.4.1 SECDED

The class of SECDED codes ($t = 1$, $q = 2$, $d_{min} = 4$) is simple and effective against random radiation-induced soft bit flips in memory. They can correct all possible single-bit errors and detect all possible double-bit errors. The parity-check matrix **H** of a linear SECDED code satisfies the following properties.

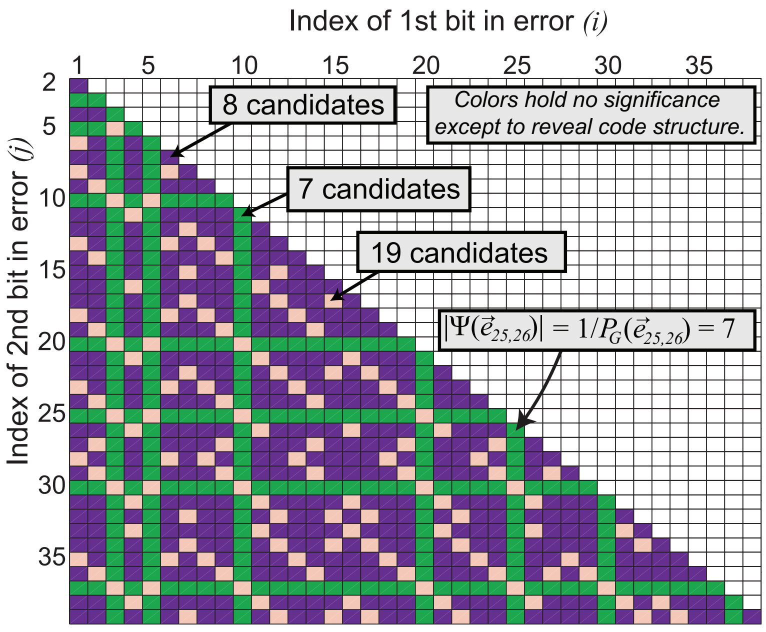
- All columns are distinct (and non-zero).
- The minimum number of columns to form a linearly dependent set is 4.

The simplest way to construct a SECDED code is to extend a $[2^r-1, 2^r-r-1, 3]_2$ Hamming code [286] with an extra parity bit, (forming a $[2^r, 2^r-r-1, 4]_2$ SECDED code), but these *complete* constructions do not exist for $k = 32$ and $k = 64$. A complete construction has the same number of candidates for all possible $(t+1)$-symbol DUEs.[2] They have $\mu = |\Psi(\vec{e})|\ \forall\ \vec{e}$, and therefore,

[2]In fact, for a complete SECDED code, the only possible DUEs are *exactly* two bits. In non-complete constructions, for example, some triple-bit errors are also DUEs, while others are MCEs, etc.

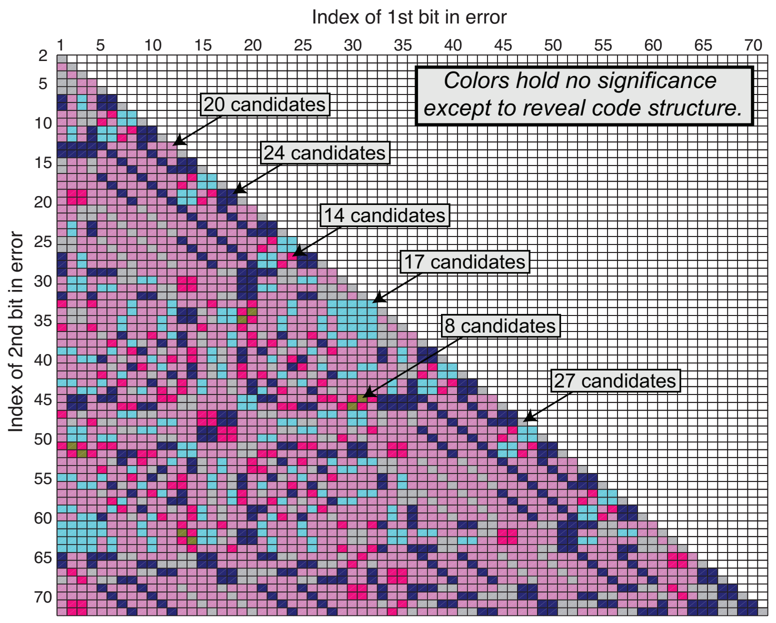


(a) Hsiao Code [287]: avg. no. candidates $\mu = 12.04$

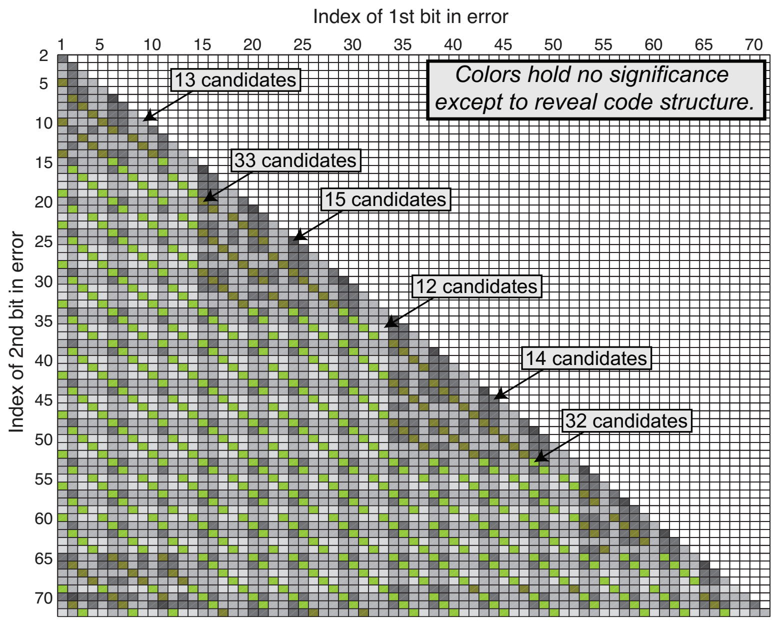


(b) Davydov Code [295]: avg. no. candidates $\mu = 9.67$

Figure 6.3: Number of candidate codewords $|\Psi(\vec{e}_{i,j})|$ for the Hsiao and Davydov $[39,32,4]_2$ SECDED codes. Indices i and j represent the positions of the two bit-errors that cause a DUE. There are $N = \binom{39}{2} = 741$ total ways to have a two-bit error.



(a) Hsiao Code [287]: avg. no. candidates $\mu = 20.73$



(b) Davydov Code [295]: avg. no. candidates $\mu = 16.62$

Figure 6.4: Number of candidate codewords for the Hsiao and Davydov $[72,64,4]_2$ SECDED codes. There are $N = \binom{72}{2} = 2556$ total ways to have a two-bit error.

$\overline{P_G} = 1/\mu$.

We are specifically interested in $k = 32$ and $k = 64$ message sizes. Hsiao's $[39,32,4]_2$ and $[72,64,4]_2$ constructions are the most common implementations of SECDED because they minimize the number of decoder logic gates [287]. These Hsiao codes are derived by truncating $[127,120,3]_2$ and $[63,57,3]_2$ Hamming codes, respectively, that were each supplemented with an extra overall parity bit.

Davydov proposed alternative and more structured SECDED codes that instead minimize the probability of an MCE when there is a triple-bit error ($wt_q(\vec{e}) = 3$) by minimizing $W_2(4)$ [295]. We find that Davydov codes have an additional advantage in context of SDECC: Lemma 3 tells us that these Davydov SECDED constructions also minimize the average number of candidate codewords μ. This can lend them an advantage for heuristic recovery.

Figs. 6.3 and 6.4 depict how the structure of the $[39,32,4]_2$ and $[72,64,4]_2$ SECDED constructions determines the number of candidate codewords for all N possible DUE patterns, respectively. For instance, the Hsiao codes (Figs. 6.3a and 6.4a) are less structured and have higher μ compared to the Davydov codes (Figs. 6.3b and 6.4b), which achieve the optimal μ (but not necessarily the optimal $\overline{P_G}$). For the SECDED codes in this chapter, the average number of candidate codewords μ ranges from 9.67 to 20.73, as shown in Table 6.3.

6.4.2 DECTED

DECTED codes ($t = 2$, $q = 2$, $d_{min} = 6$) can correct random 2-bit errors and detect 3-bit errors. While they are not typically used in commodity memory systems due to high overheads, they attract continued interest by industry and researchers. The parity-check matrix **H** of a linear DECTED code satisfies the following properties.

- All columns are distinct (and non-zero).
- The minimum number of columns to form a linearly dependent set is 6.

The easiest way to construct the code is to add an extra parity bit to a $[2^r - 1, 2^r - 2r - 1, 5]_2$ BCH code [296], extending it to a complete $[2^r, 2^r - 2r - 1, 6]_2$ DECTED code. Just like we had with the complete SECDED case, $\mu = |\Psi(\vec{e})| \; \forall \; \vec{e}$ and $\overline{P_G} = 1/\mu$.

Unfortunately, like with SECDED, there exist no such complete DECTED constructions for $k = 32, 64$. For this work, we simply add one extra parity bit to the $[127, 113, 5]_2$ and $[63, 51, 5]_2$ BCH codes [296] and then truncate them to obtain our own $[45, 32, 6]_2$ and $[79, 64, 6]_2$ DECTED constructions, respectively. For the DECTED codes in this chapter, a baseline random-candidate recovery policy has around a 20-30% chance of success.

6.4.3 SSCDSD (ChipKill-Correct)

SSCDSD codes with 4-bit symbols ($t = 1$, $q = 16$, $d_{min} = 4$) are a non-binary equivalent of SECDED codes. They can correct any error that falls within a single symbol, and detect all errors that fall within two distinct symbols, making them superior when memory faults are spatially correlated. We use Kaneda's Reed-Solomon-based $[36, 32, 4]_{16}$ construction [288]. Messages are 128 bits long and codewords are 144 bits long, so they are convenient to deploy in industry-standard DDRx-based DRAM systems that are 72 bits wide. When two DRAM channels are run in lockstep with x4 DRAM chips, $[36, 32, 4]_{16}$ SSCDSD codes have the *ChipKill-Correct* property [289]. This is because they can completely correct any errors resulting from a single-chip failure, and detect any errors caused by a double-chip failure. We find that despite there being 141750 possible ways to have a double-chip DUE, on average, there are just 3.38 candidate codewords per DUE, with the random-candidate chance of success being 39.88%. Thus, we expect ChipKill to deliver the best results in our evaluation.

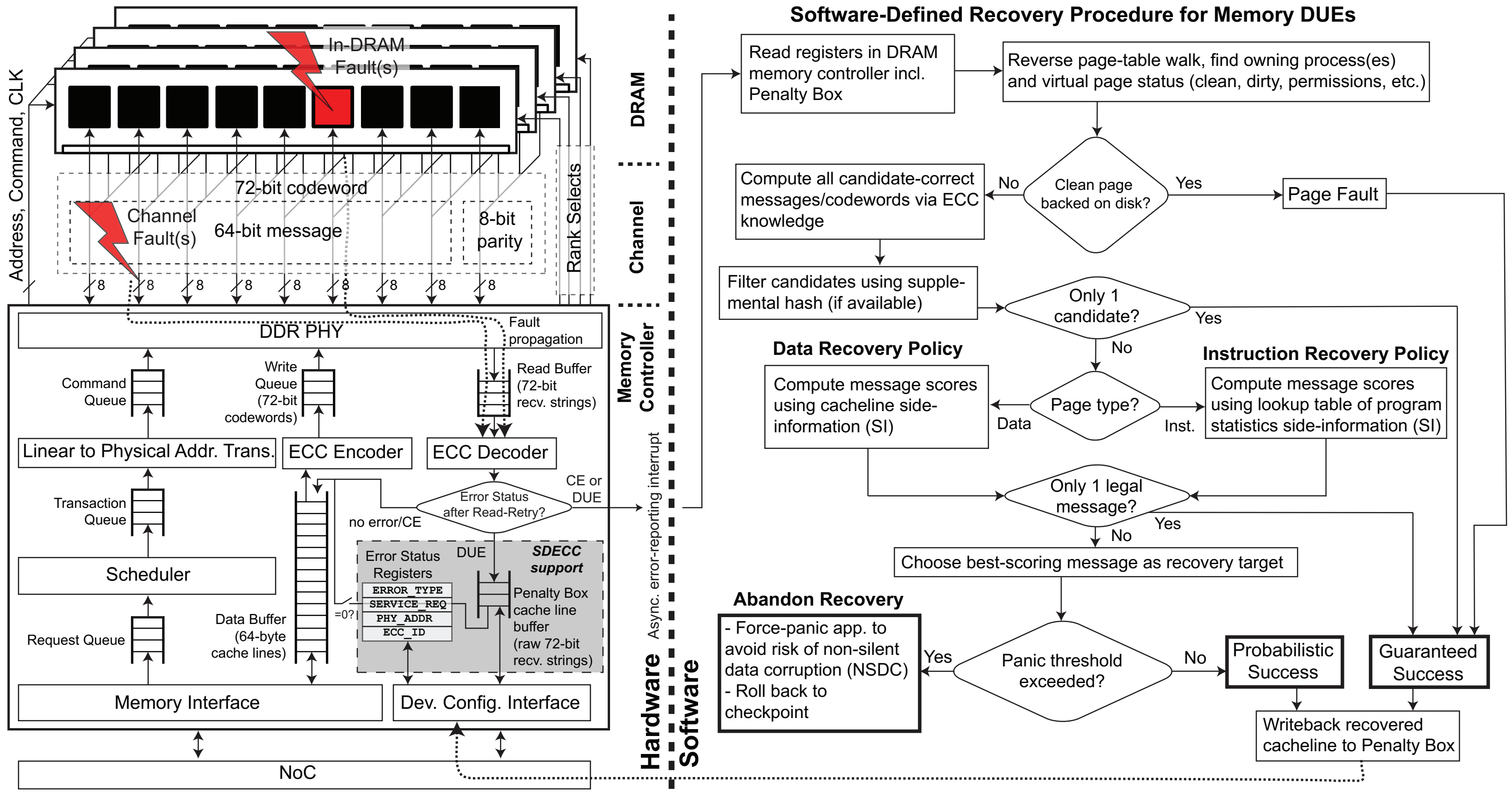

In-DRAM Fault(s)
Address, Command, CLK
72-bit codeword
Channel Fault(s)
64-bit message
8-bit parity
Rank Selects
DRAM
Channel
Memory Controller
DDR PHY
Fault propagation
Command Queue
Write Queue (72-bit codewords)
Read Buffer (72-bit recv. strings)
Linear to Physical Addr. Trans.
ECC Encoder
ECC Decoder
Error Status after Read-Retry?
CE or DUE
no error/CE
Transaction Queue
Scheduler
Error Status Registers
DUE
SDECC support
ERROR_TYPE
SERVICE_REQ
PHY_ADDR
ECC_ID
Penalty Box cache line buffer (raw 72-bit recv. strings)
Request Queue
Data Buffer (64-byte cache lines)
Memory Interface
Dev. Config. Interface
NoC
Async. error-reporting interrupt
Hardware
Software
Software-Defined Recovery Procedure for Memory DUEs
Read registers in DRAM memory controller incl. Penalty Box
Reverse page-table walk, find owning process(es) and virtual page status (clean, dirty, permissions, etc.)
Clean page backed on disk?
No
Yes
Page Fault
Compute all candidate-correct messages/codewords via ECC knowledge
Filter candidates using supplemental hash (if available)
Only 1 candidate?
Data Recovery Policy
Instruction Recovery Policy
Compute message scores using cacheline side-information (SI)
Page type?
Data
Inst.
Compute message scores using lookup table of program statistics side-information (SI)
Only 1 legal message?
Choose best-scoring message as recovery target
Abandon Recovery
- Force-panic app. to avoid risk of non-silent data corruption (NSDC)
- Roll back to checkpoint
Panic threshold exceeded?
Probabilistic Success
Guaranteed Success
Writeback recovered cacheline to Penalty Box

Figure 6.5: Block diagram of a general hardware and software implementation of SDECC. The figure depicts a typical DDRx-based main memory subsystem with 64-byte cache lines, x8 DRAM chips, and a $[72,64,4]_2$ SECDED ECC code. Hardware support necessary to enable SDECC is shaded in gray (hash support not shown). The instruction recovery policy is outside the scope of this chapter.

6.5 SDECC Architecture

SDECC consists of both hardware and software components to enable recovery from DUEs in main memory DRAM. We propose a simple hardware/software architecture whose block diagram is depicted in Fig. 6.5 and will be referred throughout this section. Although the software flow includes an instruction recovery policy, we do not present it in this chapter because DUEs on instruction fetches are likely to affect clean pages that can be remedied using a page fault (as shown in the figure). In addition to basic hardware/software support, we also describe an implementation of the optional second-tier hash support for pruning candidates prior to recovery.

6.5.1 Penalty Box Hardware

The key addition to hardware is the *Penalty Box*: a small buffer in the memory controller that can store each codeword from a cacheline (shown on the left-hand side of Fig. 6.5). When a DUE occurs on a demand read (prefetch DUEs should have the request dropped), the ECC decoder writes the raw contents of the afflicted cacheline to the Penalty Box as-is (including the parity bits). It also asserts a new `SERVICE_REQ` bit in the error status register. The memory controller also blocks the forwarding of the afflicted cacheline to the requesting hardware resource. To avoid deadlock and performance degradation of the system, the controller still allows other memory requests to complete as usual in an out-of-order fashion. After the Penalty Box and `SERVICE_REQ` bit are set, the memory controller raises an asynchronous *non-maskable-interrupt* (NMI) to the OS to report the error. Similar registers and interrupts are supported in existing systems, e.g., Intel's Machine-Check Architecture [246].

Overheads. The area and power overhead of the essential SDECC hardware support is negligible. For example, with a typical $[72,64,4]_2$ SECDED or $[36,32,4]_{16}$ SSCDSD ChipKill-Correct ECC used in DRAM with 64-byte cache lines and a DDRx burst length of eight, the Penalty Box requires just 576 flip-flops arranged in a 72-wide and eight-deep shift register. The area required per Penalty Box is approximately $736\mu\text{m}^2$ when synthesized with 15nm Nangate technology — this is approximately one millionth of the total die area for a 14nm Intel Broadwell-EP server processor [297]. This shift register would add a negligible amount of leakage power. Our SDECC design incurs no latency or bandwidth overheads for the vast majority of memory accesses where no DUEs occur. This is because the Penalty Box and error-reporting interrupt are not on the critical path of memory accesses.

6.5.2 Software Stack

The OS responds to the interrupt (right-hand side of Fig. 6.5) and reads the Penalty Box and the Error Status Register through a device configuration interface (e.g., PCI). Software then reverse-walks the page tables to determine which process(es) own the physical page containing the DUE-afflicted cacheline. It also checks the status and permissions of the mapped virtual pages. If the page is backed on disk (clean), then a viable recovery solution is to unmap the DUE-afflicted physical page and re-allocate a new one filled with clean data from disk. If not, then we rely on a SDECC recovery policy. In this chapter, we assume that DUE-afflicted pages are dirty.

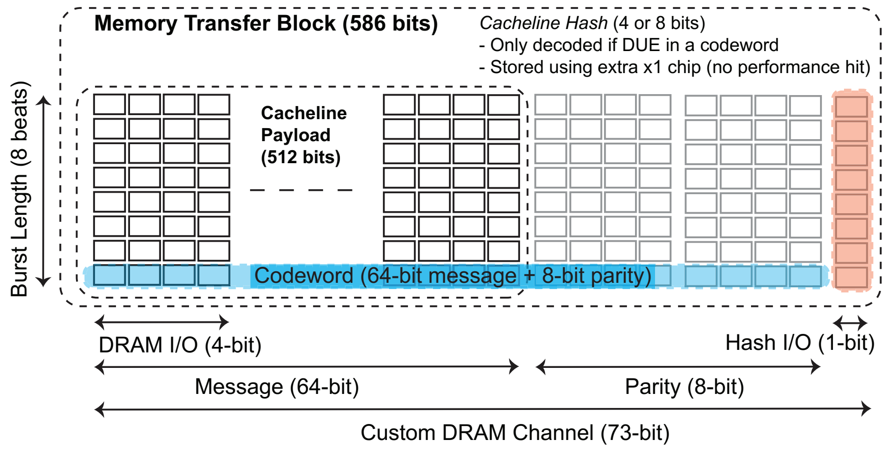
The first task of the recovery policy is to compute the list of candidate codewords for the DUE. If a hash is available, it is used to prune the list of candidates. A heuristic recovery policy (presented in Sec. 6.6) scores each remaining candidate message using available SI. Successful recovery via choosing the best candidate message is probabilistic. Using appropriate statistical metrics, if the best-scoring candidate message is not sufficiently likely to be correct, then the policy forces the machine to panic or roll-back to a checkpoint. Recovery is abandoned whenever there are multiple DUEs per cacheline or if a second DUE arrives while the first is being handled; we do not consider these scenarios explicitly.

Overheads. When a DUE occurs, the latency of the handler and recovery policy is negligible compared to the expected mean time between DUEs or typical checkpoint interval of several hours. For instance, the total execution time of DUE recovery in our un-optimized offline MATLAB implementation is 9.6 ms (using a human-friendly string-based software ECC decoder for Alg. 5). A C-based implementation of SDECC for ChipKill (our most complex case with $q = 16$) takes 2.2 million dynamic RISC-V instructions; at 3 GHz and one instruction per cycle, this requires about 733μs, making it similar to current error-reporting latencies that were reported in Chapter 5 and [12]. This would fall within the measured range of 750 μs to 130 ms per error-reporting interrupt as reported by others [12].

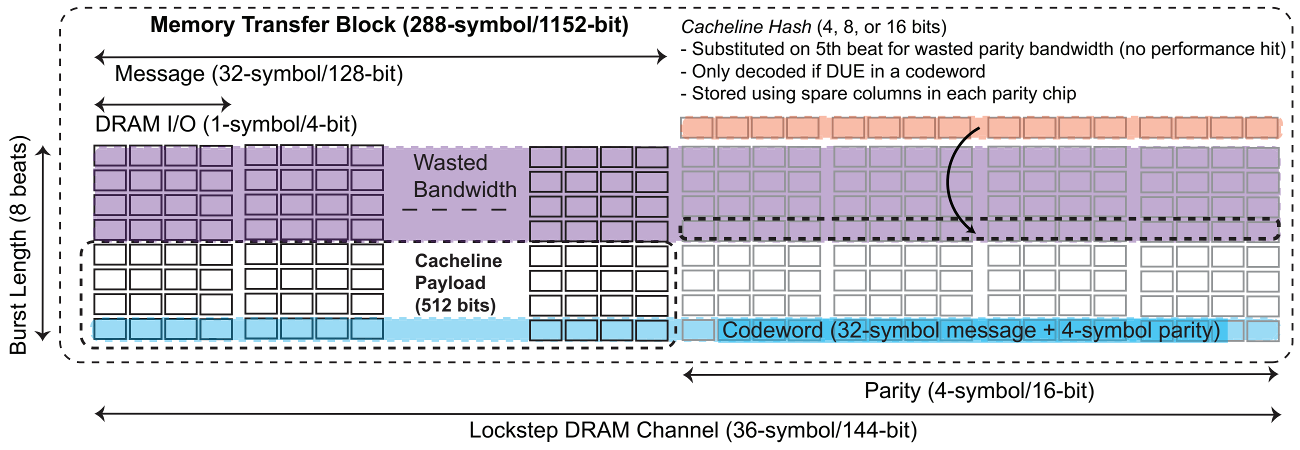
6.5.3 Lightweight Hash Implementation

We compute a small universal hash in two steps that is easy to implement in hardware. First, we take the vertical parity of the cacheline to generate a kb-bit intermediary value. We then compact it to an h-bit hash using h randomly-generated balanced parity trees where each with $kb/2$ inputs. Experimentally, we find random inputs to distribute nearly uniformly across 2^h possible hash buckets (the ideal limit).

Our hash could be computed in one clock cycle and is only on the critical path for memory writes; the single-cycle latency can be hidden by pipelining. The critical path is three 2-input XOR levels for vertical parity (up to 448 gates total) and up to six logic levels for the random trees (up to 1008 gates total). For 15nm Nangate, the total required XOR gate area is up to 644 μm^2. Our approach resembles the X-Compact tree from VLSI test compaction [298] and micro-architectural fingerprinting for processor error detection [299]. Commonly-used CRCs with the same number of check bits are a poor substitute for our hash function: they usually do not approach the universal hashing limit and are also infeasible to compute in a single clock cycle.


Memory Transfer Block (586 bits)
Cacheline Hash (4 or 8 bits)
- Only decoded if DUE in a codeword
- Stored using extra x1 chip (no performance hit)
Cacheline Payload (512 bits)
Codeword (64-bit message + 8-bit parity)
Burst Length (8 beats)
DRAM I/O (4-bit)
Hash I/O (1-bit)
Message (64-bit)
Parity (8-bit)
Custom DRAM Channel (73-bit)

(a) $[72, 64, 4]_2$ SECDED (modified DDRx channel)


Memory Transfer Block (288-symbol/1152-bit)
Cacheline Hash (4, 8, or 16 bits)
- Substituted on 5th beat for wasted parity bandwidth (no performance hit)
- Only decoded if DUE in a codeword
- Stored using spare columns in each parity chip
Message (32-symbol/128-bit)
DRAM I/O (1-symbol/4-bit)
Wasted Bandwidth
Cacheline Payload (512 bits)
Burst Length (8 beats)
Codeword (32-symbol message + 4-symbol parity)
Parity (4-symbol/16-bit)
Lockstep DRAM Channel (36-symbol/144-bit)

(b) $[36, 32, 4]_{16}$ SSCDSD (ChipKill-correct)

Figure 6.6: Proposed optional hardware support for short cacheline hashes with conventional SECDED and ChipKill-correct configurations. Lightweight hashes are useful to increase reliability of SDECC by pruning candidate messages in the event of a codeword DUE.

The second-tier hash could be accommodated in current DDRx memory systems with minor modifications. The hash is written to memory alongside the cacheline; during reads, if a DUE occurs, the original hash is stored in one additional h-bit register in the Penalty Box. Fig. 6.6 depicts two possible configurations for storing and accessing hash bits for $[72,64,4]_2$ SECDED and $[36,32,4]_{16}$ SSCDSD ChipKill-Correct memory organizations.

For SECDED (Fig. 6.6a), we propose that the standard 72-bit DRAM channel be widened by one bit to accommodate transfer of the hash bits in parallel with cacheline data as it is written. This would require an extra pin per memory module. Up to eight hash bits can be supported per cacheline; they would be stored using either an extra 1-bit-wide low-capacity DRAM device per rank, or by converting a single x4 DRAM per rank to a x5 DRAM. This design would have no impact on memory performance and a negligible impact on energy, but requires non-standard DRAM parts. For an $h = 8$-bit hash per cacheline, 1.56% additional storage is needed (128 MB per 16 GB rank).

For our ChipKill arrangement (Fig. 6.6b), we propose to transfer the hash during DDRx beats that would otherwise be wasted bandwidth (because the transfer size is larger than a cacheline). The hash bits could be stored using a few spare columns in each parity chip; up to 16 hash bits per cacheline could be supported for ChipKill with no externally-visible storage, performance, or energy overhead. If some spare columns contain local hard or soft faults, they are unlikely to have a system-level reliability impact because they are only used if there is a DUE on that particular cacheline. Corrupted hashes are not a major concern: on average, for the $[36,32,4]_{16}$ SSCDSD ChipKill-Correct code with an $h = 16$-bit hash, the probability of MCE caused by a corrupted hash (Eqn. 6.6) is just 0.003%.

6.6 Data Recovery Policy

In this work, we focus on recovery of DUEs in data (i.e., memory reads due to processor loads) because they are more vulnerable than DUEs in instructions (i.e., memory reads due to instruction fetches) as explained earlier.

There are potentially many sources of SI for recovering DUEs. Based on the notion of *data* *similarity*, we propose a simple but effective data recovery policy called *Entropy-Z* that chooses the candidate that minimizes overall cacheline Shannon entropy.

6.6.1 Observations on Data Similarity

Entropy is one of the most powerful metrics to measure data similarity. We make two general observations about the prevalence of low data entropy in memory.

- Observation 1. There are only a few primitive data types supported by hardware (e.g., integers, floating-point, and addresses), which typically come in multiple widths (e.g., byte, halfword, word, or quadword) and are often laid out in regular fashion (e.g., arrays and structs). We also note that primitive types are often vastly overprovisioned for typical data. For example, a 32-bit `int` may be used to store a binary flag or index a short loop.
- Observation 2. In addition to spatial and temporal locality in their memory access patterns, applications have inherent *value locality* in their data, regardless of their hardware representation. For example, an image-processing program is likely to work on regions of pixels that exhibit similar color and brightness, while a natural language processing application will see certain characters and words more often than others.

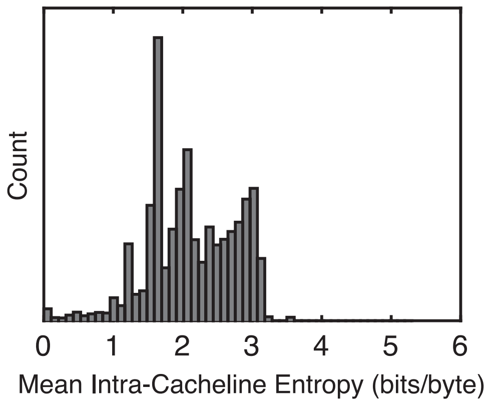
Similar observations have been made to compress memory [300–305] and to predict [306] or approximate processor load values [307–309].

We observe low byte-granularity intra-cacheline entropy throughout the integer and floating-point benchmarks in the SPEC CPU2006 suite. Let $P(X)$ be the normalized relative frequency distribution of a $\texttt{linesz}\times b$-bit cacheline that has been carved into equal-sized Z-bit symbols, where each symbol χ_i can take 2^Z possible values.[3] Then we compute the Z-bit-granularity `entropy` as follows:

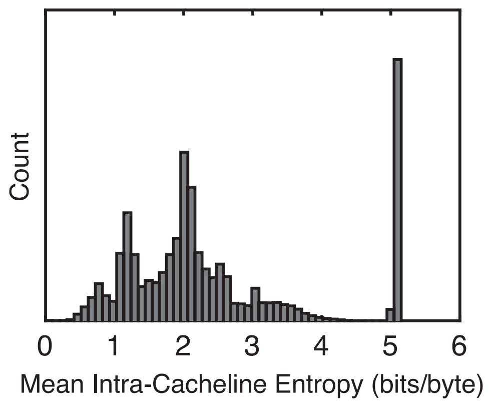
$$\texttt{entropy} = -\sum_{i=1}^{\texttt{linesz}\times b/Z} P(\chi_i) log_2 P(\chi_i). \tag{6.7}$$

Consider four representative examples for $Z = 8$ and $\texttt{linesz}\times b = 512$ bits in Fig. 6.7. The maximum possible intra-cacheline entropy here is six bits/byte because there can be only $2^6 = 64$

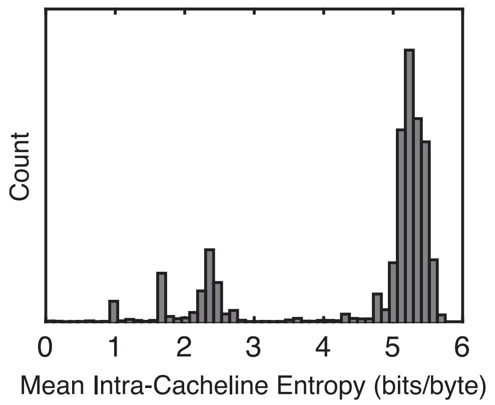
[3]Entropy symbols are not to be confused with the codeword symbols, which can also be a different size.



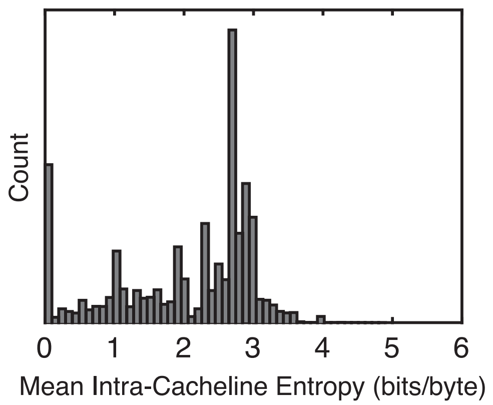
(a) `464.h264ref` (integer)



(b) `473.astar` (integer)



(c) `444.namd` (float)



(d) `450.soplex` (float)

Figure 6.7: Byte-granularity entropy distributions of 64-byte dynamic cacheline read data from two integer and two floating-point SPEC CPU2006 benchmarks.

distinct byte values in a cacheline; anything less can be exploited as SI by SDECC recovery. We find that although floating-point values tend to have higher entropy *within* a word compared to integer values, entropy *between* neighboring words is often comparable. The average intra-cacheline byte-level entropy of the SPEC CPU2006 suite to be 2.98 bits/byte (roughly half of maximum).

6.6.2 Entropy-Z Policy

We leverage these observations using our proposed data recovery policy, described in Alg. 6. Essentially, with this policy, SDECC chooses the candidate message that minimizes overall cacheline entropy.

Algorithm 6 *Entropy-Z* data recovery policy. Given a q-ary list of n-symbol candidate codewords $\Psi(\vec{x})$, a q-ary list of n-symbol error-free neighboring cacheline codewords L_n (the SI), and a `PanicThreshold` value, produce a q-ary k-symbol recovery target message $\vec{m_{\text{target}}}$ and a flag `SuggestToPanic`.

```
$M \leftarrow \Psi(\vec{x})$ with the $r$ parity symbols stripped //Extract candidate messages
$L_k \leftarrow L_n$ with the $r$ parity symbols stripped //Extract cacheline SI
Declare candidate entropy list entropy with $|M|$ elements
for $i = 1 : |M|$ do
    entropy$[i] \leftarrow Z$-bit calculation for candidate cacheline //Eqn. 6.7
    //($M[i]$ inserted into appropriate position in $L_k$)
end for
entropy$_{\text{min}} \leftarrow$ min(entropy$[i] \forall i$)
$i_{\text{min}} \leftarrow$ argmin(entropy$[i] \forall i$)
$\vec{m_{\text{target}}} \leftarrow M[i_{\text{min}}]$
if tie for entropy$_{\text{min}}$ or mean(entropy$[i] \forall i$) $>$ PanicThreshold then
    SuggestToPanic $\leftarrow$ True
else
    SuggestToPanic $\leftarrow$ False
end if
```

There is a chance that any SDECC recovery policy might choose the wrong candidate message, resulting in an MCE and possibly NSDC. On the other hand, conventional ECCs are too conservative: they force panics or checkpoint rollbacks every time a DUE is encountered. That approach avoids any risk of NSDC, but necessarily sacrifices system availability and can also cause non-committed state to be lost entirely. With SDECC, we aim to successfully recover from as many DUEs as possible while minimizing the probability that we cause an MCE.

We mitigate the risk that our policy chooses the wrong candidate message by deliberately forcing a *panic* whenever there is a tie for minimum entropy or if the mean cacheline entropy is above a specified threshold `PanicThreshold`. The downside to this approach is that some forced panics will be false positives, i.e., they would have otherwise recovered correctly.

In the rest of the chapter, unless otherwise specified, we use $Z = 8$ bits, `linesz`$\times b = 512$ bits and `PanicThreshold` $= 4.5$ bits (75% of maximum entropy), which we determine to work well across a range of applications. Additionally, as we show later, the *Entropy-8* policy performs very well compared to several alternatives.

6.7 Reliability Evaluation

We evaluate the impact of SDECC on system-level reliability through a comprehensive error injection study on memory access traces. Our objective is to estimate the fraction of DUEs in memory that can be recovered correctly using the SDECC architecture and policies while ensuring a minimal risk of MCEs.

6.7.1 Methodology

The SPEC CPU2006 benchmarks are compiled against GNU/Linux for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [310] using the official tools [311]. Each benchmark is executed on top of the RISC-V proxy kernel [312] using the Spike simulator [313] that we modified to produce representative memory access traces. We only include the 20 benchmarks which successfully ran to completion. Each trace consists of randomly-sampled 64-byte demand read cachelines, with an average interval between samples of one million accesses.

Each trace is analyzed offline using a MATLAB model of SDECC. For each benchmark and ECC code, we randomly choose 1000 q-ary messages from the trace, encode them, and inject $min(1000,N)$ randomly-sampled $(t+1)$-symbol DUEs. For each codeword/error pattern combination, we compute the list of candidate codewords using Alg. 5 and apply the data recovery policy using Alg. 6. A *successful recovery* occurs when the policy selects a candidate message that matches the original; otherwise, we either cause a *forced panic* or recovery fails by accidentally inducing an MCE. Variability in the reported results is negligible over many millions of individual experiments.

Note that the *absolute* error magnitudes for DUEs and SDECC's impact on *overall* reliability should not be compared directly between codes with distinct $[n,k,d_{min}]_q$ (e.g., a double-bit error for SECDED is very different from a double-chip DUE for ChipKill). Rather, we are concerned with the *relative* fraction of DUEs that can be saved using SDECC for a given ECC code. For evaluations that include second-tier hashes we assume there is no error in the hash itself, as explained earlier.

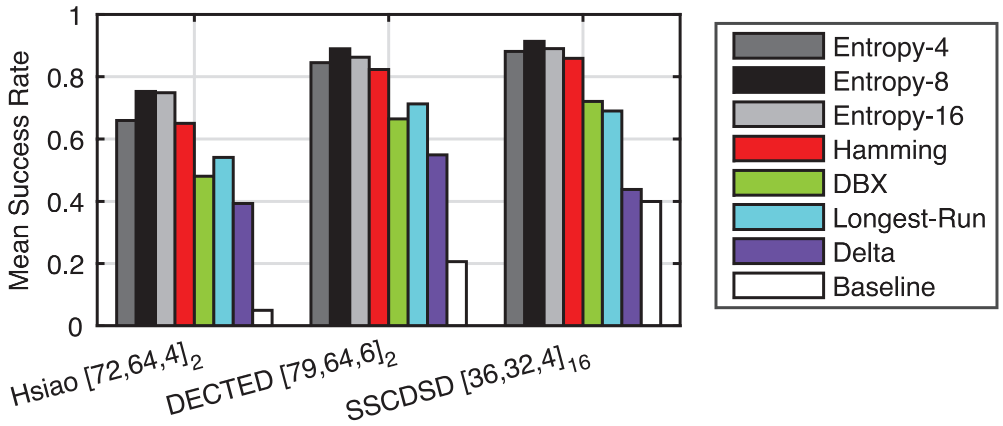


Figure 6.8: Comparison of raw success rate (no forced panics) for different SDECC data recovery policies averaged over all benchmarks. No hashes are used.

6.7.2 Comparison of Data Recovery Policies

We first compare the *raw* successful recovery rates of six different policies for three ECCs *without including any forced panics nor any second-tier hash*. Thus any un-successful recovery here is an MCE. The raw success rate averaged over the SPEC CPU2006 suite for each policy is shown for three ECC constructions in Fig. 6.8. The depicted baseline represents the average probability $\overline{P_G}$ that we randomly select the original codeword out of a list of candidates for all possible DUEs.

The alternative policies under consideration are the following. *Hamming* chooses the candidate that minimizes the average binary Hamming distance to the neighboring words in the cacheline. *DBX* chooses the candidate that maximizes 0/1-run lengths in the output of the DBX transform [305] of the cacheline.[4] *Longest-Run* is inspired by frequent pattern compression (FPC) [301] and chooses the candidate message with the longest run of 0/1 in the cacheline. *Delta* is inspired by frequent value compression (FVC) [300] and chooses the candidate message that minimizes the sum of squared integer deltas to the other words in the cacheline.

Our *Entropy-Z* policy variants recovered the most DUEs overall. Of these three, *Entropy-8* ($Z = 8$) performed better than $Z = 4$ and $Z = 16$. The 8-bit entropy symbol size performs best because its alphabet size ($2^8 = 256$ values) matches well with the number of entropy symbols per

[4]The goal of the DBX transform [305] is to reduce the entropy in a cacheline of data that is homogeneously typed (this could be an array of packed data types that include integers, floats, structs, etc.). The output of the transform ideally has long sequences of 0s that are easily compressed [305].

Table 6.4: Percent Breakdown of SDECC *Entropy-8* Policy without Hashes (S = success, P = forced panic, M = MCE)

	panics taken			**panics not taken**			***random* baseline**		
	S	**P**	**M**	**S**	**P**	**M**	**S**	**P**	**M**
***conv.* baseline**	-	100	-						
$[39,32,4]_2$ **Hsiao**	69.1	25.6	5.3	72.7	-	27.3	8.5	-	91.5
$[39,32,4]_2$ **Davydov**	70.3	25.2	4.5	76.0	-	24.0	11.7	-	88.3
$[72,64,4]_2$ **Hsiao**	71.6	23.7	4.7	75.3	-	24.7	5.0	-	95.0
$[72,64,4]_2$ **Davydov**	74.0	21.9	4.1	77.7	-	22.3	6.9	-	93.2
$[45,32,6]_2$ **DECTED**	77.5	20.3	2.2	85.5	-	14.5	28.2	-	71.8
$[79,64,6]_2$ **DECTED**	84.0	14.5	1.5	89.0	-	11.0	20.5	-	79.5
$[36,32,4]_{16}$ **SSCDSD**	85.7	12.8	1.5	91.5	-	8.5	39.9	-	60.1

cacheline (64) and with the byte-addressable memory organization. For instance, both *Entropy-4* and *Entropy-16* do worse than *Entropy-8* because the entropy symbol size results in too many aliases at the cacheline level and because the larger symbol size is less efficient, respectively.

The other four policies all significantly under-performed our *Entropy-Z* variants, with the exception of *Hamming*. It performs nearly as well as the *Entropy-4* policy for integer workloads but fails on many low-entropy cases that have low Hamming distances.

Because *Entropy-8* performed the best for all benchmarks and for all ECC constructions, we exclusively use it in all remaining evaluations.

6.7.3 Recovery Breakdown Without Hashes

Having established *Entropy-8* as the best recovery policy, we now consider the impact of its forced panics on the the successful recovery rate and the MCE rate. Again, we evaluate SDECC for each ECC using its conventional form, without any second-tier hashes to help prune the lists of candidates.

The overall results with forced panics *taken* (main results, gray cell shading) and *not taken* are shown in Table 6.4. The results for *Entropy-8* on three codes shown earlier in Fig. 6.8 are repeated in this table for comparison (i.e., the corresponding success rates when panics are not taken). There are two baseline DUE recovery policies: *conventional* (always panic for every DUE) and *random* (choose a candidate randomly, i.e., $\overline{P_G}$).

We observe that when panics are taken the MCE rate drops significantly by a factor of up to $7.3\times$ without significantly reducing the success rate. This indicates that our `PanicThreshold` mechanism appropriately judges when we are unlikely to correctly recover the original information.

These results also show the impact of code construction on successes, panics, and MCEs. When there are fewer average candidates μ then we succeed more often and induce MCEs less often. The $[72,64,4]_2$ SECDED constructions perform similarly to their $[39,32,4]_2$ variants even though the former have lower baseline $\overline{P_G}$. This is a consequence of our *Entropy-8* policy: larger n combined with lower μ provides the greatest opportunity to differentiate candidates with respect to overall intra-cacheline entropy. For the same n, however, the effect of SECDED construction is more apparent. The Davydov codes recover about 3-4% more frequently than their Hsiao counterparts when panics are not taken (similar to the baseline improvement in $\overline{P_G}$). When panics are taken, however, the differences in construction are less apparent because the policy `PanicThreshold` does not take into account Davydov's typically lower number of candidates.

The results also suggest that completely omitting the forced panic portion of our *Entropy-8* policy may be useful for high-entropy floating-point benchmarks such as `444.namd` and `454.calculix`, especially for the stronger codes like DECTED and ChipKill; this would improve those benchmarks' successful recovery rate by up to $3\times$, at the cost of more MCEs. This could be acceptable if they are approximation-tolerant.

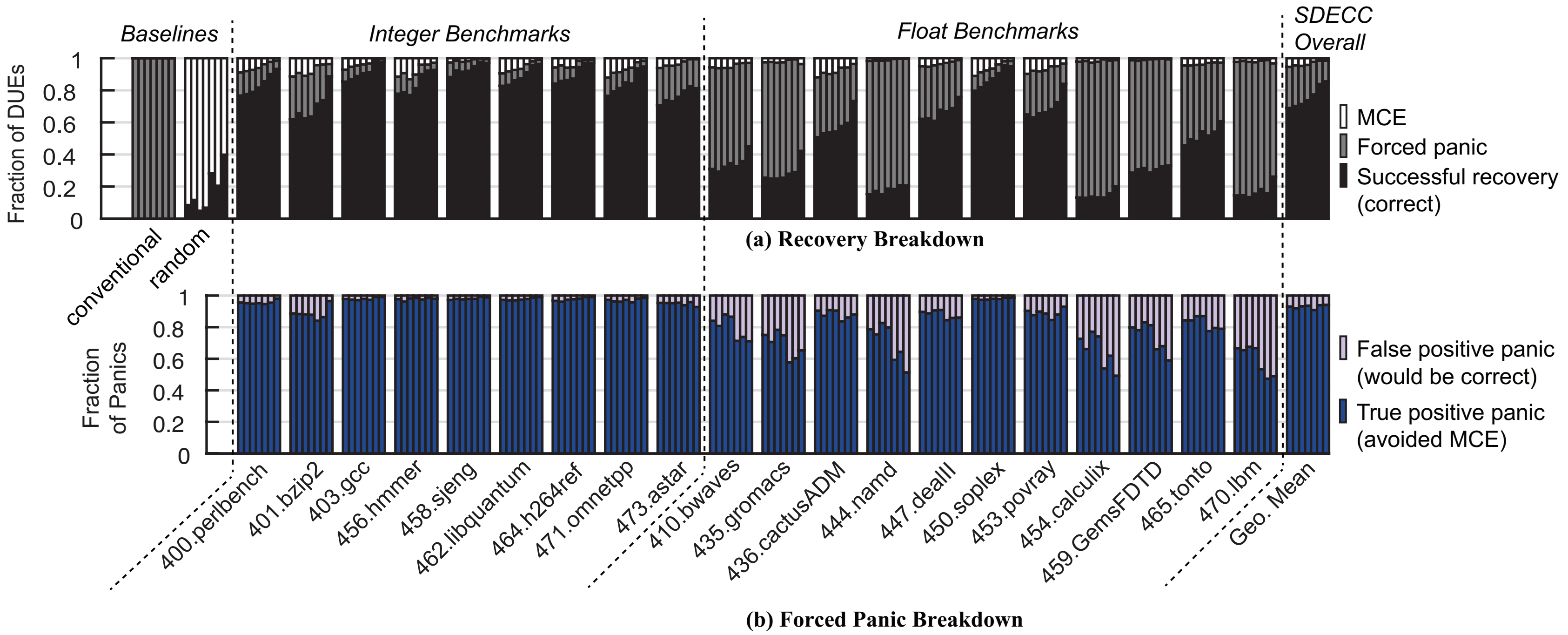


Figure 6.9: Detailed breakdown of DUE recovery results when forced panics are taken and no hashes are used. Results are shown for all seven ECC constructions, listed left to right within each cluster: $[39,32,4]_2$ Hsiao SECDED – $[39,32,4]_2$ Davydov SECDED – $[72,64,4]_2$ Hsiao SECDED – $[72,64,4]_2$ Davydov SECDED – $[45,32,6]_2$ DECTED – $[79,64,6]_2$ DECTED – $[36,32,4]_{16}$ SSCDSD ChipKill-Correct. *(a)* Recovery breakdown for the *Entropy-8* policy, where each DUE can result in a successful recovery (black), forced panic (gray), or unsuccessful recovery causing an MCE (white). *(b)* Breakdown of forced panics (gray bars in *(a)*). A true positive panic successfully mitigated a MCE, while a false positive panic was too conservative and thwarted an otherwise-successful recovery.

We examine the breakdown between successes, panics, and MCEs in more detail. Fig. 6.9 depicts the DUE recovery breakdowns for each ECC construction and SPEC CPU2006 benchmark when forced panics are taken. Fig. 6.9(a) shows the fraction of DUEs that result in success (black), panics (gray), and MCEs (white). For clarity, the two baselines from Table 6.4 are repeated on the left and the same panic taken results from the table are repeated on the right (SDECC Overall). Fig. 6.9(b) further breaks down the forced panics (gray from Fig. 6.9(a)) into a fraction that are *false positive* (light purple, and would have otherwise been correct) and others that are *true positive* (dark blue, and managed to avoid an MCE). Each cluster of seven stacked bars corresponds to the seven ECC constructions.

We achieve much lower MCE rates than the *random* baseline yet also panic much less often than the *conventional* baseline for all benchmarks, as shown by Fig. 6.9(a). Our policy performs best on integer benchmarks due to their lower average intra-cacheline entropy. For certain floating-point benchmarks, however, there are many forced panics because they frequently have high data entropy above `PanicThreshold` (e.g., as seen earlier with `444.namd` in Fig. 6.7c). A `PanicThreshold` of 4.5 bits for these cases errs on the side of caution as indicated by the false positive panic rate, which can be up to 50%. Without more side information, for high-entropy benchmarks, we believe it would be difficult for any alternative policy to frequently recover the original information with a low MCE rate and few false positive panics.

With almost no hardware overheads, SDECC used with SSCDSD ChipKill can recover correctly from up to 85.7% of double-chip DUEs while eliminating 87.2% of would-be panics; this could improve system availability considerably. However, SDECC with ChipKill introduces a 1% risk of converting a DUE to an MCE. Without further action taken to mitigate MCEs, this small risk may be unacceptable when application correctness is of paramount importance.

6.7.4 Recovery with Hashes

The second-tier hash can dramatically reduce the SDECC panic and MCE rates by pruning the list of candidate messages before applying the recovery policy.

The recovery breakdowns for second-tier hashes per cacheline on overall MCE rates using

Table 6.5: Prct. Breakdown of SDECC *Entropy-8* Policy with Hashes (S = success, P = panic, M = MCE)

checksum size	**panics taken**			**panics not taken**			***random* baseline**		
	S	**P**	**M**	**S**	**P**	**M**	**S**	**P**	**M**
***conv.* baseline**	-	100	-						
$[72,64,4]_2$ **Hsiao – 2-bit DUEs**									
none	71.6	23.7	4.7	75.3	-	24.7	5.0	-	95.0
4-bit	87.8	11.4	0.8	93.7	-	6.3	28.8	-	71.2
8-bit	98.56	1.36	0.08	99.4	-	0.6	86.6	-	13.3
$[36,32,4]_{16}$ **SSCDSD ChipKill-Correct – 2-chip DUEs**									
none	85.7	12.8	1.5	91.5	-	8.5	39.9	-	60.1
4-bit	98.05	1.86	0.09	99.2	-	0.8	77.0	-	23.0
8-bit	99.940	0.058	0.002	99.98	-	0.02	98.1	-	1.9
16-bit	99.9999	9e-5	0*	100*	-	0*	99.992	-	0.008
**out of 20 million DUE trials*									

$[72,64,4]_2$ Hsiao SECDED and $[36,32,4]_{16}$ SSCDSD ChipKill-Correct ECCs are shown in Table 6.5. The results for the non-hash cases are repeated from Table 6.4 for comparison. We do not include 16-bit hashes for the SECDED code because they are unsupported in our architecture.

The results show that even a small 4-bit hash added to every cacheline can reduce the induced MCE rate by up to substantial 16.6×. When high reliability is required, we suggest to use SDECC with a hash of at least 8 bits. Using SDECC with ChipKill and an 8-bit hash, we successfully recovered from 99.940% of double-chip DUEs; with a 16-bit hash, no MCE occurred at all in 20 million trials.

SDECC with 16-bit hashes can recover from nearly 100% of double-chip DUEs with 4× lower storage overhead than a pure DSC Double-ChipKill-Correct ECC solution and with no common-case performance or notable energy overheads. This result would be especially useful to high-reliability and high-availability systems that require good performance per dollar and energy efficiency.

Table 6.6: Prct. Breakdown of Output Quality using *Entropy-8* Policy and Hsiao $[72, 64, 4]_2$ SECDED Code without Hash

	blackscholes	**fft**	**inversek2j**	**jmeint**	**jpeg**	**sobel**
Success	83.8	49.5	82.9	90.4	92.4	90.8
Forced Panic	9.6	38.6	11.4	4.9	4.6	6.0
MCE Total	6.4	11.8	5.5	4.5	2.8	3.1
Breakdown of MCE Total						
Benign	4.8	6.5	4.2	3.2	1.5	2.5
Crash	0.5	0.9	0.2	0.8	0.6	0.5
Hang	0.4	0.0	0.0	0.0	0.0	0.0
Tol. NSDC	0.5	3.3	0.6	0.1	0.4	0.0
Intol. NSDC	0.1	1.0	0.4	0.4	0.3	0.1

6.8 Evaluation for Approximation-Tolerant Applications

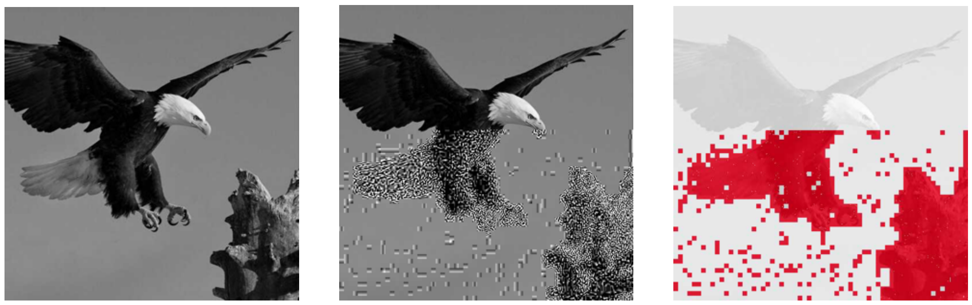
We have found that SDECC *without* second-tier hashes can still recover a large fractions of DUEs and requires almost no hardware changes. There are many approximate computing applications where some degree of output error is tolerable, but errors should be controlled as much as possible without adding too much overhead. We briefly study the effect of SDECC on application output quality using SECDED (without hash) due to its low latency, area, performance, and energy overheads which are well suited for approximate applications and cost-sensitive systems.

6.8.1 Methodology

We built AxBench [314] for RV64G in a similar fashion to Sec. 6.7.1 although we could not run `kmeans` with the proxy kernel successfully. We use our modified version of Spike to run each benchmark to completion 1000 times. For each run, using the $[72, 64, 4]_2$ Hsiao SECDED code, we inject one DUE on a random demand memory read, emulate the candidate message computation and *Entropy-8* recovery policy, and observe the effects on program behavior. No hashes are used.

6.8.2 Impact on Output Quality

The percent breakdown of attempted DUE recoveries (success, forced panic, or total MCEs) for each benchmark is shown in the top part of Table 6.6. The bottom part of the table breaks down the



(a) Correct “golden” image (b) Worst-case image (c) Image difference

Figure 6.10: An example of an intolerable NSDC for the `jpeg` benchmark from the AxBench suite [314].

MCE total further with respect to all DUEs injected. For normal program termination, output quality is judged using application-specific metrics defined by AxBench. Consistent with AxBench, we define a tolerable output to be within 10% of the “golden” result [314]. Successes and *benign* MCEs both cause 0% output error, while *tolerable NSDCs* are MCEs that result in $0\% <$ output error $< 10\%$. *Intolerable NSDCs* result in output error $\geq 10\%$. For abnormal program terminations, we characterize the cause: intentional forced panic caused by our policy, or unintentional *crashes* and *hangs* caused by induced MCEs.

For most AxBench benchmarks, our *Entropy-8* recovery policy results in similar success, forced panic, and total MCE rates to our findings with the traces from the SPEC CPU2006 suite. The most challenging case here is `fft`, which has roughly $2\times$ the MCE rate and $3\times$ to $4\times$ the forced panic rate of the other five benchmarks; this is because the program’s inputs and computations tend to produce high-entropy data.

The MCE breakdown demonstrates that intolerable NSDCs induced by SDECC are uncommon. Most induced MCEs are actually benign; this agrees with prior work on SDCs [315–317]. We find that a significant fraction of MCEs cause unintended crashes where the final outcome is no worse than the conventional baseline that always panics for every DUE. A small fraction of the MCEs result in measurable output error, but in the majority of cases, even these produce tolerable NSDCs within our 10% output quality window.

Fig. 6.10 depicts an example of a worst-case intolerable NSDC caused by an accidental MCE for the jpeg benchmark. The “golden” correct image output – which is the eventual outcome for about 94% of DUEs with SDECC – appears in Fig. 6.10a. Fig. 6.10b shows the worst-case output out of 1000 DUE injection experiments. This was caused when SDECC accidentally converted a DUE into an MCE that eventually resulted in an intolerable NSDC — similarly bad outputs occur about 0.3% of the time. Finally, Fig. 6.10c indicates the difference between the golden and worst-case outputs. Note that in a conventional baseline system, no output image would have been produced at all because it always forces panics when DUEs occur.

In the worst case overall, 1.0% of attempted DUE recoveries result in intolerable NSDCs, while in the best case, it is just 0.1%. For a naïve system that simply truncates the parity bits when a SECDED DUE occurs, the jmeint benchmark would have an intolerable NSDC rate that is 9.8× higher and a combined crash/hang rate that is 15.7× higher than our approach, respectively, even though the application is considered approximation-tolerant. This is because DUE parity-truncation only produces a legal candidate message about 1% of the time for SECDED. Therefore, SDECC is a useful low-cost aid to improve availability and reliability of approximate computing systems with minimal overheads.

6.9 Discussion

We briefly estimate the system-level availability benefits of SDECC, discuss the alternative use of stronger codes, outline ways to eventually verify the correctness of SDECC recovery, and speculate on dynamic support for fault models.

6.9.1 System-Level Benefits

We project the impact of SDECC on a typical supercomputer workload. Consider a hypothetical warehouse-scale computer that is similar to Blue Waters with 22640 compute nodes, where each has 64 GB of DRAM protected using the $[36,32,4]_{16}$ SSCDSD ChipKill-Correct code, and the memory double-chip DUE rate is 15.98 FIT/GB [284]. We use Tiwari’s checkpoint model [285]. Suppose an application nominally runs for 500 hours on the system, and that checkpoints take

Table 6.7: Projected Benefits of SDECC for a Supercomputing Application using $[36,32,4]_{16}$ SSCDSD ChipKill-Correct ECC

scheme/hash size	**opt. chkpt. intvl. [285]**	**speedup**	**util.**	**MTT ind. MCE**
Baseline	6.6 hours	-	84.4%	N/A
SDECC/none	18.4 hours	12.0%	94.5%	2.9 Khours
SDECC/4-bit	48.2 hours	16.1%	98.0%	48.0 Khours
SDECC/8-bit	272.9 hours	18.1%	99.7%	2.2 Mhours
SDECC/16-bit	N/A	18.5%	100%	N/A

30 minutes to save or restore and are taken according to the estimated optimal checkpoint interval. We assume that panics caused by memory DUEs are the only cause of failure that warrants checkpointing, and again that hashes in error.

The projected results for the baseline system and SDECC both with and without hashes is shown in Table 6.7. We find that SDECC alone can deliver a substantial 12% speedup of the application even if no hashes are used. However, an induced MCE is expected to occur once every 2900 hours, or around once in every 5.5 runs of the application. If we use an 8-bit hash can substantially reduce the optimal checkpoint interval to deliver a 18.1% speedup, with an induced MCE occurring only once every 2.2 million hours. A 16-bit hash could obviate the need to checkpoint the application entirely (the expected SDECC forced panic rate is just 0.9 ppm). Thus, we believe our approach could substantially improve the reliability and availability of a supercomputer when memory errors are a significant source of failures.

6.9.2 SDECC vs. Stronger Codes

One cannot achieve 100% DUE recovery rates without using considerably stronger ECCs or larger second-tier hashes, both of which are impractical. For instance, a SECDED code could be either be upgraded to a DEC code (estimated 2$\times$ parity storage, 2$\times$ latency, and 14$\times$ area overhead vs. SECDED [318]). Alternatively, a $[77,64,5]_2$ 4-error-detect checksum construction could be used, but it is not supported by our architecture because it needs 13 second-tier checksum bits per cacheline (only 8 checksum/hash bits can be supported). For ChipKill, we would need either a Double ChipKill-Correct construction (estimated 2$\times$ parity storage, 2$\times$ bandwidth overheads vs. SSCDSD [319, 320]) or a significantly more complex $[38,32,5]_{16}$ 4-symbol-detect checksum construction, which needs 24 extra checksum bits per cacheline ($1.5\times$ to $3\times$ more than proposed for our hashes).

6.9.3 Eventual Verification of Recovery Targets

Ideally, we wish to eliminate the risk of MCEs entirely with a mechanism for eventual verification of the recovered message. We envision several approaches that are based on the notion of correctness speculation (similar in philosophy to load value prediction [306]). One promising method is to checkpoint the state of the victim application when the DUE occurs, while the program continues execution with a recovery target. The system would attempt to catch incorrect behavior at run-time. The methods would include using assertions and checks for null pointers, out-of-bound accesses, control flow [321,322], hangs, crashes, etc. to catch induced MCEs. If an MCE is caught, the program would be rolled back to the point of the DUE and a different candidate would be tried. Alternatively, the program can be forked several times, where each copy uses a different candidate message, and their state is voted upon after some time. We leave the exploration of these ideas to future work.

6.9.4 Flexible Support for Fault Models

If a non-uniform fault model or fault map is provided, then SDECC could leverage it directly when determining the probabilities of each candidate message.

For instance, suppose that double-adjacent-bit errors are much more likely than other types of double-bit errors. Then one can use a SECDED-double-adjacent-error-correctable (SECDED-DAEC) code [323–326] in hardware. Unfortunately, the drawback of SECDED-DAEC codes compared to traditional SECDED codes is that in the former, all non-adjacent double-bit errors are guaranteed to be a MCE. SDECC with a SECDED-DAEC code could make smarter decisions about double-adjacent and other double-bit errors by considering the strength of message side information and the relative likelihood of a double-bit fault model. Unlike prior works in SECDED-DAEC, SDECC would allow the system to adapt to different fault models as appropriate.

In another example, suppose that we are given DRAM fault models [327–330]. Assume that we have refresh-induced failures in DRAM, where a $1 \rightarrow 0$ bit flip might be more likely than $0 \rightarrow 1$ bit flips (assume all data is stored using "true positive" values). Then SDECC with an ordinary SECDED code can, on average, reduce the average number of candidate codewords μ by $4\times$. This is because we can assume, due to the fault model assumption, that only $11 \rightarrow 00$ perturbations need to be tried (we would not consider the $00 \rightarrow 11$, $01 \rightarrow 10$, or $10 \rightarrow 01$ possibilities). We leave the exploration of SDECC with fault models to future work.

6.10 Related Work

6.10.1 Resilient Memory Architecture

Recently, the community has become concerned about worsening memory reliability, which can have profound implications for large-scale systems [12,143,258–260,262,269–271,283,331–333]. Problems with memory resiliency can largely be attributed to manufacturing process variations [10, 73, 74]. Accordingly, researchers have generally focused on lowering the overhead of strong ECC implementations [157, 161–163, 291, 292, 319, 320, 334–342] and dealing with hard faults [161, 162, 335, 340, 343–347].

Three state-of-the-art works relate closely to our contributions. Bamboo ECC [339], Error Pattern Transformation [346], and XED [340] each propose ways to reduce the occurrence of DUEs without necessarily increasing code strength. Bamboo ECC [339] proposes new ways to organize the placement of ECC codewords in memory to correct errors resulting from expected fault models and dramatically improve memory resiliency without increasing code strength. Error Pattern Transformation [346] recognizes that many common memory faults will result in DUEs. By remapping how codewords are stored in memory, these common faults can be spread into different codewords in such a way to cause CEs instead, thereby improving reliability. XED [340] aims to achieve ChipKill-level reliability by exposing on-DIMM SECDED ECC information to the system in a low-overhead and standards-compatible way. Unlike the work proposed in this chapter, these three approaches do not consider how to handle DUEs when they actually do occur.

Others have sought to exploit memory variability in fault-tolerant or approximation-tolerant situations to improve performance [11, 213, 228, 308, 348], energy [1, 2, 8, 116, 266, 327, 349–352], and cost [261, 262, 353]. Finally, several works have exposed the security vulnerabilities caused by memory imperfections [244, 329, 354].

6.10.2 List Decoding

SDECC is related to the theory of list decoding [355–358]. Unlike a conventional ECC decoder, which returns a single codeword, list decoders always return a set of codewords. A list decoder has "corrected" an error if the original codeword is within the list. Unfortunately, the list decoding theory has not produced a low-cost and computationally-efficient decoder suitable for use with memory. List decoding perhaps saw little interest over the decades because in general naïve implementations, the algorithm has exponential complexity. Recent works have demonstrated polynomial-time decoding algorithms, with an emphasis on Reed-Solomon codes [359, 360], but they are still far from being feasibly used in a memory system. Known list decoding algorithms still have unacceptable computational complexity that prevent them from being used in real systems.

The distinction of our approach is that it retains all the advantages of conventional ECCs – i.e., a small amount of hardware can decode uniquely and quickly in the common cases when there are no errors or just CEs – yet it can also produce a list of candidate codewords whenever a DUE occurs. We also describe a novel methodology to choose the best candidate codeword given SI about memory contents.

6.10.3 ECC Alternatives to (t)SC$(t+1)$SD Codes

There have been many advances in coding theory since Hamming's seminal work created the field in 1950 [286]. For instance, the development of LDPC codes in 1962 [361], which had been mostly ignored for decades, has recently had great success when applied to storage systems and wireless communications. RAM-type memories, however, rely almost exclusively on simple (t)SC$(t+1)$SD codes for caches and DRAM, such as those studied in this chapter. This is because latency, energy/access, and area are critical metrics. Other work has proposed using one-step majority-logic-decodable (OS-MLD) ECCs to protect memory with high fault rates, such as STT-RAM [362]. Although they are simple to decode, they have high redundancy needs r and do not lend themselves to convenient n and k. The contributions of this chapter only apply to (t)SC$(t+1)$SD codes.

6.10.4 Error-Mitigation Mechanisms

There are a number of hardware, firmware, and software methods to mitigate memory faults and errors. Aside from dedicated test routines that detect hard faults, nearly all techniques require the presence of an underlying ECC implementation for basic EDAC capabilities. These mechanisms can be broadly categorized into provisioning and opportunistic techniques.

Provisioning techniques statically add redundancy in advance of faults occurring. Note that ECCs themselves can be considered to be a provisioning method. Common but simplistic methods include sparing and mirroring of memory resources. Often, these are layered on top of an underlying ECC implementation, but are usually very costly in terms of bit storage, performance, and energy.

Opportunistic techniques are a form of dynamic reliability management. Like the provisioning techniques, they are usually layered on top of a provisioned ECC. Common methods include scrubbing (scan the memory proactively for errors and remove them before they become uncorrectable), bit steering (replace a failed memory chip with a working parity chip), and page retirement (unmap faulty memory space in the OS).

Patrol scrubbing actively scans the DRAM when the memory controller is idle, reading locations to check for the presence of faults. If faults result in CEs, a *demand scrub* is used to proactively eliminate them from the system by writing back the corrected codeword to memory. Scrubbing can eliminate benign soft faults that are otherwise at risk of producing worse errors later.

Bit steering allows for graceful degradation of memory reliability in presence of hard faults by switching the roles of hardware structures. For instance, some SSCDSD ChipKill-Correct implementations can dynamically replace a failed DRAM chip with one of the chips used for storing parity bits. This downgrades the code to SSC capability for all subsequent faults.

Page retirement disables physical memory regions that are believed to be faulty at the software level. This is a preventative measure, not a correction mechanism after-the-fact; but it can still improve memory reliability when hard faults are the dominant cause of errors.

6.11 Conclusion

SDECC is a new approach to improve the resiliency of systems by recovering from a large fraction of memory DUEs. SDECC is based on the fundamental observation that when a DUE occurs, there are a small number of candidate codewords that can be computed, wherein one is practically guaranteed to be correct. Policies that leverage SI about memory contents can be used to recover successfully from many DUEs when they occur, while incurring negligible overheads in the common cases. SDECC scales well with more powerful ECC codes, compounding its effectiveness as a general resilience technique, and there is still significant room for improvement of both codes and recovery policies. Directions for future work include adaptive software and ECC support for memory fault models and development of software mechanisms that can eventually verify the correctness of SDECC recovery.

CHAPTER 7

ViFFTo: Virtualization-Free Fault Tolerance for Embedded Scratchpad Memories

Achieving hardware reliability at low cost is a primary design consideration for IoT devices. It is difficult to efficiently address both hard and soft faults in embedded software-managed memories at the same time. To address this challenge, we propose *ViFFTo*, a holistic approach to achieve both hard and soft fault tolerance for embedded scratchpad memories in microcontroller-class systems that lack support for memory virtualization.

ViFFTo is comprised of two steps: *FaultLink* and *Software-Defined Error-Localizing Codes* (SDELCs). At software deployment time, FaultLink generates a custom application binary for each individual chip to avoid hard faults found at test-time. It does this by optimally packing small sections of program code and data into fault-free segments of the memory address space at link-time. To inexpensively deal with unpredictable soft faults at run-time, SDELC copes uses novel *Ultra-Lightweight Error-Localizing Codes* (UL-ELCs) that require half of the parity bits compared to single-error-correcting (SEC) Hamming codes. Similar to basic single-error-detecting (SED) parity, our UL-ELCs detect single-bit errors, yet additionally, localize them to a specific chunk of the codeword. SDELC heuristically recovers from localized errors using a small embedded C library that leverages side information about application memory contents. ViFFTo improves the min-VDD of embedded memory by up to 440 mV and correctly recovers from up to 90% (70%) of random bit errors in data (instructions) with just three parity bits per 32-bit word.

Collaborators:

- Irina Alam, UCLA
- Clayton Schoeny, UCLA

- Prof. Lara Dolecek, UCLA
- Prof. Puneet Gupta, UCLA

Source code and data are available at:

- `https://github.com/nanocad-lab?&q=viffto`
- `http://nanocad.ee.ucla.edu`

7.1 Introduction

Moore's Law has been the primary driving force behind CMOS technology scaling over the past several decades and has now led to the Internet-of-Things (IoT) revolution. For embedded edge devices in the IoT, hardware design is driven by the need for the lowest possible cost and energy consumption. On-chip memories comprise a major fraction of cost and energy for embedded devices [363]. They consume significant chip area and are particularly susceptible to parameter variations and defects resulting from the manufacturing process [364]. Meanwhile, much of an embedded system's total power comes from SRAM memory, particularly in sleep mode. The embedded systems community has thus increasingly turned to software-managed on-chip memories – also known as *scratchpad memories* (SPMs) [365] – due to their 40% lower energy as well as latency and area benefits versus caches [366].

It is challenging to simultaneously achieve low energy, high reliability, and low cost for embedded memory. For example, an effective way to reduce on-chip SRAM power is to reduce the supply voltage [367]. However, this causes cell hard fault rates to rise exponentially [147] and increases susceptibility to radiation-induced soft faults, thus degrading yield at low voltage and increasing cost. Thus, designers traditionally include spare rows and columns in the memory arrays [368] to deal with manufacturing defects and employ large voltage guardbands [75] to ensure reliable operation. Unfortunately, this limits the energy proportionality of memory, thus reducing battery life for duty-cycled embedded systems [369], a critical aspect of the IoT. Although many low-voltage approaches have been proposed for hardware-managed caches, fewer have addressed this problem for software-managed scratchpads with minimal cost.

To improve scratchpad reliability at low cost, we propose *ViFFTo* — a new virtualization-free fault-tolerance technique that is comprised of two steps. *FaultLink* first guards applications against known hard faults, which enables a *Software-Defined Error-Localizing Code* (SDELC) to focus on dealing with unpredictable soft faults. *The key idea is to first automatically customize an application binary to accommodate each chip's fault map with no source code disruptions, and then heuristically recover from single-bit run-time soft faults at run-time using a novel code construction.* The contributions of this chapter are the following.

- We present FaultLink, a novel link-time approach that extends the software construction toolchain with new fault-tolerance features for scratchpad memories. FaultLink relies on byte-level fault maps of each software-managed region of memory.
- We present an algorithm for FaultLink that automatically synthesizes custom fault-aware linker scripts for each chip. FaultLink first compiles the program using specific compiler flags to carve up the typical monolithic sections, e.g., `.text`, `.data`, stack, heap, etc. It then attempts to optimally pack program sections into memory segments that correspond to contiguous regions of address space that are free of hard faults.
- We propose SDELC, a hardware/software hybrid technique that allows the system to heuristically recover from unpredictable single-bit soft faults in instruction and data memories that cannot be handled using FaultLink alone. SDELC relies on available *side information* (SI) about the typical patterns of application instructions and data in memory.
- We describe the novel class of *Ultra-Lightweight Error-Localizing Codes* (UL-ELCs) – used by SDELC – that are stronger than basic single-error-detecting (SED) parity, yet they have no more than half of the storage overheads of a single-error-correcting (SEC) Hamming code. Like SED, UL-ELC codes can detect single-bit errors, but additionally, they can localize them to a chunk within a memory codeword.

By experimenting on real test chips and in a simulator, we find that FaultLink enables applications to run correctly on embedded memories while reducing min-VDD by up to 440 mV with no hardware changes. We demonstrate that while a FaultLink-enabled application correctly executes on memory containing hard faults, our approach can additionally recover from up to 90% of random single-bit soft faults in 32-bit data memory words and up to 70% of errors in instruction memory using a 3-bit UL-ELC code (9.375% storage overhead) in conjunction with embedded SDELC library. SDELC can even be used to recover up to 70% of errors using a basic SED parity code (3.125% storage overhead), which requires just $1/6$ of the parity bits of a Hamming code (18.75% storage overhead).

To the best of our knowledge, this is the first work to both (i) customize an application binary *on a per-chip basis at link-time to accommodate the unique patterns of hard faults in embedded scratchpad memories, and (ii) use error-localizing codes with software-defined recovery to cope with random bit flips at run-time.*

This chapter is organized as follows. Background material and related works are presented in Sec. 7.2. We then describe the high-level ideas of FaultLink and SDELC to achieve low-cost embedded fault-tolerant memory in Sec. 7.3. FaultLink and SDELC are described in greater detail in Secs. 7.4 and 7.5, respectively. ViFFTo is evaluated in Sec. 7.6 before a discussion of other considerations and opportunities for future work are presented in Sec. 7.7. We conclude the chapter in Sec. 7.8.

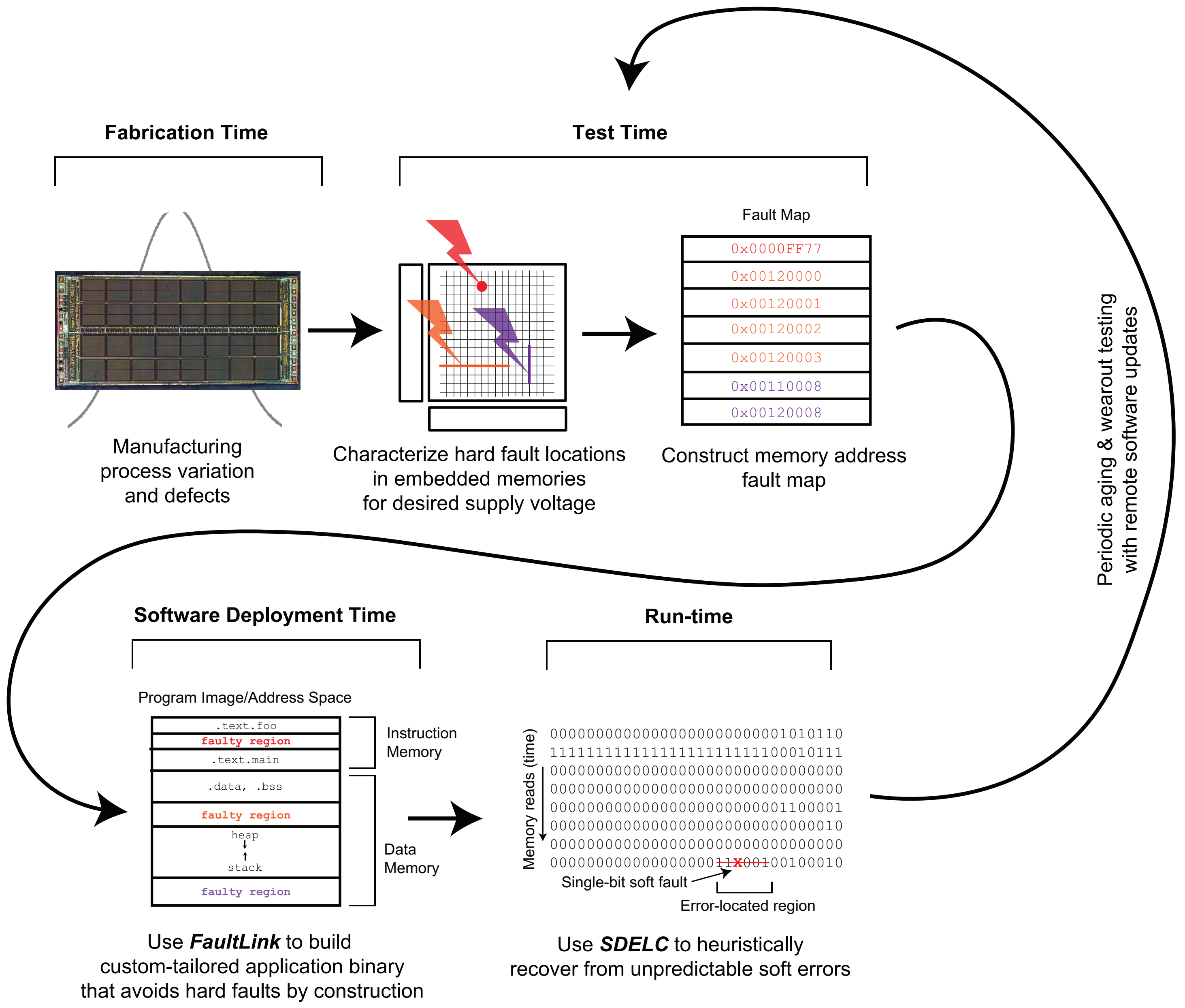


Figure 7.1: Our high-level approach to tolerating both hard and soft faults in on-chip scratchpad memories.

7.2 Background and Related Work

We present the essential background on scratchpad memory, the nature of SRAM faults, sections and segments used by software construction linkers, and error-localizing codes needed to understand our contributions. We then summarize the related work.

7.2.1 Scratchpad Memory

In energy and cost-conscious embedded systems, software-managed scratchpad memories (SPMs) are often used in lieu of hardware-managed caches to achieve higher area efficiency and lower power. SPMs are small on-chip memories that, like caches, can help speed up memory accesses that exhibit spatial and temporal locality. Also like caches, SPMs can also be separated into data and instruction memory. Unlike data caches, however, the application programmer must – with the help of the compiler and linker – explicitly partition data into each physical data SPM and memory area, which each comprise distinct regions of the address space. Banakar et al. showed that SPMs have on average 33% lower area requirements and can reduce energy by 40% compared to equivalently-sized caches [366].

7.2.2 Program Sections and Memory Segments

The Executable and Linkable Format (ELF) is the ubiquitous standard on Unix-based systems for representing compiled object files, static and dynamic shared libraries, as well as program executable images in a portable manner [370]. ELF files contain a header that specifies the ISA, ABI, and more for the executable. The primary data structures in ELF for representing information are program sections and memory segments.

- A *section* is a contiguous chunk of bytes with an assigned name: sections can contain instructions, data, or even debug information. For instance, the well-known `.text` section typically contains all executable instructions in a program, while the `.data` section contains initialized global variables.
- A *segment* represents a contiguous region of the memory address space (i.e., ROM, instruc-

tion memory, data memory, etc.). When a final output binary is produced, the linker maps sections to segments. Each section may be mapped to at most one segment; each segment can contain one or more non-overlapping sections.

The toolchain generally takes a section-centric view of a program, while at run-time the segment-centric view represents the address space layout.

7.2.3 Tolerating SRAM Faults

There are several types of SRAM faults. In this chapter, we define *hard faults* to include all recurring and/or predictable failure modes that can be characterized via testing at fabrication time or in the field. These include manufacturing defects, weak cells at low voltage, and in-field device/-circuit aging and wearout mechanisms [10]. A common solution to hard faults is to characterize memory, generate a *fault map*, and then deploy it in a micro-architectural mechanism to hide the effects of hard faults.

We define *soft faults* to be unpredictable *single-event upsets* (SEUs) that do not generally re-occur at the same memory location and hence cannot be fault-mapped. The most well-known and common type of soft fault is the radiation-induced bit flip in memory [371]. Soft faults, if detected and corrected by an *error-correcting code* (ECC), are harmless to the system.

7.2.4 Error-Correcting Codes (ECCs)

ECCs are mathematical techniques that transform *message* data stored in memory into *codewords* using a hardware encoder. When soft faults affect codewords, causing bit flips, the ECC hardware decoder is designed to detect and/or correct a limited number of errors. ECCs used for random-access memories are typically based on linear block codes.

The encoder implements a binary generator matrix **G** and the complementary decoder implements the parity-check matrix **H** to detect/correct errors. To encode a binary message $\vec{m}$, we multiply its bit-vector by **G** to obtain the codeword $\vec{c}$: $\vec{m}\mathbf{G} = \vec{c}$. To decode, we multiply the stored codeword (which may have been corrupted by errors) with the parity-check matrix to obtain the syndrome $\vec{s}$, which provides error detection and correction information: $\mathbf{H}\vec{c}^T = \vec{s}$. Typical ECCs used for memory have the generator and parity-check matrices in systematic form, i.e., the message bits are directly mapped into the codeword and the redundant parity bits are appended at the end. This makes it easy to extract data from decoded codewords.

Typical ECC-based approaches can tolerate random bit-level soft faults but they quickly become ineffective when multiple errors occur due to hard faults. Meanwhile, powerful schemes like ChipKill [289] have unacceptable overheads and are not suited for embedded memories.

7.2.5 Related Work

We briefly summarize related work on fault-tolerant caches, reliable software-managed memories, as well as error-localizing and unequal error protection codes.

7.2.5.1 Fault-Tolerant Caches

Many fault-tolerant cache techniques do not apply to SPMs because they sacrifice cache capacity to tolerate hard faults or save energy; this affects the software-visible memory address space and hence they cannot be applied to SPMs. Examples include PADded Cache [154], Gated-VDD [151], Process-Tolerant Cache [155], Variation-Aware Caches [372], Bit Fix/Word Disable [167], ZerehCache [159], Archipelago [169], FFT-Cache [149], VS-ECC [162], FLAIR [163], Macho [174], DPCS [8], DARCA [373], and others [150,263]. Like SDELC, Correctable Parity Protected Cache recovers soft faults using parity and additional hardware bookkeeping whenever data is added, modified, or removed from the cache [374], but the method is not applicable to SPMs.

7.2.5.2 Reliable Software-Managed Memories

The community has proposed other methods for tolerating variability and hard faults that can apply to SPMs. OS page retirement [375], Embedded RAIDs-on-Chip [110], Flikker [266], VaMV [116] and ViPZonE [1] all propose to use OS memory virtualization to directly manage memory variations and/or hard faults, but they are not supported in low-cost IoT devices that lack support for virtual memory, nor do they guarantee hard faults to be avoided at the software build time. Others have proposed to add small fault-tolerant buffers that assist SPM checkpoint/restore [376], re-compute corrupted data upon detection [377], build radiation-tolerant SPMs using hybrid non-volatile memory [378], and duplicate instruction storage [379] and data storage to guard against soft errors [380]. Farbeh et al. [379] use basic parity and a software handler to recover from instruction SPM soft errors, but they employ duplication rather than our SDELC approach that uses heuristically recovers information. Similar to FaultLink, Volpato et al. proposed a post-compilation approach to improve energy efficiency via SPMs [381] but it does not deal with fault tolerance. Meanwhile, traditional fault avoidance using dynamic bit-steering [382] is too costly for small memories, while spare rows and columns cannot scale to many faults that could arise from deep voltage scaling. Unlike our work, none of the above approaches can simultaneously deal with both hard and soft SPM faults with minimal hardware changes to existing systems.

7.2.5.3 Error-Localizing and Unequal Error Protection Codes

In the 1960s Wolf et al. introduced *error-localizing codes* (ELCs) – which attempt to detect errors and identify the erroneous fixed-length *chunk* of the codeword – and established some fundamental bounds [383–385]. ELCs have since been adapted in theory to byte-addressable memory systems [386] but until now had not found use in any practical system. Others have studied how to add SEC capabilities to ELCs that are capable of localizing multi-bit errors [386, 387], or focused on infinite-length codes [388], iterative codes [389, 390], extremely long-length codes [391], and codes that are strictly better than Hamming codes [392, 393].

To the best of our knowledge, ELCs in the regime between SED and SEC capabilities has not been previously studied. We describe the basics of Ultra-Lightweight ELCs (UL-ELCs) that lie in this regime and apply them to recover from a majority of single-bit soft faults.

Our UL-ELC codes are also related to unequal error protection (UEP) codes [392]. However, direct application of these codes is not possible in our regime as UEP coding research has mainly focused on information-theoretic results.

7.3 Approach

We propose FaultLink and SDELC that together form a novel hybrid approach to low-cost embedded memory fault-tolerance. They specifically address the unique challenges posed by software-managed on-chip memories, which we often refer to as scratchpad memories (SPMs).

The high-level concept is illustrated in Fig. 7.1. At fabrication time, process variation and defects may result in hard faults in embedded memories. During test-time, these are characterized and maintained in a per-chip fault map. When the system developer deploys application software onto devices, they use FaultLink to customize their binary for each chip in a way that avoids the hard fault locations. Finally, at run-time, unpredictable soft faults are detected, localized, and recovered heuristically using Software-Defined Error-Localizing Codes. (Note that FaultLink is not heuristic and does not introduce extra errors.) We now explain the approaches of the FaultLink and SDELC steps further.

7.3.1 FaultLink: Avoiding Hard Faults at Link-Time

Conventional software construction toolchains assume that there is a contiguous memory address space in which they can place program code and data. For embedded targets, the address space is often partitioned into a region for instructions and a region for data. On a chip containing hard faults, however, the specified address space can contain faulty locations. With a conventional compilation flow, a program could fetch, read, and/or write from these unreliable locations, making the system unreliable.

FaultLink is a modification to the traditional embedded software toolchain to make it memory "fault-aware." At chip test-time, or periodically in the field using built-in-self-test (BIST), the on-chip scratchpad memories are characterized to identify memory addresses that contain hard faults.

At software deployment time – i.e., when the application is actually programmed onto a particular device – FaultLink customizes the application binary image to work correctly on that particular chip given the fault map as an input. FaultLink does this by linking the program to guarantee that no hard-faulty address is ever read or written at runtime. However, the fault mapping approach taken by FaultLink cannot avoid random bit flips at run-time; these are instead addressed at low cost using SDELC.

7.3.2 Software-Defined Error-Localizing Codes (SDELCs): Recovering Soft Faults at Run-Time

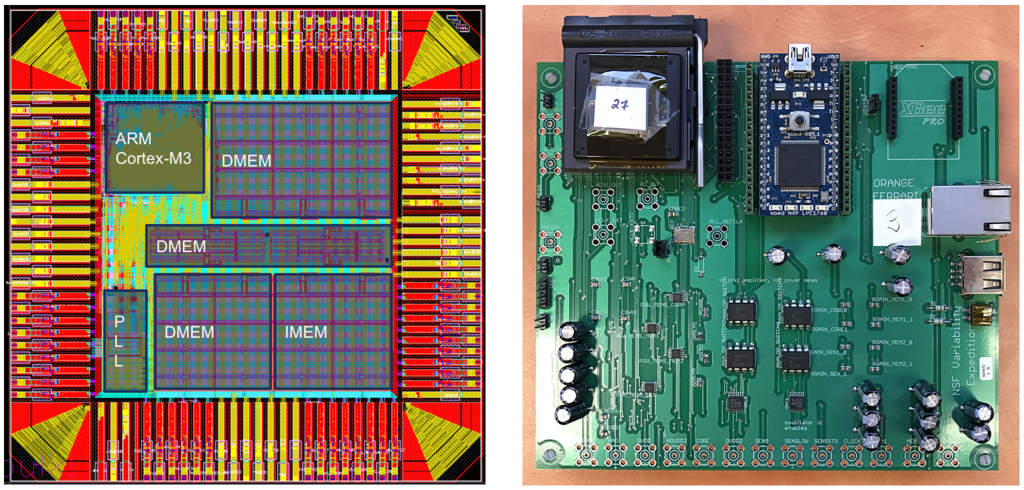
Typically, either basic parity is used to detect random single-bit errors or a Hamming code is used to correct them. Unfortunately, Hamming codes are expensive for small embedded memories: they require six bits of parity per memory word size of 32 bits (an 18.75% storage overhead). On the other hand, basic parity only adds one bit per word (3.125% storage overhead), but without assistance, it cannot correct errors.

To address this problem, we propose Software-Defined Error-Localizing Codes (SDELCs) that are built on our novel idea of Ultra-Lightweight Error-Localizing Codes (UL-ELCs). UL-ELCs have lower storage overheads than Hamming codes: they can detect and then *localize* any single-bit error to a chunk of a memory codeword. We construct distinct UL-ELC codes for instruction and data memory that allows a software-defined recovery policy to heuristically recover the error by applying different semantics depending on the error location. The policies leverage available side information (SI) about memory contents to choose the most likely *candidate codeword* resulting from a localized bit error. In this manner, we attempt to correct a majority of single-bit soft faults without resorting to a stronger and more costly Hamming code. SDELC can even be used to recover many errors using just a basic SED parity code. The approach is similar to one recently proposed by others for conventional ECCs [14].

We now discuss FaultLink in greater depth before revisiting the details of SDELC in Sec. 7.5.

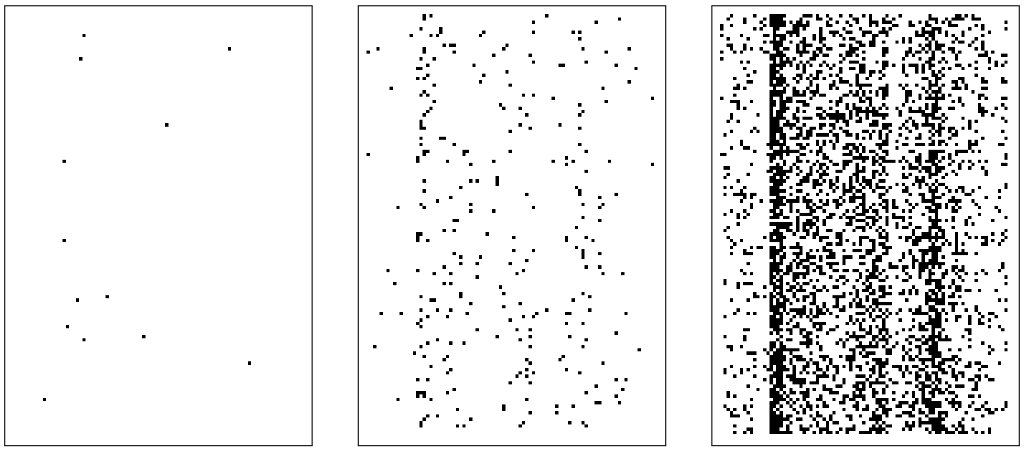
7.4 FaultLink: Dealing with Hard Faults

We motivate FaultLink with fault mapping experiments on real test chips, describe the overall FaultLink toolchain flow, and present the details of the *Section-Packing* problem that FaultLink



(a) Chip floorplan (b) Board

Figure 7.2: Test chip and board used to collect SPM hard fault maps for FaultLink.



(a) 750 mV (b) 700 mV (c) 650 mV

Figure 7.3: Measured voltage-induced hard fault maps of the 176 KB data SPM for one test chip. Black pixels represent faulty byte locations.

solves.

7.4.1 Test Chip Experiments

We characterized the voltage scaling-induced fault maps for eight “Orange Ferrari” ARM Cortex-M3 microcontroller test chips [7, 85, 185, 186] fabricated for various projects under the umbrella of the NSF Variability Expedition. Each chip has 176 KB of on-chip data memory and 64 KB of

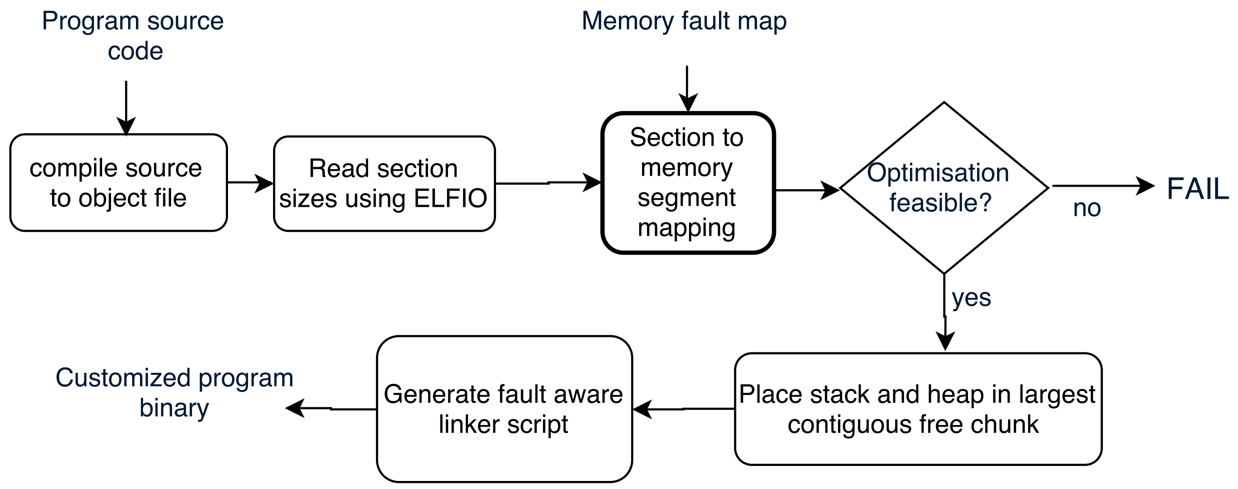

Program source code
Memory fault map
compile source to object file
Read section sizes using ELFIO
Section to memory segment mapping
Optimisation feasible?
no
FAIL
yes
Place stack and heap in largest contiguous free chunk
Generate fault aware linker script
Customized program binary

Figure 7.4: FaultLink procedure: given program source code and a memory fault map, produce a per-chip custom binary executable that will work in presence of known hard fault locations in the SPMs.

instruction memory. The chip's floorplan and test board are shown in Fig. 7.2. The locations of voltage-induced SRAM hard faults in the data SPM for one chip are shown in Fig. 7.3 as black dots. Its byte-level fault address map appears as follows.

```
0x200057D6
0x200086B4
...
0x2002142F
0x200247A9
```

Without further action, this chip would be useless at low voltage for running embedded applications; either the min-VDD would be increased, compromising energy, or the chip would be discarded entirely. We now describe how the FaultLink toolchain leverages the fault map to produce workable programs in the presence of potentially many hard faults.

7.4.2 Toolchain

FaultLink utilizes the standard GNU tools for C/C++ without modification. The overall procedure is depicted in Fig. 7.4. The programmer compiles their code into object files but does not

Table 7.1: Notation for Section-Packing Problem Formulation

Term	**Definition**
M	Set of program sections
N	Set of fault-free memory segments
m_i	Size of program section i in bytes
n_j	Size of memory segment j in bytes
y_j	1 if segment j contains at least one section; else 0
z_{ij}	1 if section i is mapped to segment j; else 0

proceed to link them. The code must be compiled using GCC's `-ffunction-sections` and `-fdata-sections` flags, which instruct GCC to place each subroutine and global variable into their own named sections in the ELF object files.[1] Our FaultLink tool then uses the ELFIO C++ library [394] to parse the object files and extract section names, sizes, etc. FaultLink then produces a customized binary for the given chip by solving the Section-Packing problem.

7.4.3 Fault-Aware Section-Packing

Section-Packing is a variant of the NP-complete Multiple Knapsacks problem. We formulate it as an optimization problem and derive an analytical approximation for the probability that a program's sections can be successfully packed into a memory containing hard faults.

7.4.3.1 Problem Formulation

Given a disjoint set of contiguous program sections M and a set of disjoint hard fault-free contiguous memory segments N, we wish to pack each program section into exactly one memory segment such that no sections overlap or are left unpacked. If we find a solution, we output the $M \rightarrow N$ mapping; otherwise, we cannot pack the sections (the program cannot accommodate that chip's fault map). An illustration of the Section-Packing problem is shown in Fig. 7.5, with the program sections on the top and fault-free memory regions on the bottom.

Let m_i be the size of program section i in bytes and n_j be the size of memory segment j, y_j be 1 if segment j contains at least one section, otherwise let it be 0, and z_{ij} be 1 if section i is mapped

[1]According to GCC documentation, this can potentially impact performance and code size. For all the benchmarks in this chapter, the impact to code size was less than 1% and there was no measurable impact to performance.

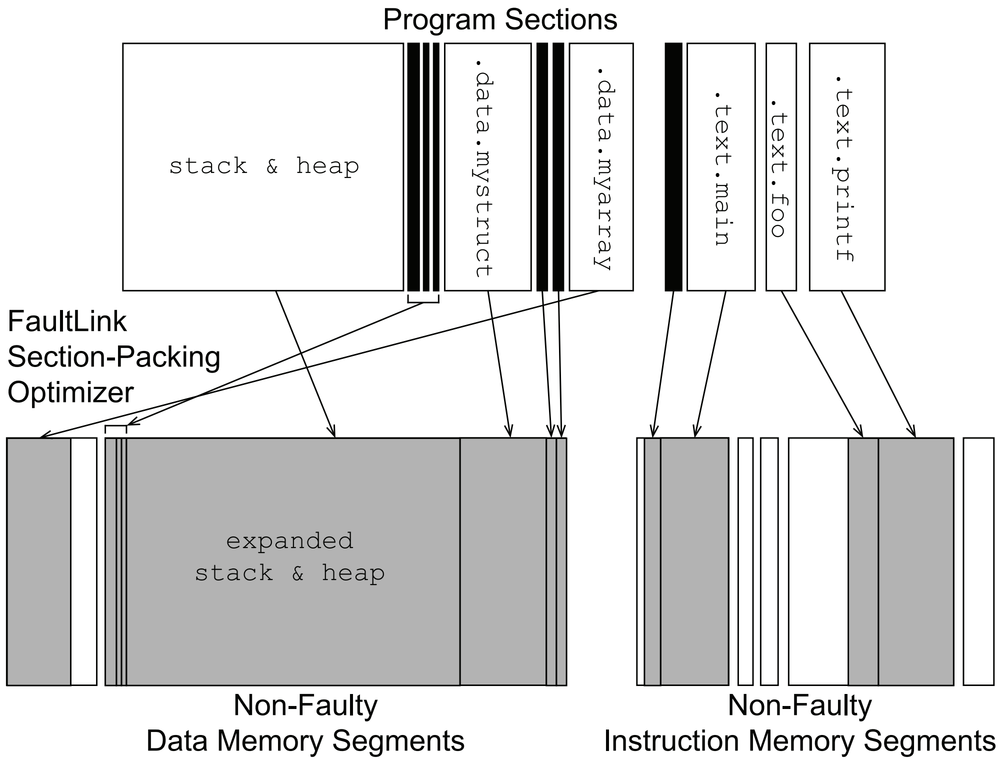


Figure 7.5: FaultLink attempts to pack contiguous program sections into contiguous disjoint segments of non-faulty memory.

to segment j, otherwise let it be 0. The notation just introduced is summarized in Table 7.1. The optimization problem is formulated as an integer linear program (ILP) as follows.

$$\textbf{Minimize:} \sum_{j \in N} y_j$$

Subject to:

$$\sum_{i \in M} m_i \cdot z_{ij} \leq n_j \cdot y_j \; \forall j \in N$$

$$\sum_{j \in N} z_{ij} = 1 \; \forall i \in M$$

$$z_{ij} = 0 \text{ or } 1 \; \forall i \in M; j \in N$$

$$y_j = 0 \text{ or } 1 \; \forall j \in N$$

We solve this ILP problem using CPLEX. We specifically include the objective that minimizes the number of packed segments because the solution naturally avoids memory regions that have higher fault densities. To pack any benchmark onto any fault map that we evaluated, CPLEX required no more than 14 seconds in the worst case; if a solution cannot be found or if there are few faults, typically FaultLink will complete much quicker.[2]

7.4.3.2 Analytical Section-Packing Estimation

We observe that the size of the maximum contiguous program section often comprises a significant portion of the overall program size. Most FaultLink section-packing failures occur when the largest program section is larger than all non-faulty memory segments. If this is true more generally, then we can estimate the probability of successful FaultLink section-packing.

Our analytical formulation is based on the probability distribution of the longest consecutive sequences of coin flips [395]. Let L_k be a random variable representing the length of the largest run of heads in k independent flips of a biased coin (with p as the probability of heads). The following equation is an approximation for the limiting behavior of L_k, i.e., the probability that longest run of heads is less than x and assuming $k(1-p) \gg 1$ [395]:

$$P(L_k < x) \approx e^{-p^{(x - \log_{p^{-1}}(k(1-p)))}}. \tag{7.1}$$

We now apply this formula to estimate the behavior of FaultLink. Let b be the i.i.d. bit-error-rate and s be the probability of no errors occurring in a 32-bit word, i.e., $s = (1-b)^{32}$. Let `size` be the memory size in bytes and $m_{\max}$ be the size in bytes of the largest contiguous program section. Using Eqn. 7.1, we plug in $p = s$, $k = \texttt{size}/4$, and $x = m_{\max}/4$ to approximate the probability of there *not* being a memory segment that is large enough to store the largest program section[3]:

$$P\left(L_{\texttt{size}/4} < \frac{m_{\max}}{4}\right) \approx e^{-s^{\left(\frac{m_{\max}}{4} - \log_{s^{-1}}\left(\frac{\texttt{size}}{4}(1-s)\right)\right)}}. \tag{7.2}$$

We will apply this equation later in the evaluation to estimate FaultLink yield and min-VDD over

[2]If a faster solution is needed, a greedy ILP relaxation can be used.

[3]The division by four is to convert the lengths of the sections and segments from bytes to 32-bit words.

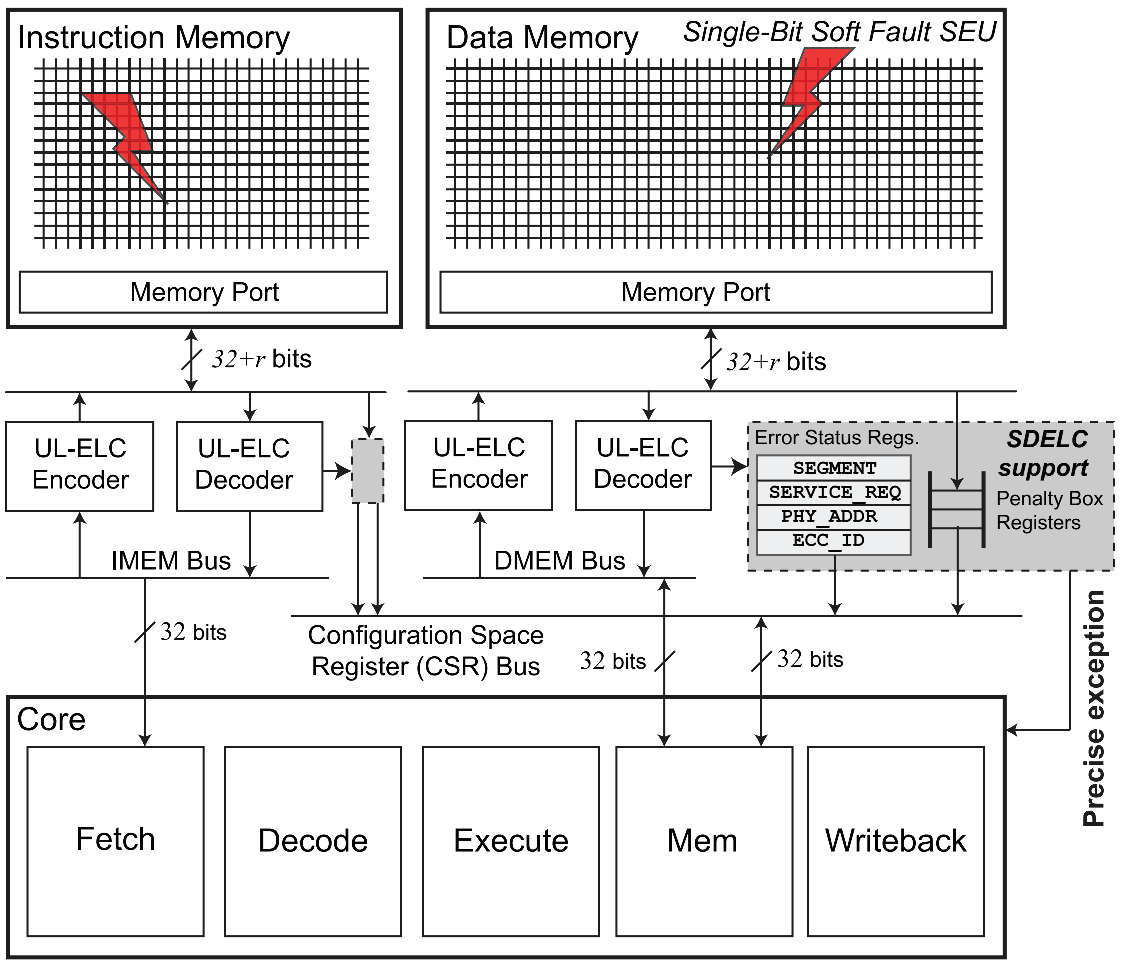


Figure 7.6: Architectural support for SDELC on an microcontroller-class embedded system with split on-chip instruction and data SPMs. The single-issue core has an in-order pipeline. We assume that hard faults are already protected against using FaultLink.

a theoretical population of faulty chips for each benchmark.[4]

7.5 SDELC: Dealing with Soft Faults

We describe the SDELC architecture, the concept of UL-ELC codes, and two SDELC recovery policies for instruction and data memory errors.

[4]If there are multiple large program sections, one could approximate the probability that they all fit using *k*th maximal spacing bounds [396].

7.5.1 Architecture

The SDELC architecture is illustrated in Fig. 7.6. Each memory has its own UL-ELC code. When a codeword containing a single-bit soft fault is read, the UL-ELC decoder detects and localizes the error to a specific chunk of the codeword and places error information in a *Penalty Box* register (shaded in gray in the figure). A precise exception is then generated, and software traps to a handler that implements the appropriate SDELC recovery policy for instructions or data, which we will discuss shortly.

Once the trap handler has decided on a candidate codeword for recovery, it must correctly commit the state in the system such that it appears *as if* there was no memory control flow disruption. For instruction errors, because the error occurred during a fetch, the program counter (`pc`) has not yet advanced. To complete the trap handler, we write back the candidate codeword to instruction memory. If it is not accessible by the load/store unit, one could use hardware debug support such as JTAG. We then return from the trap handler and re-execute the previously-trapped instruction, which will then cause the `pc` to advance and re-fetch the instruction that had been corrupted by the soft error. On the other hand, data errors are triggered from the memory pipeline stage by executing a load instruction. We write back the chosen candidate codeword to data memory to scrub the error, update the register file appropriately, and manually advance `pc` before returning from the trap handler.

7.5.2 Ultra-Lightweight Error-Localizing Codes (UL-ELC)

Localizing an error is more useful than simply detecting it. If we determine the error is from a *chunk* of length ℓ bits, there are only ℓ *candidate codewords* for which a single-bit error could have produced the received (corrupted) codeword.

A naïve way of localizing a single-bit error to a particular chunk is to use a trivial chunked parity code, i.e., we can assign a dedicated parity-bit to each chunk. However, this method is very inefficient because to create C chunks we need C parity bits: essentially, we have simply split up memory words into smaller pieces.

We create simple, custom *Ultra-Lightweight* ELCs (UL-ELCs) that – given r redundant parity bits – can localize any single-bit error to one of $C = 2^r - 1$ possible chunks. This is because there are $2^r - 1$ distinct non-zero columns that we can use to form the parity-check matrix $\mathbf{H}$ for our UL-ELC (for single-bit errors, the error syndrome is simply one of the columns of $\mathbf{H}$). To create a UL-ELC code, we first assign to each chunk a distinct non-zero binary column vector of length r bits. Then each column of $\mathbf{H}$ is simply filled in with the corresponding chunk vector. Note that r of the chunks will also contain the associated parity-bit within the chunk itself; we call these *shared chunks*, and they are precisely the chunks whose columns in $\mathbf{H}$ have a Hamming weight of 1. Since there are r shared chunks, there must be $2^r - r - 1$ *unshared chunks*, which each consist of only data bits. Shared chunks are unavoidable because the parity bits must also be protected against faults, just like the message bits.

UL-ELCs form a middle-ground between basic parity SED error-detecting codes (EDCs) and Hamming SEC ECCs. In the former case, $r = 1$, so we have a $C = 1$ monolithic chunk ($\mathbf{H}$ is a row vector of all ones). In the latter case, $\mathbf{H}$ uses each of the $2^r - 1$ possible distinct columns exactly once: this is precisely the $(2^r - 1, 2^r - r - 1)$ Hamming code. An UL-ELC code has a minimum distance of two bits by construction to support detection and localization of single-bit errors. Thus, the set of candidate codewords must also be separated from each other by a Hamming distance of exactly two bits.[5]

The two key properties of UL-ELC that do not apply to ELC codes in general are: *(i)* the length of the data message is independent of r, and *(ii)* each chunk can be an arbitrary length. The freedom to choose the length of the code and chunk sizes allow the UL-ELC design to be highly adaptable. Additionally, UL-ELC codes can offer SEC protection on up to $2^r - r - 1$ selected message bits by having the unshared chunks each correspond to a single data bit.

[5]Generally, a minimum codeword distance of two bits is required for SED, three bit-separation is needed for SEC, four bits are needed to correct any single-bit error and detect any double-bit error (SECDED), etc.

7.5.3 UL-ELC Example

For an example of an UL-ELC construction, consider the following **H** parity-check matrix with $r = 3$ and thus $N = 7$ chunks:

$$\mathbf{H} = \begin{array}{c} \\ \\ c_1 \\ c_2 \\ c_3 \end{array} \begin{array}{cccccccccccc} C_1 & C_2 & C_3 & C_4 & C_4 & C_5 & C_6 & C_6 & C_7 & C_5 & C_6 & C_7 \\ d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 & p_1 & p_2 & p_3 \\ \end{array}$$

$$\begin{array}{c|cccccccccccc} & C_1 & C_2 & C_3 & C_4 & C_4 & C_5 & C_6 & C_6 & C_7 & C_5 & C_6 & C_7 \\ & d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 & p_1 & p_2 & p_3 \\ \hline c_1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ c_2 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ c_3 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{array},$$

where d_i represents the ith data bit, p_j is the jth redundant parity bit, c_k is the kth parity-check equation, and C_l enumerates the distinct error-localizing chunk that a given bit belongs to. In this example, we have seven chunks. d_1, d_2, and d_3 each have the SEC property because no other bits are in their respective chunks. d_4 and d_5 make up an *unshared chunk* C_4 because no parity bits are included in C_4. The remaining data bits belong to *shared chunk* because each of them also includes at least one parity bit. Notice that any data or parity bits that belong to the same chunk C_l have identical columns of **H**. For instance, d_7, d_8, and p_2 all belong to C_6 and share the column value $[0;1;0]$.

A graphical representation of this example is shown in Fig.7.7. We see that the technique used to create a UL-ELC code can be envisioned as a Hamming-style code with pre-grouped data bits. As in the Hamming code, any single-bit error produces a syndrome that corresponds to a unique linear combination of check equations. For example, if the decoder produces the syndrome $\mathbf{s} = [0;1;1]$, then check equations 2 and 3 are not satisfied. Looking at Fig. 7.7, it is apparent that if we assume only a single-bit error has occurred, then the error must be in the chunk containing d_4 and d_5. This example is for demonstration purposes only; in the case presented here with nine data bits, only four parity bits are required for a shortened Hamming code.

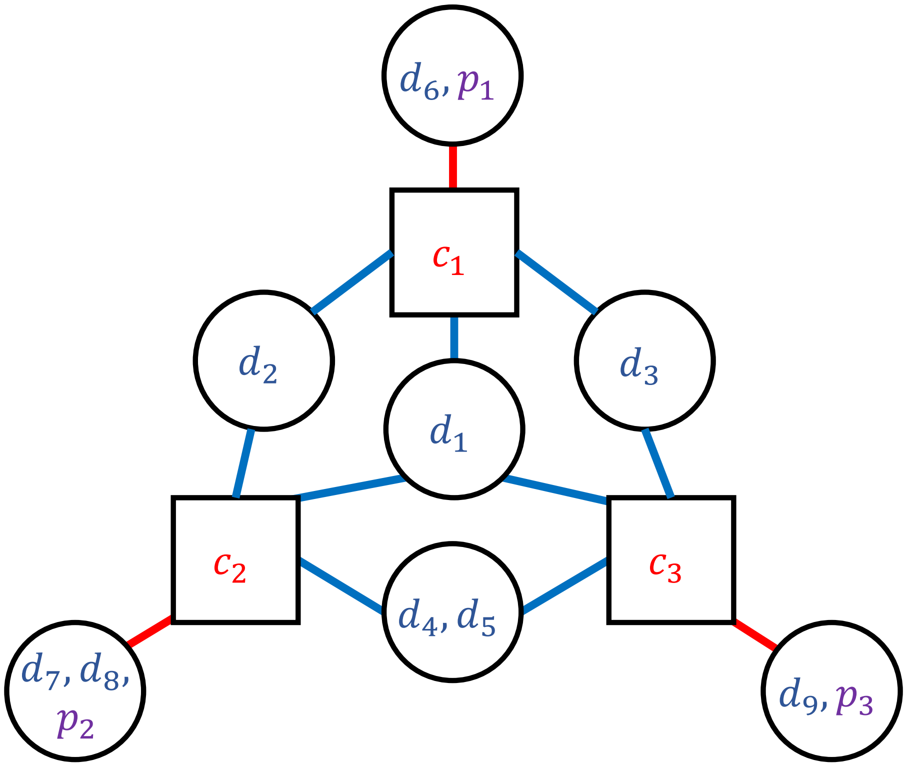


Figure 7.7: A visualization of our example UL-ELC code with seven chunks. Each circle represents a chunk, and each square represents a check equation. Each message and parity bit belongs to at most one chunk. The blue and red lines connect the check equations to unshared and shared chunks, respectively.

7.5.4 Recovering SEUs in Instruction Memory

We describe an UL-ELC construction and recovery policy for dealing with single-bit soft faults in instruction memory. The code and policy are jointly crafted to exploit SI about the ISA itself. Our implementation targets the open-source and free 64-bit RISC-V (RV64G) ISA [310], but the approach is general and could apply to any other fixed-length or variable-length RISC or CISC ISA. Note that although RISC-V is actually a little-endian architecture, for sake of clarity we use big-endian in this chapter.

Table 7.2: Proposed 7-Chunk UL-ELC Construction with $r = 3$ for Instruction Memory (RV64G ISA v2.0 [310])

bit $\rightarrow$	`31` `27`	`26` `25`	`24` `20`	`19` `15`	`14` `12`	`11` `7`	`6` `0`	`-1` `-3`
Type-`U`	`imm[31:12]`					`rd`	`opcode`	`parity`
Type-`UJ`	`imm[20|10:1|11|19:12]`					`rd`	`opcode`	`parity`
Type-`I`	`imm[11:0]`			`rs1`	`funct3`	`rd`	`opcode`	`parity`
Type-`SB`	`imm[12|10:5]`		`rs2`	`rs1`	`funct3`	`imm[4:1|11]`	`opcode`	`parity`
Type-`S`	`imm[11:5]`		`rs2`	`rs1`	`funct3`	`imm[4:0]`	`opcode`	`parity`
Type-`R`	`funct7`		`rs2`	`rs1`	`funct3`	`rd`	`opcode`	`parity`
Type-`R4`	`rs3`	`funct2`	`rs2`	`rs1`	`funct3`	`rd`	`opcode`	`parity`

Chunk	C_1 (shared)	C_2 (shared)	C_3 (shared)	C_4	C_5	C_6	C_7	C_3	C_2	C_1
Parity-	00000	00	11111	00000	111	11111	1111111	1	0	0
Check	00000	11	00000	11111	000	11111	1111111	0	1	0
H	11111	00	00000	11111	111	00000	1111111	0	0	1

7.5.4.1 Code Construction

Our UL-ELC construction for instruction memory has seven chunks that align to the finest-grain boundaries of the different fields in the RISC-V codecs. These codecs, the chunk assignments, and the complete parity-check matrix **H** are shown in Table 7.2. The bit positions -1, -2, and -3 correspond to the three parity bits that are appended to a 32-bit instruction in memory. The `opcode`, `rd`, `funct3`, and `rs1` fields are the most commonly used – and potentially the most critical – among the possible instruction encodings, so we assign each of them a dedicated chunk that is unshared with the parity bits. The fields which vary more among encodings are assigned to the remaining three shared chunks, as shown in the figure. The recovery policy can thus distinguish the impact of an error in different parts of the instruction. For example, when a fault affects a shared chunk C_1, the fault is either in one of the five MSBs of the instruction, or in the last parity bit. Conversely, when a fault is localized to unshared chunk C_7 in Table 7.2, the UL-ELC decoder can be certain that the `opcode` field has been corrupted.

Consider another example with a fault in the unshared chunk C_6 that guards the `rd` destination register address field for most instruction codecs. Suppose a fault.bit 7 (the least-significant bit of chunk C_6/`rd`) is flipped by Assume the original instruction stored in memory was `0x0000beef`,[6] which decodes to the assembly code `jal t4, 0xb000`. The 5-bit `rd` field is protected with our UL-ELC construction using a dedicated unshared chunk C_6. Thus, the candidate messages are the following instructions.

```
<0x0000b66f> jal a2, 0xb000
<0x0000ba6f> jal s4, 0xb000
<0x0000beef> jal t4, 0xb000
<0x0000bc6f> jal s8, 0xb000
<0x0000bf6f> jal t5, 0xb000
```

The above instructions are all Hamming distances of two bits apart. Our instruction recovery policy can decide which destination register is most likely for the `jal` instruction based on program statistics collected a priori via static or dynamic profiling (the SI). We discuss this procedure next.

[6]RISC-V is a little-endian architecture, but for sake of clarity, we use big-endian here.

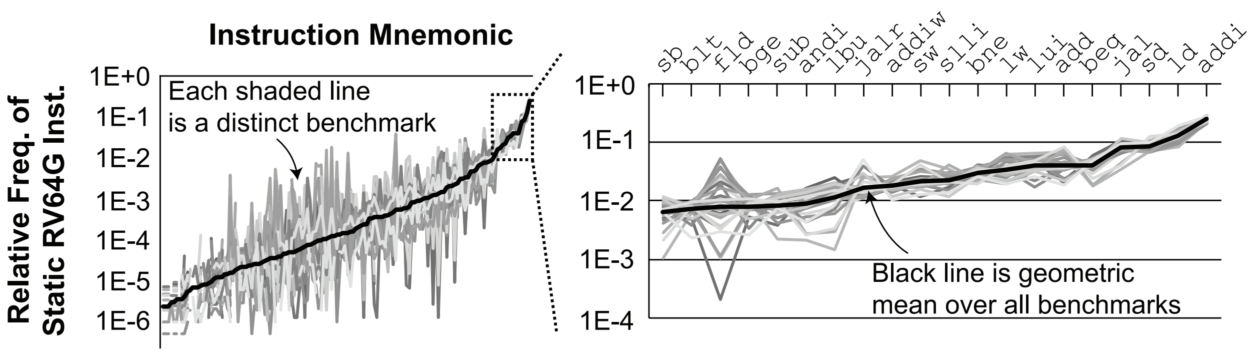


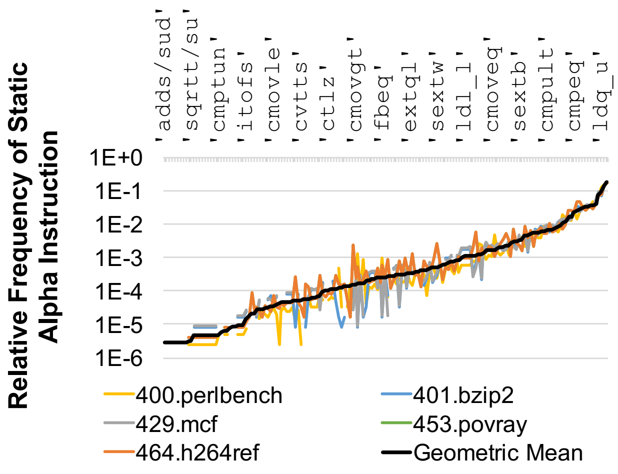
Figure 7.8: The relative frequencies of dynamic instructions roughly follow power law distributions. Results shown are for RISC-V.

7.5.4.2 Observations on Instruction Side Information

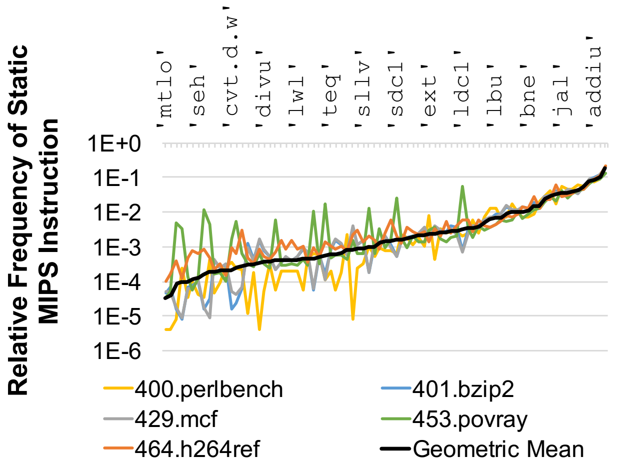
Using the instruction-based UL-ELC construction, we will apply a heuristic recovery policy that is based on SI about program code. The SI is based on three observations.

Observation 1. As shown in Table 7.3, we find that in three distinct RISC ISAs, most bit patterns decode to illegal instructions. This fact can be used to dramatically improve the chances of a successful SDELC recovery. For example, in RISC-V, whenever a single-bit fault affects the `opcode` or – when applicable – the `funct3`, `funct2`, and `funct7` function-code fields, many candidates are likely to be illegal.

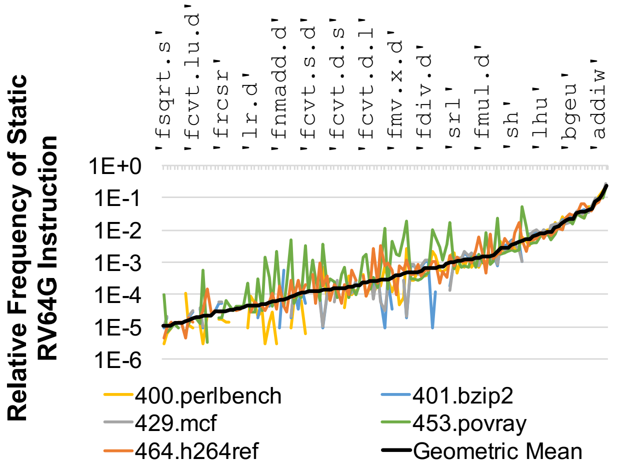
Observation 2. We find that the relative frequencies of legal instructions follow power-law distributions. Fig. 7.8 depicts the relative frequencies of dynamic RISC-V instructions for 20 SPEC CPU2006 benchmarks. Clearly, the most-frequent 20 instructions occur far more often and in a more consistent rank ordering than others. This can be used to favor more common instructions during heuristic recovery. Notice that each benchmark follows an individual power-law distribution of its own, which makes it difficult to aggregate the frequencies across benchmarks. Fortunately, the variability of rank orderings and relative frequencies of the *top 20* most common instruction mnemonics (which comprise 87% of all instructions) is small across the benchmarks. Therefore, noise in the rank orderings of the infrequent instructions is unimportant. Similar trends are evident for static instructions in three different ISAs, is indicated by Fig. 7.9. Typically, we find that there is less benchmark-to-benchmark variability in the relative frequencies and rank ordering of instructions. In this work, we use the dynamic distribution for our evaluations.



(a) Alpha



(b) MIPS



(c) RISC-V

Figure 7.9: Relative frequencies of static instructions for three ISAs.

Table 7.3: Illegal Instructions in Different RISC ISAs

Instruction Set Architecture (ISA)	**Prct. of 32-Bit Patterns that are Illegal Instructions**
RV64G	92.33%
MIPS1-R3000	72.44%
Alpha	66.87%

Observation 3. In many RISC-V instruction codecs, the most-significant bits (MSBs) represent immediate values (as shown in Table 7.3). Accordingly, these MSBs are usually low-magnitude signed integers. In other codecs, the MSBs represent 0-dominant function codes. We can exploit this trend by preferring candidates that have long pads of 0s or 1s.

7.5.4.3 Recovery Policy

The instruction recovery policy used in our evaluation consists of three steps.

- Step 1. We apply a software-implemented instruction decoder to filter out any candidate messages that are illegal instructions.
- Step 2. Next, we estimate the probability of each valid message using a small pre-computed lookup table that contains the relative frequency that each instruction appears.
- Step 3. We choose the instruction that is most common according to our SI lookup table. In the event of a tie, we choose the instruction with the longest leading-pad of 0s or 1s.

7.5.4.4 Possible Improvements to Recovery Policy

Our recovery policy could be improved to take advantage of more detailed side information when available. We find that certain *combinations* of mnemonic and register operands are much more common than others, as indicated by Fig. 7.10. For instance, the `addi` mnemonic is generally common on its own, but occurs especially often when the `a5` register is one of its operands. This is largely a consequence of the ISA, ABI, and compiler design. Many combinations never occur at all in a program. Because storing a 2D lookup table of joint mnemonic-register frequency is costly for embedded systems, we do not evaluate this approach further.

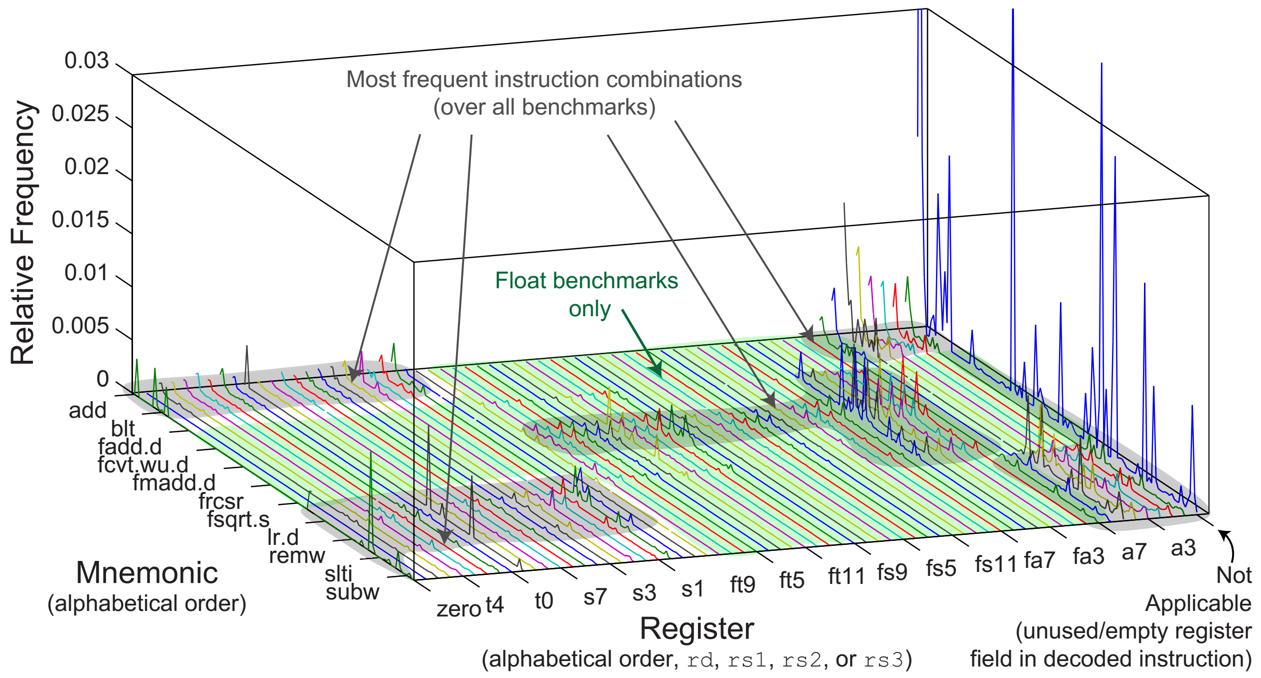


Figure 7.10: The joint relative frequency distribution of RV64G mnemonics and registers is sparse.

7.5.5 Recovering SEUs in Data Memory

In general-purpose embedded applications, data may come in many different types and structures. Because there is no single common data type and layout in memory, we propose to simply use evenly-spaced UL-ELC constructions and grant the software trap handler additional control about how to recover from errors, similar to the general idea from SuperGlue [397].

We build SDELC recovery support into the embedded application as a small C library. The application can push and pop custom SDELC error handler functions onto a registration stack. The handlers are defined within the scope of a subroutine and optionally any of its callees and can define specific recovery behaviors depending on the context at the time of error. Applications can also enable and disable recovery at will.

When the application does not disable recovery nor specify a custom behavior, all data memory errors are recovered using a default error handler implemented by the library. The default handler computes the average Hamming distance to nearby data in the same 64-byte chunk of memory (similar to taking the intra-cacheline distance in cache-based systems). The candidate with the minimum average Hamming distance is selected. This policy is based on the observation that spatially-local and/or temporally-local data tends to also be correlated, i.e., it exhibits *value locality* [306] that has also been used in numerous works for cache and memory compression [300, 301, 303]. The Hamming distance is a good measure of data correlation, as shown later in Fig. 7.16 during the evaluation.

The application-defined error handler can specify recovery rules for individual variables within the scope of the registered subroutine. They include globals, heap, and stack-allocated data. This is implemented by taking the runtime address of each variable requiring special handling. For instance, an application may wish critical data structures to never be recovered heuristically; for these, the application can choose to force a crash whenever a soft error impacts their memory addresses. In other scenarios, the application can make smarter decisions about the optimal recovery target for a corrupted variable in memory. For instance, suppose a loop index `int i` is corrupted, but we know at compile-time that it only increments from 0 to 99. Then any candidate that resolves to an integer less than 0 or greater than 99 can be definitively ruled out.

The SDELC library support can increase system reliability, but the programmer is required to spend effort annotating source code for error recovery. This is similar to annotation-based approaches taken by others for various purposes [1, 110, 116, 266, 314, 398]. However, a limitation is that programmer effort spent annotating regions of code and specific variables are not necessarily reflected in the probability that those memory locations are in error. Thus, the default Hamming distance-based policy is critical to the average probability of recovery.

7.6 Evaluation

We evaluate FaultLink and SDELC primarily in terms of their combined ability to proactively avoid hard faults and then heuristically recover from soft faults in SPMs.

7.6.1 Hard Fault Avoidance using FaultLink

We first demonstrate how applications can run on real test chips at low voltage with many hard faults in on-chip memory using FaultLink, and then evaluate the yield benefits at low voltage for a

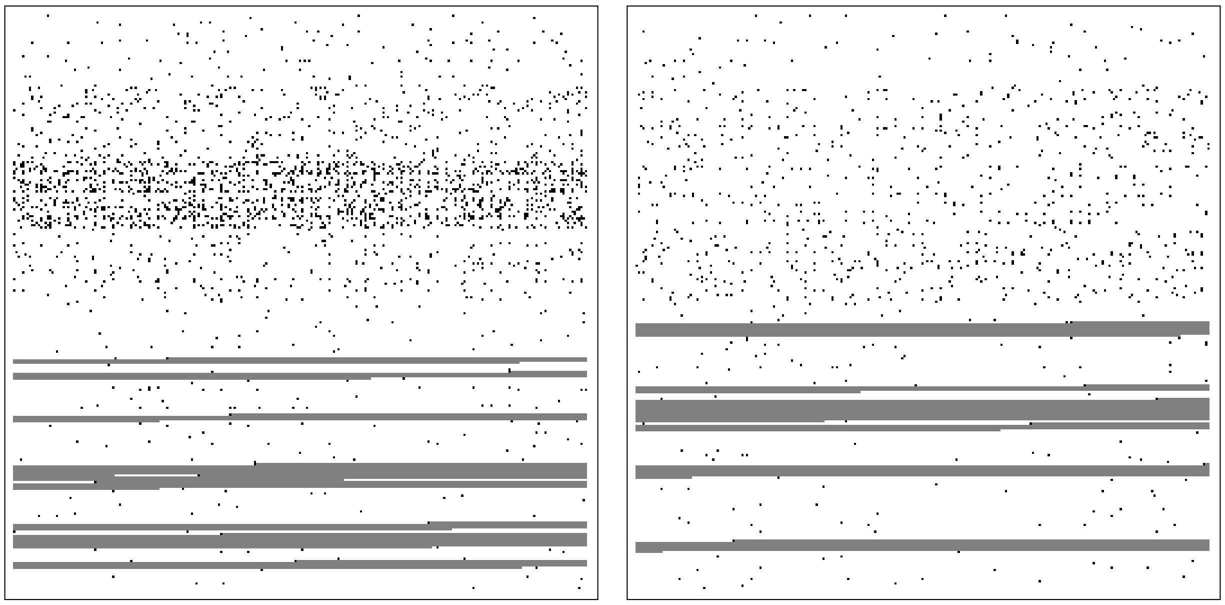

(a) Chip 1 (b) Chip 2

Figure 7.11: Result from applying FaultLink to the `sha` benchmark for two real test chips' 64 KB instruction memory at 650 mV. Black dots represent faulty byte locations, while gray regions represent program sections that have been packed into non-faulty segments (white regions).

synthetic population of chips.

7.6.1.1 Voltage Reduction on Real Test Chips

We first apply FaultLink to a set of small embedded benchmarks that we build and run on eight of our microcontroller-class 45nm *"real test chips."* Each chip has 64 KB of instruction memory and 176 KB of data memory. The five benchmarks are `blowfish` and `sha` from the mibench suite [399] as well as `dhrystone`, `matmulti` and `whetstone`. We characterized the hard voltage-induced fault maps of each test chip's SPMs in 50 mV increments from 1 V (nominal VDD) down to 600 mV using March-SS tests [187] and applied FaultLink to each benchmark for each chip individually at every voltage. Note that the standard C library provided with the ARM toolchain uses split function sections, i.e., it does not have a monolithic `.text` section. For each FaultLink-produced binary that could be successfully packed, we ran them to completion on the real test chips. The FaultLink binaries were also run to completion on a simulator to verify that no hard fault locations are ever accessed.

FaultLink-packed instruction SPM images of the `sha` benchmark for two chips are shown in Fig. 7.11 with a runtime VDD of 650 mV. There were about 1000 hard-faulty byte locations in each SPM (shown as black dots). Gray regions represent `sha`'s program sections that were mapped into non-faulty segments (white areas).

We observe that FaultLink produced a unique binary for each chip. Unlike a conventional binary, the program code is not contiguous in either chip because the placements vary depending on the actual fault locations. In all eight test chips, we noticed that lower addresses in the first instruction SPM bank are much more likely to be faulty at low voltage, as seen in Fig. 7.11. This could be caused either by the design of the chip's power grid, which might have induced a voltage imbalance between the two banks, or by within-die/within-wafer process variation. Chip 1 (Fig. 7.11a) also appears to have a cluster of weak rows in the first instruction bank. Because FaultLink chooses a solution with the sections packed into as few segments as possible, we find that the mapping for both chips prefers to use the second bank, which tends to have larger segments.

We achieved an average min-VDD of 700 mV for the real test chips. This is a reduction of 125 mV compared to the average non-faulty min-VDD of 825 mV, and 300 mV lower than the official technology specification of 1 V.[7] FaultLink did not require more than 14 seconds on our machine to optimally section-pack any program for any chip at any voltage.

7.6.1.2 Yield at Min-VDD for Synthetic Test Chips

To better understand the min-VDD and yield benefits of FaultLink using a wider set of benchmarks and chip instances, we created a series of randomly-generated *synthetic fault maps*. For instruction and data SPM capacities of 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, and 4 MB, we synthesized 100 fault maps for each in 10 mV increments for a total of 700 *"synthetic test chips."* We used detailed Monte Carlo simulation of SRAM bit-cell noise margins in the corresponding 45 nm technology. Six more benchmarks were added from the AxBench approximate computing C/C++ suite [314] that are too big to fit on the real test chips: `blackscholes`, `fft`, `inversek2j`, `jmeint`,

[7]We expect the actual voltage savings to be greater, but we were only able to characterize the fault maps in 50 mV increments. Statistically, however, the measured non-faulty min-VDD would increase slightly if we were able to characterize more than eight chips.

`jpeg`, and `sobel`. These AxBench benchmarks were compiled for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [310] and privileged specification v1.7 using the official tools. This is because unlike the standard C library for our ARM toolchain, the library included with the RISC-V toolchain has a monolithic `.text` section. This allows us to consider the impact of the library sections on min-VDD.

The expected min-VDD for 99% chip yield across 100 synthetic chip instances for seven memory capacities is shown in Fig. 7.12. The vertical bars represent our analytical estimates calculated using Eqn. 7.2. The red line represents the empirical worst case out of 100 synthetic test chips, while the blue line is the lowest non-faulty voltage in the worst case of the 100 chips. Finally, the green line represents the nominal VDD of 1 V.

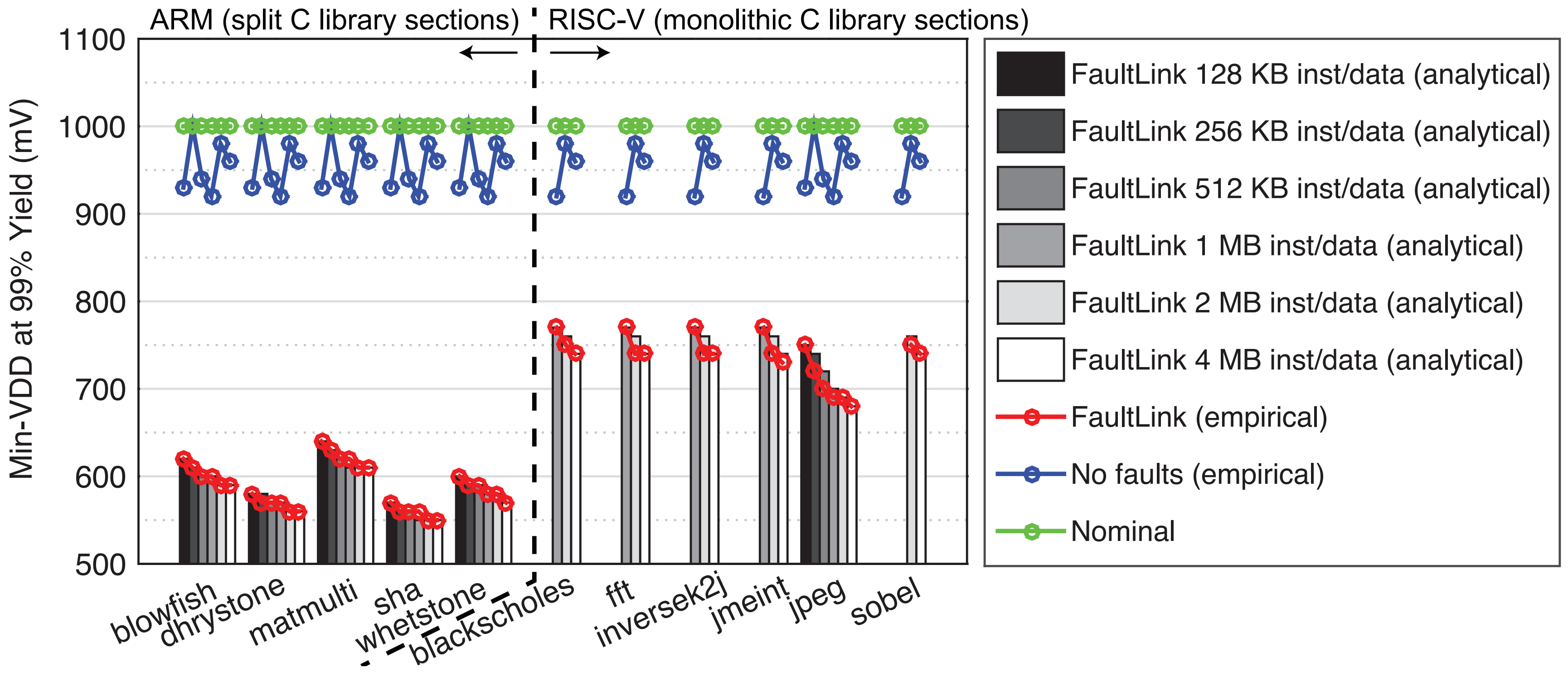


Figure 7.12: Achievable min-VDD for FaultLink at 99% yield. Bars represent the analytical lower bound from Eqn. 7.2 and circles represent our actual results using Monte Carlo simulation for 100 synthetic fault maps.

The expected min-VDD for 99% chip yield across 100 synthetic chip instances for seven memory capacities is shown in Fig. 7.12. The vertical bars represent our analytical estimates calculated using Eqn. 7.2. The red line represents the empirical worst case out of 100 synthetic test chips, while the blue line is the lowest non-faulty voltage in the worst case of the 100 chips. Finally, the green line represents the nominal VDD of 1 V.

FaultLink reduces min-VDD for the synthetic test chips at 99% yield by up to 450 mV with respect to the nominal 1 V and between 370 mV and 430 mV with respect to the lowest non-faulty voltage. All but `jpeg` from the AxBench suite were too large to fit in the smaller SPM sizes (hence the "missing" bars and points). When the memory size is over-provisioned for the smaller programs, min-VDD decreases moderately because the segment size distribution does not have a strong dependence on the total memory size.

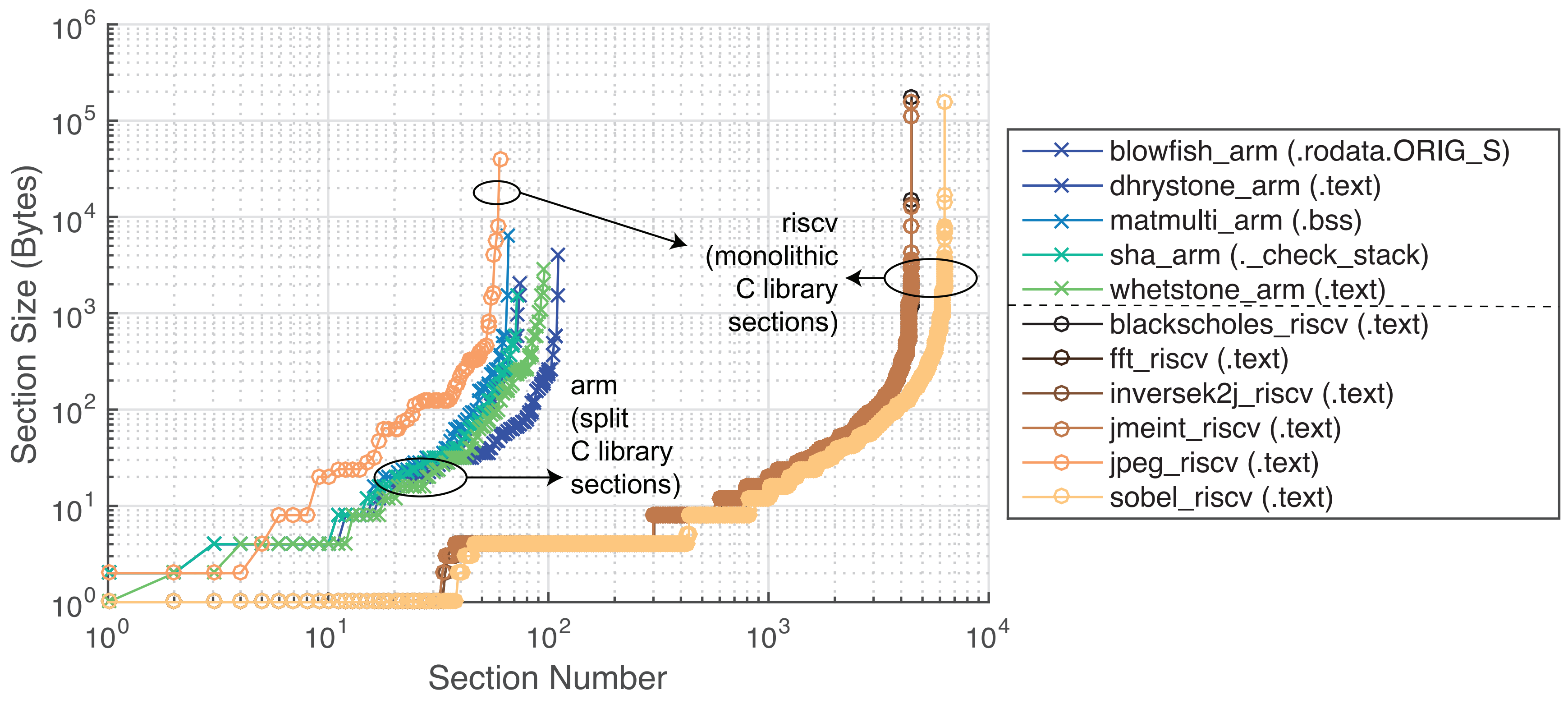


Figure 7.13: Distribution of program section sizes. Packing the largest section into a non-faulty contiguous memory segment is the most difficult constraint for FaultLink to satisfy and limits min-VDD.

The voltage-scaling limits are nearly always determined by the length of the longest program section, which must be packed into a contiguous fault-free memory segment. This is strongly indicated by the close agreement between the empirical min-VDDs and the analytical estimates, the latter of which had assumed the longest program section is the cause of section-packing failure.

To examine this further, the program section size distribution for each benchmark is depicted in Fig. 7.13. The name of the largest section is shown in the legend for each benchmark.

We observe all distributions have long tails, i.e., most sections are very small but there are a few sections that are much larger than the rest. We confirm that the largest section for each benchmark – labeled in the figure legend – is nearly always the cause of failure for the FaultLink section-packing algorithm at low voltage when many faults arise. Recall that the smaller ARM-compiled benchmarks have split C library function sections, while the AxBench suite that was compiled for RISC-V has a C library with a monolithic `.text` section; we observe that the latter RISC-V benchmarks have significantly longer section-size tails than the former benchmarks. This is why the AxBench suite does not achieve the lowest min-VDDs in Fig. 7.12. Notice that program size is not a major factor: `jpeg` for RISC-V is similar in size to the ARM benchmarks, but it still does not match their min-VDDs. If the RISC-V standard library had used split function sections, the AxBench min-VDDs would be significantly lower.

FaultLink does not require any hardware changes; thus, energy-efficiency (voltage reduction) and cost (yield at given VDD) for IoT devices can be considerably improved.

7.6.2 Soft Fault Recovery using SDELC

SDELC guards against unpredictable soft faults at run-time that cannot be avoided using FaultLink. To evaluate SDELC, Spike was modified to produce representative memory access traces of all 11 benchmarks as they run to completion. Each trace consists of randomly-sampled memory accesses and their contents. We then analyze each trace offline using a MATLAB model of SDELC. For each workload, we randomly select 1000 instruction fetches and 1000 data reads from the trace and exhaustively apply all possible single-bit faults to each of them. Because FaultLink has already been applied, there is never an intersection of both a hard and soft fault in our experiments.

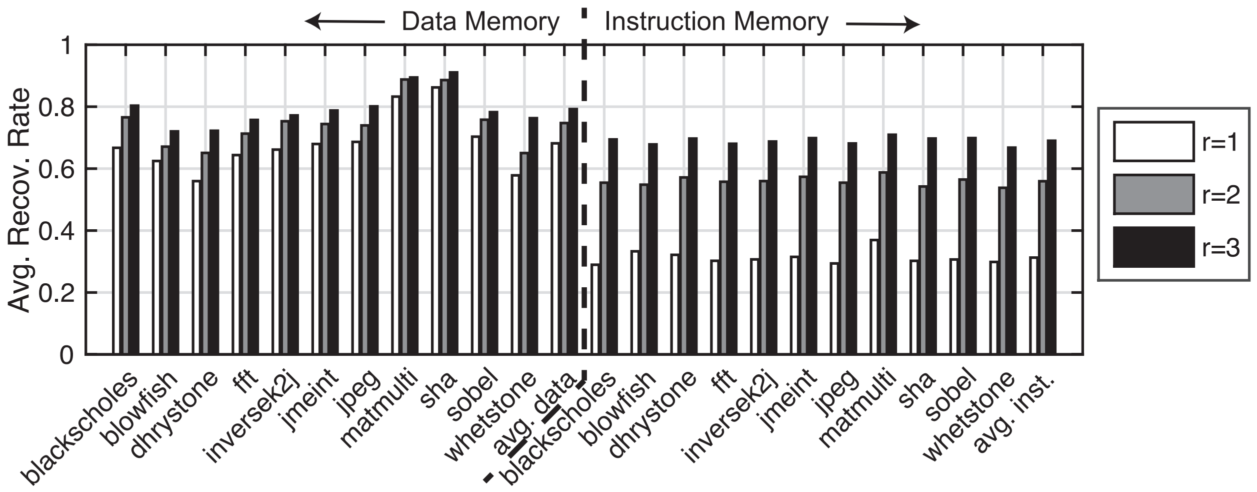


Figure 7.14: Average rate of recovery using SDELC from single-bit soft faults in data and instruction memory. Benchmarks have already been protected against known hard fault locations using FaultLink. r is the number of parity bits in our UL-ELC construction.

We evaluate SDELC recovery of the random soft faults using three different UL-ELC codes ($r = 1, 2, 3$). Recall that the $r = 1$ code is simply a single parity bit, resulting in 33 candidate codewords. (For basic parity, there are 32 message bits and one parity bit, so there are 33 ways to have had a single-bit error.) For the data memory, the UL-ELC codes were designed with the chunks being equally sized: for $r = 2$, there are either 11 or 12 candidates depending on the fault position (34 bits divided into three chunks), while for $r = 3$ there are always five candidates (35 bits divided into seven chunks). For the instruction memory, chunks are aligned to important field divisions in the RV64G ISA. Chunks for the $r = 2$ UL-ELC construction match the fields of the Type-U instruction codecs (the `opcode` being the unshared chunk). Chunks for the $r = 3$ UL-ELC code align with fields in the Type-R4 codec (as presented in Table 7.2). A *successful recovery* for SDELC occurs when the policy corrects the error; otherwise, it fails by accidentally mis-correcting the error.

7.6.2.1 Overall Results

The overall SDELC results are presented in Fig. 7.14. The recovery rates are relatively consistent over each benchmark, especially for instruction memory faults, providing evidence of the general efficacy of SD-ELC. One important distinction between the memory types is the sensitivity to the

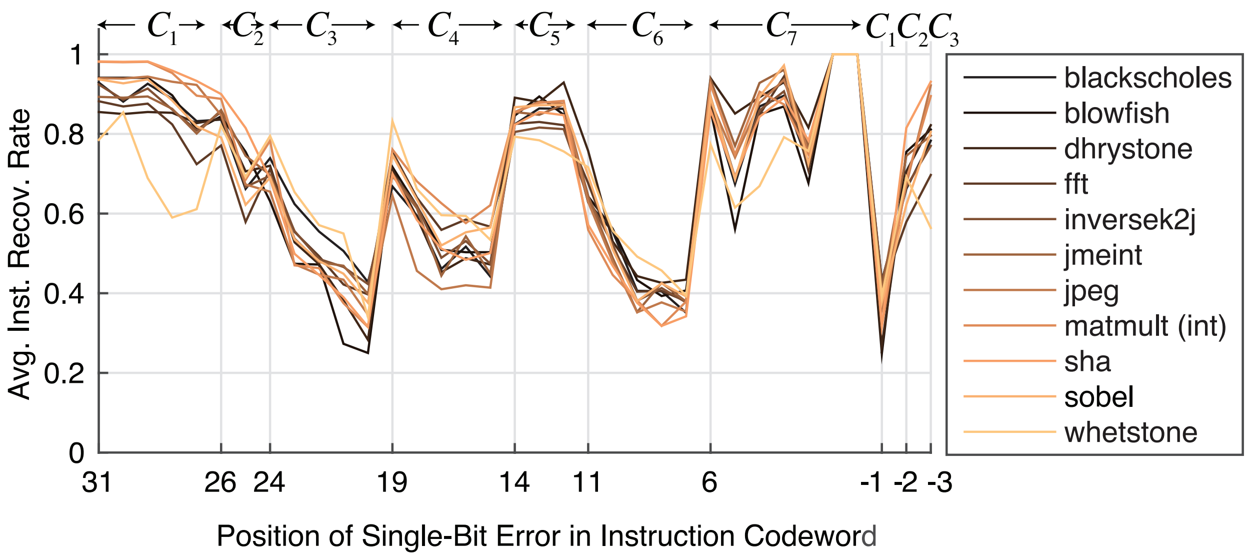


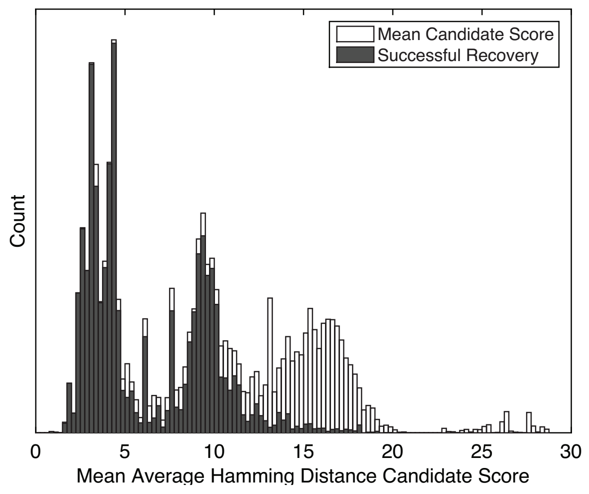
Figure 7.15: Sensitivity of SDELC instruction recovery to the actual position of the single-bit fault with the $r = 3$ UL-ELC construction.

number r of redundant parity bits per message. For the data memory, the simple $r = 1$ parity yielded surprisingly high rates of recovery using our policy (an average of 68.2%). Setting r to three parity bits increases the average recovery rate to 79.2% thanks to fewer and more localized candidates to choose from. On the other hand, for the instruction memory, the average rate of recovery increased from 31.3% with a single parity bit to 69.0% with three parity bits.

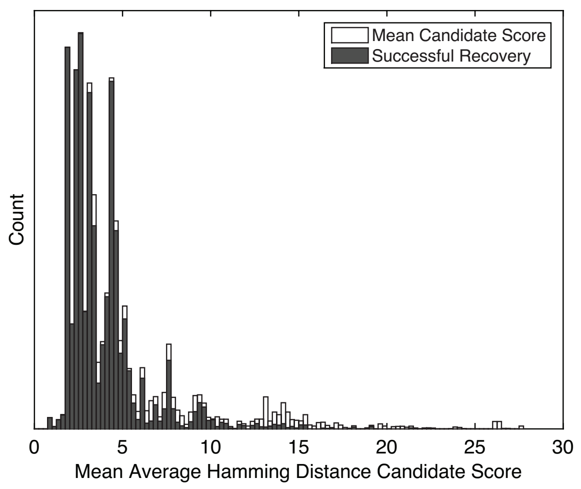
These results are a significant improvement over a guaranteed system crash as is traditionally done upon error detection using single-bit parity. Moreover, we achieve these results using no more than half the overhead of a Hamming SEC code, which can be a significant cost savings for small IoT devices. Based on our results, we recommend using $r = 1$ parity for data, and $r = 3$ UL-ELC constructions to achieve 70% recovery for both memories with minimal overhead. Next, we analyze the instruction and data recovery policies in more detail.

7.6.2.2 Recovery Policy Analysis

The average instruction recovery rate as a function of bit error position for all benchmarks is shown in Fig. 7.15. Error positions -1, -2, and -3 correspond to the three parity bits in our UL-ELC construction from Table 7.2.



(a) `blowfish`



(b) `sha`

Figure 7.16: Sensitivity of SDELC data recovery to the mean candidate Hamming distance score for two benchmarks and $r = 1$ parity code.

We observe that the SDELC recovery rate is highly dependent on the erroneous chunk. For example, errors in chunk C_7 – which protects the RISC-V `opcode` instruction field – have high rates of recovery because the power-law frequency distributions of legal instructions are a very strong form of side information. Other chunks with high recovery rates, such as C_1 and C_5, are often (part of) the `funct2`, `funct7`, or `funct3` conditional function codes that similarly leverage the power-law distribution of instructions. Moreover, many errors that impact the `opcode` or function codes cause several candidate codewords to decode to illegal instructions, thus filtering the number of possibilities that our recovery policy has to consider. For errors in the chunks that often correspond to register address fields (C_3, C_4, and C_6), recovery rates are less because the side information on register usage by the compiler is weaker than that of instruction relative frequency. However, errors towards the most-significant bits within these chunks recover more often than the least-significant bits because they can also correspond to immediate operands. Indeed, many immediate operands are low-magnitude signed or unsigned integers, causing long runs of 0s or 1s to appear in encoded instructions. These cases are more predictable, so we recover them frequently, especially for chunk C_1 which often represents the most-significant bits of an encoded immediate value.

The sensitivity of SD-ELC data recovery to the mean candidate Hamming distance score for two benchmarks is shown in Fig.7.16. White bars represent the relative frequency that a particular

Hamming distance score occurs in our experiments. The overlaid gray bars represent the fraction of those scores that we successfully recovered using our policy.

When nearby application data in memory is correlated, the mean candidate Hamming distance is low, and the probability that we successfully recover from the single-bit soft fault is high using our Hamming distance-based policy. Because applications exhibit spatial, temporal, and value locality [306] in memory, we thus recover correctly in a majority of cases. On the other hand, when data has very low correlation – essentially random information — SD-ELC does not recover any better than taking a random guess of the bit-error position within the localized chunk, as expected.

7.7 Discussion

Performance overheads. FaultLink does not add any performance overheads because it is purely a link-time solution, while its impact on code size is less than 1%. SDELC recovery of soft faults, however, requires about 1500 dynamic instrutions (a few μs on a typical microcontroller), although it varies depending on the specific recovery action taken and the particular UL-ELC code. However, for low-cost IoT devices that are likely to be operated in low-radiation environments with only occasional soft faults, the performance overhead is not a major concern. Simple recovery policies could be implemented in hardware, but then software-defined flexibility and application-specific support would be unavailable.

Memory reliability binning. FaultLink could bring significant cost savings to both IoT manufacturers and IoT application developers throughout the lifetime of the devices. Manufacturers could sell chips with hard defects in their on-chip memories to customers instead of completely discarding them, which increases yield. Customers could run their applications on commodity devices with or without hard defects at lower-than-advertised supply voltages to achieve energy savings. Fault maps for each chip at typical min-VDDs are small (bytes to KBs) and could be stored in a cloud database or using on-board flash. Several previous works have proposed heterogeneous reliability for approximate applications to reduce cost [261, 350, 351, 400].

Coping with aging and wearout using FaultLink. Because IoT devices may have long lifetimes, aging becomes a concern for the reliability of the device. Although explicit SPM wearout patterns cannot be predicted in advance, fault maps could be periodically sampled using BIST and uploaded to the cloud. Because IoT devices by definition already require network connectivity for their basic functionality and to support remote software updates and patching of security vulnerabilities, it is not disruptive to add remote FaultLink support to adapt to aging patterns. Because running FaultLink remotely takes just a few seconds, customers would not be affected any worse than the downtime already imposed by routine software updates and the impact on battery life would be minimum.

Risk of SDCs from SDELC. SDELC introduces a risk of mis-correcting single-bit soft faults that cannot be avoided without using a full Hamming SEC code. However, for low-cost IoT devices running approximation-tolerant applications, SDELC reduces the parity storage overhead by up to $6\times$ compared to Hamming while still recovering most single-bit faults. Similar to observations by others [315], in our experiments, we found that no more than 7.2% of all single-bit instruction faults and 2.3% of data faults result in an intolerable silent data corruption (SDC), i.e., more than 10% output error [314]. The rest of the faults are either successfully corrected, benign, or cause crashes/hangs. The latter are no worse than crashes from single-bit parity detection, which is common on state-of-the-art devices that have already been designed and deployed in the field. Their reliability can be improved with remote software updates to avoid hard faults at low voltage using FaultLink and recover soft faults using a small SDELC C library along with their existing parity.

Directions for future work. The FaultLink and SDELC approaches can be further extended. In this chapter, we split monolithic program sections such as `.text` and `.data` into smaller sections and packed them into non-faulty memory segments. However, at high fault rates, the largest program section often becomes the limiting factor when there is no segment large enough to fit it. One could extend FaultLink to accommodate hard faults within packed sections. For instruction memory, one approach could be to insert unconditional jump instructions to split up basic blocks, similar to a recent cache-based approach [401]. For data memory, one could use smaller split stacks [402] and design a fault-aware `malloc()`. For SDELC, one could design more sophisticated recovery policies using stronger forms of side information, and use profiling methods to automatically annotate program regions that are likely to experience faults and/or are approximation-tolerant.

7.8 Conclusion

We proposed FaultLink and SDELC, two complementary techniques – together referred to as ViFFTo – that improve memory resiliency for IoT devices in the presence of hard and soft faults. FaultLink tailors a given program binary to each individual embedded memory chip on which it is deployed. This improves both device yield by avoiding manufacturing defects and saves runtime energy by accounting for variation-induced parametric failures at low supply voltage. Meanwhile, SDELC implements low-overhead heuristic error correction to cope with random single-event upsets in memory without the higher area and energy costs of a full Hamming code. Directions for future work include designing a FaultLink-compatible remote software update mechanism for IoT devices in the field and supporting new failure modes of emerging non-volatile memories with SDELC.

CHAPTER 8

Conclusion

This chapter reviews the contributions of each chapter, outlines directions for future work, and provides an outlook on the potential of Opportunistic Memory Systems.

8.1 Overview of Contributions

A series of techniques were proposed to opportunistically *exploit* (Part 1) and *cope* (Part 2) with hardware variability in memory systems. In Part 1, ViPZonE, DPCS, and X-Mem each explored ways to improve the energy efficiency of memory systems, while in Part 2, the impact of corrected memory errors on performance were studied, while SDECC and ViFFTo were proposed to deal with faults that would otherwise cause catastrophic detected-but-uncorrectable errors.

8.1.1 ViPZonE

ViPZonE (a power variation-aware memory management scheme, described in Chapter 2) improves the energy efficiency and energy proportionality of off-chip DRAM main memory with no hardware changes. The approach relies on the pre-characterization of memory power variability that can be performed by the platform firmware or OS at boot-time; the results are then kept in storage and used during runtime. The power consumption of distinct memory modules was measured on a hardware testbed. A Linux kernel was customized with a ViPZonE page allocator and virtual memory system call interface. When full memory bandwidth is unnecessary, applications can then explicitly allocate memory in a power variation-aware manner to reduce energy with minimal impact to performance. Legacy applications that are not ViPZonE-aware, or those needing all memory bandwidth, can allocate memory as normal to avoid performance degradation. Therefore, ViPZonE proved that opportunistic memory systems can exploit hardware variability to save energy even on existing commodity systems.

8.1.2 DPCS

Dynamic Power/Capacity Scaling (DPCS) was proposed in Chapter 3 to opportunistically improve the energy proportionality of on-chip SRAM caches in presence of hardware variability. Like ViPZonE, this approach opportunistically saves energy by exploiting the inherent variations in memory, but unlike ViPZonE, as a cache micro-architectural technique it does not require any software support. To support DPCS, the minimum non-faulty voltage of each cache line is first determined using built-in-self-test circuits. The cache tag array is augmented with a few bits per line to maintain a voltage fault map. A new observation that SRAM obeys the Fault Inclusion Property allows for compact multi-level fault maps with low storage overhead. At runtime, the DPCS cache controller performs fault-tolerant voltage-scaling to save energy when the full cache capacity is not needed. At low voltage, the fault map is used to disable and power gate individual faulty blocks, reducing capacity, and at high voltage, the blocks are made available to increase performance. DPCS achieved lower power at the same effective cache capacity compared to prior works that use lower voltages because its minimalistic design incurs much lower hardware overheads to implement. DPCS is a valuable supplement to opportunistic processor dynamic voltage/frequency scaling (DVFS) to enable more energy-proportional operation in presence of hardware variability.

8.1.3 X-Mem

X-Mem was proposed in Chapter 4 (end of Part 1) as a new open-source, cross-platform, and e<u>x</u>tensible <u>mem</u>ory micro-benchmarking tool that surpasses the capabilities of all prior known tools. It was used in a series of case studies of interest to cloud providers and subscribers to better understand how the memory hierarchy details of different systems can influence the performance of cloud applications. In the final case study, X-Mem was used to study the efficacy of DRAM latency variation-aware tuning for cloud providers. Although DRAM power variations were successfully exploited with ViPZonE, it was found that exploiting the chip-to-chip differences in internal memory delay bring little benefit to a real cloud server. This is because for realistic workloads, tuning the off-chip memory latency tends to have a large impact on performance only when there is large bandwidth demand. It is precisely in this scenario where DRAM timing parameters matter the least to performance, because overall CPU-to-memory access latency is dominated by queuing delays. Moreover, a large fraction of round-trip unloaded memory latency on a modern server is due to on-chip delays, which are not subject to variation-aware tuning.

8.1.4 Performability

X-Mem found additional uses in Chapter 5 (beginning of Part 2), where it was extended to inject correctable memory errors into the main memory of a real cloud server running live applications. It was found that even though corrected memory errors do not pose a correctness/reliability problem, they can cause tremendous slowdowns in application performance, harming system availability. This was previously found to be a major issue in the field, where memory variability often results in increased susceptibility to both hard and soft faults. This is because corrected memory errors need to be reported to the system and datacenter to aid hardware serviceability, but they are implemented using a slow firmware/software interrupt-based reporting mechanism. Based on these insights, new analytical models were developed based on queuing theory to predict the impact of memory errors on theoretical batch and interactive applications. These models can be used to project the benefits of more efficient memory error-handling schemes for different classes of cloud workloads.

8.1.5 SDECC

Software-Defined ECC (SDECC, Chapter 6) proposed a new class of techniques for heuristically recovering from detected-but-uncorrectable memory errors. Unlike all known prior work, SDECC contributed new coding-theoretical results and applied them to a novel architecture that allows software to recover from rare but severe errors that would otherwise cause machine panics or time/energy-consuming checkpoint rollbacks. SDECC leveraged side information about data contents of memory to choose the best candidate codeword out of a short list of possibilities that can be quickly computed in software based on knowledge of the hardware ECC implementation. SDECC applies to a wide range of currently-used and practical ECC codes. SDECC requires essentially no hardware changes or overheads compared to existing implementations. SDECC is a powerful, general, and opportunistic methodology to recover from rare yet severe errors that exceed the correction guarantees of any ECC code by itself. It can be used to dramatically improve the reliability and availability of systems in presence of hardware variability, especially those that are approximation-tolerant.

8.1.6 ViFFTo

Finally, Virtualization-Free Fault Tolerance (ViFFTo, in Chapter 7) proposed a holistic methodology to opportunistically deal with both hard and soft faults in software-managed embedded memories, which are ubiquitous in IoT devices. ViFFTo is comprised of two steps – FaultLink and Software-Defined Error-Localizing Codes (SDELC) – that deal with hard faults at link-time and soft faults at run-time, respectively. It does this without expensive hardware overheads, which is important to the cost-sensitive IoT device market. In ViFFTo, each manufactured chip is first characterized for hard faults that arise from manufacturing defects or voltage-scaling in presence of parametric process variation, similar to the method used in DPCS. Unlike DPCS, however, the fault map is not stored on the device; rather, it is provided to the application/system developer who purchases the chip. The software application is compiled once, but linked separately and independently in a fault-aware manner for each distinct chip at software deployment time. In this way, the application is guaranteed to operate correctly in presence of known hard fault locations in the memory and can be used to safely scale the voltage below the minimum non-faulty VDD. At run-time, random single-bit soft faults are recovered heuristically using SDELC; it is much simpler and lower-overhead than a Hamming code, while more powerful than a simple single-parity detection bit. Similar to the approach taken by SDECC in Chapter 6, SDELC recovers soft faults heuristically using an embedded SDELC C library that supports application-defined rules for correct recovery. ViFFTo is a low-cost opportunistic technique that is well suited to saving energy and improving longevity of IoT devices that will inevitably experience faults as a result of hardware variability.

8.2 Directions for Future Work

There are many future possibilities for Opportunistic Memory Systems. In the short term, they include Software-Defined ECC with support for fault models; application-specific fault-tolerance for hardware accelerators; and adapting techniques in this dissertation to emerging non-volatile memory devices. In the long term, possible directions include joint abstractions for heterogeneity and variability, and a Checkerboard Architecture. These potential directions are summarized below.

8.2.1 Software-Defined ECC with Fault Models

There is much room for future work on SDECC, which has opened a new area for research that spans both coding theory and computer architecture. The recovery policies for SDECC thus far only take into account side information about data messages in memory; fault model side information is currently ignored. There are many coding-theoretic works that have developed ECCs for specific fault models, and many computer architecture studies that use fault maps or adaptive-strength conventional ECCs to exploit fault side information. However, no known prior works exploit fault side information and message side information simultaneously to recover from errors.

In a system where the primary fault model is random radiation-induced bit flips, the current policies are well suited. But suppose the primary source of faults is due to DRAM refresh failure (i.e., the refresh period is too long in presence of process and temperature variation). In this case, the faults are likely to be asymmetric and/or data-dependent [328], i.e, $1 \rightarrow 0$ could be more likely than $0 \rightarrow 1$ if a charged DRAM cell represents a logical 1. If the only permitted fault mode is $1 \rightarrow 0$, then on average, SDECC with SECDED would reduce the number of candidates by $4\times$. Therefore, SDECC could choose a better recovery candidate in light of updated posterior probabilities that account for independent message probabilities and fault probabilities.

It is also important for future work to evaluate SDECC in context of realistic fault models. Large-granularity memory faults [403] often cause an avalanche of CEs (like those observed in Chapter 5) but they could cause avalanches of DUEs as well. If preventative action (e.g., page retirement) is not taken immediately after SDECC recovery of a DUE caused by a hard fault, it is likely that future DUEs will overwhelm SDECC either by causing an accidental MCE or by dramatically slowing down the system, much like CEs do today.

8.2.2 Application-Specific Fault-Tolerance for Hardware Accelerators

SDECC and SDELC are intended to improve the reliability and availability of software applications running on hardware that occasionally has memory faults. The concept of heuristically recovering detected-but-uncorrectable ECC errors could also be extended to application-specific hardware accelerators.

Typically, accelerators are designed to speed up a core part of an application that is computation and memory intensive. They are often dataflow-oriented, so data is likely to be typed and organized in a regular manner. Relatively little is understood about the reliability of hardware accelerators and how memory errors may affect them. Given that accelerators are often approximation-tolerant and are likely to need high memory bandwidth and low access latency, having a low-overhead ECC mechanism could improve energy efficiency and performance. Detected-but-uncorrectable errors could be recovered using an application-defined heuristic in either hardware or firmware, which would allow the ECC code to be weaker (with lower overheads) than otherwise needed.

8.2.3 Adapting Techniques to Non-Volatile Memory Devices

This dissertation was primarily concerned with exploiting and coping with variability in commodity memory technologies like SRAM and DRAM. In the future, it is likely that emerging random-access non-volatile memories (NVMs) such as STT-MRAM, MeRAM, ReRAM/memristors, etc., will find mainstream use in systems. However, it is unlikely that any one NVM technology will become a universal replacement for current devices.

Although much prior work has explored new circuits, micro-architectures, hybrid architectures, and software support for NVMs, few studies have focused on opportunistic non-volatile memory systems that exploit and cope with variability. This is a ripe area for exploration because current NVMs typically suffer from susceptibility to process variations and poor scalability that currently limit their mass-market adoption. Moreover, it is a common fallacy that NVMs are immune to soft faults: while it is true that they are less susceptible to *radiation-induced* soft faults, they introduce other types. For instance, STT-MRAMs are susceptible to random self-induced bit flips due to thermal instability; reducing this fault probability requires considerably more write energy and/or latency. Many NVMs also suffer from aging and wear-out problems that volatile memories like SRAM and DRAM do not have.

For instance, DPCS could be extended to NVM-based cache architectures. Unlike DPCS for SRAM, which uses fault-tolerant voltage scaling as its low-level mechanism for power/capacity scaling, DPCS for STT-MRAM could leverage block-level variations in write latency and pulse current, as well as data retention time. In low-energy mode with reduced write pulse width, any STT-MRAM cells that fail to write correctly, consume too much write energy or fail to retain data sufficiently could be considered temporarily faulty. Blocks that contain faulty bits could be disabled at the cache level, thereby reducing usable cache capacity in exchange for lower power and faster operation. With a working DPCS mechanism for SRAM and NVM, one would have the necessary tools to build a much more energy-proportional cache architecture consisting of heterogeneous memories. The exposed power/capacity knobs to the system software would be the same regardless of the underlying memory technology, while the circuit-level reliability impact would be abstracted by the cache fault-tolerant architecture.

8.2.4 Joint Abstractions for Hardware Heterogeneity and Variability

Many techniques have been proposed to deal with hardware variability, while there exists a separate body of work on heterogeneous computer architectures. Interestingly, both variation-aware and heterogeneous architectures typically require system or application software support to exploit differences in the underlying hardware. The principal difference between them is that design-based heterogeneity is *intentional variability*, whereas the process and environmental variability considered in this dissertation is *unintentional variability*. One could design computer architectures that combine support for heterogeneity and variability under joint abstractions.

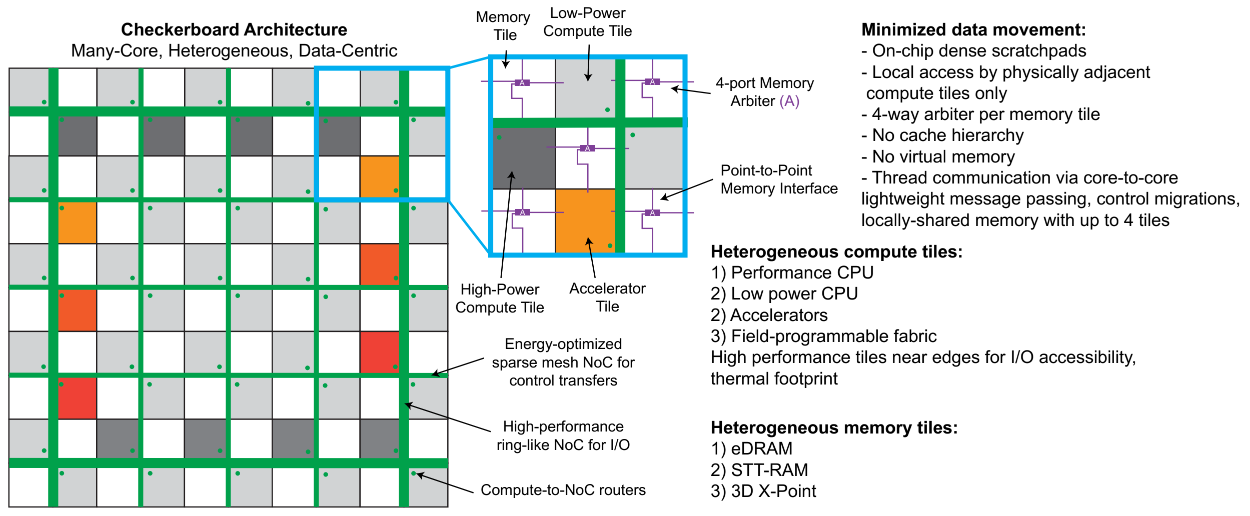

Checkerboard Architecture
Many-Core, Heterogeneous, Data-Centric
Memory Tile
Low-Power Compute Tile
4-port Memory Arbiter (A)
Point-to-Point Memory Interface
High-Power Compute Tile
Accelerator Tile
Energy-optimized sparse mesh NoC for control transfers
High-performance ring-like NoC for I/O
Compute-to-NoC routers
Minimized data movement:
- On-chip dense scratchpads
- Local access by physically adjacent compute tiles only
- 4-way arbiter per memory tile
- No cache hierarchy
- No virtual memory
- Thread communication via core-to-core lightweight message passing, control migrations, locally-shared memory with up to 4 tiles
Heterogeneous compute tiles:
1) Performance CPU
2) Low power CPU
2) Accelerators
3) Field-programmable fabric
High performance tiles near edges for I/O accessibility, thermal footprint
Heterogeneous memory tiles:
1) eDRAM
2) STT-RAM
3) 3D X-Point

Figure 8.1: Proposed heterogeneous 2D Checkerboard Architecture for future work.

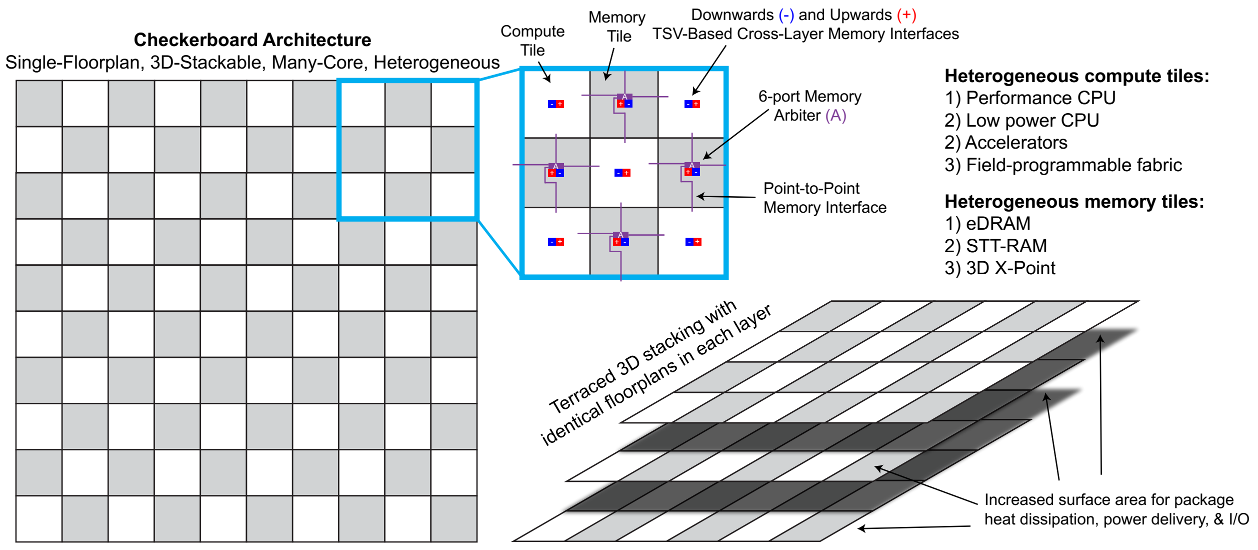

Checkerboard Architecture
Single-Floorplan, 3D-Stackable, Many-Core, Heterogeneous
Compute Tile
Memory Tile
Downwards (-) and Upwards (+) TSV-Based Cross-Layer Memory Interfaces
6-port Memory Arbiter (A)
Point-to-Point Memory Interface
Heterogeneous compute tiles:
1) Performance CPU
2) Low power CPU
2) Accelerators
3) Field-programmable fabric
Heterogeneous memory tiles:
1) eDRAM
2) STT-RAM
3) 3D X-Point
Terraced 3D stacking with identical floorplans in each layer
Increased surface area for package heat dissipation, power delivery, & I/O

Figure 8.2: Proposed heterogeneous 3D Checkerboard Architecture for future work.

8.2.5 Putting it all Together: Many-Core Checkerboard Architecture for the Cloud

Many-core processors are increasingly finding use in datacenters and supercomputers. Typically, a uniform tiled architecture with a mesh or ring interconnect is used to scale out a design. In a cloud setting, many-core chips often run a set of independent virtual machines, but current tiled architectures are designed for a flat and universally addressable memory system. This feature is unnecessary and possibly inefficient for cloud workloads.

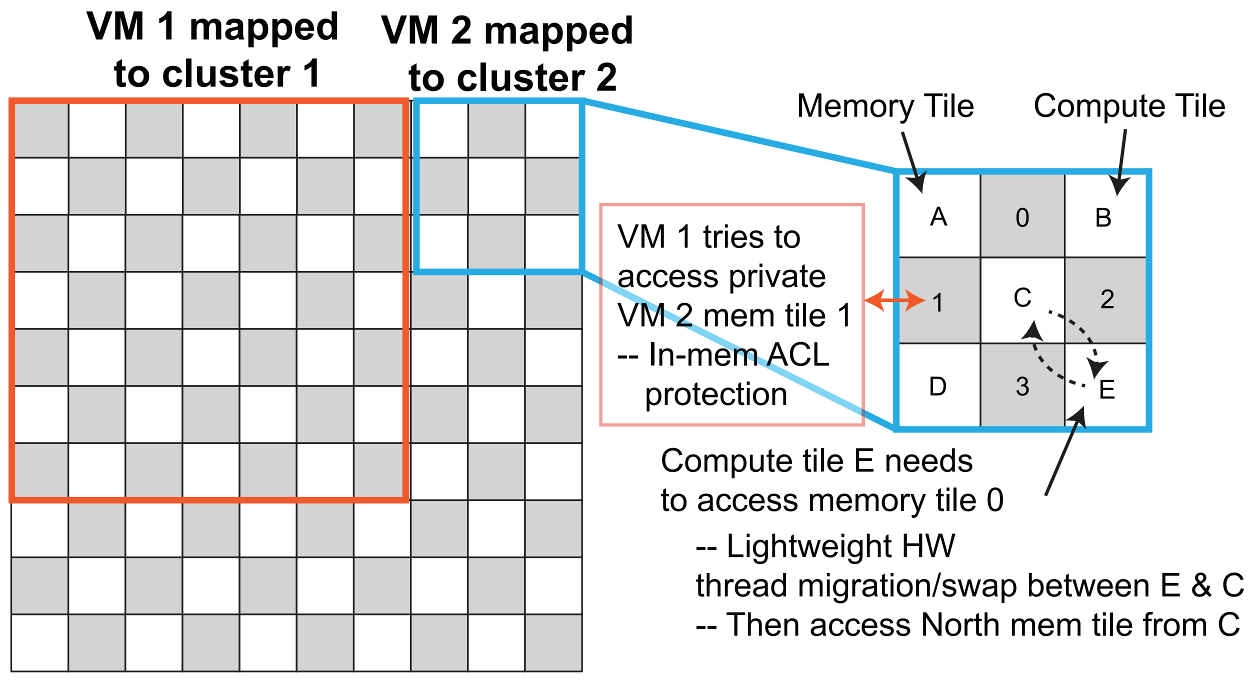

VM 1 mapped to cluster 1
VM 2 mapped to cluster 2
Memory Tile
Compute Tile
A
0
B
1
C
2
D
3
E
VM 1 tries to access private VM 2 mem tile 1 -- In-mem ACL protection
Compute tile E needs to access memory tile 0
-- Lightweight HW thread migration/swap between E & C
-- Then access North mem tile from C

Figure 8.3: Possible virtual machine mapping on a proposed 2D Checkerboard Architecture for future work.

For task and/or data-parallel cloud applications, one could design a *Checkerboard Architecture*. In such a system, a regular 2D or 3D-integrated (die-stacked) layout of alternating processor and memory tiles (a floorplan that resembles a checkerboard) would spread the computation power evenly across the chip area and ensure local memory is available to each processor tile. To improve scalability to many tiles and to achieve high energy efficiency and virtual machine isolation, each processor tile would only be able to access memory tiles that are physically adjacent; similarly, memory tiles would only be able to be accessed by its neighboring processor tiles. In a 2D Checkerboard Architecture, this would mean up to four compute tiles (which could be a heterogeneous mix of conventional cores and accelerators) can access a memory tile (which could also be a heterogeneous mix of volatile and non-volatile memories). The concept is illustrated in Fig. 8.1. In a 3D architecture, each memory/compute tile would have six neighbors (North, South, East, West, Up, and Down). This concept is illustrated in Fig. 8.2.

In either the 2D or 3D Checkerboard Architecture, the memory tiles would only be accessable through local and relative addressing, while virtual memory would deliberately be unsupported. This would enable simpler and more energy-efficient interconnects and tiles. However, it would be problematic whenever remote memory tile access is required. For example, many compute tiles might need to access a single piece of data on one tile, or a single compute tile needs to access a larger working set than is available on adjacent memory tiles. This problem could be solved as follows.

- Instead of remote memory access, hardware supports migration of lightweight thread contexts.
- Instead of virtual memory, fully relocatable programs should be built, and each memory tile receives a base address offset. In addition, each memory tile should implement basic access control (e.g., forbid accesses from the Northern adjacent tile, which belongs to a different virtual machine).
- Instead of compute threads allocating memory, virtual machines are allocated memory tiles, which automatically include their adjacent computation tiles. In other words, the computation to memory ratio is fixed by the architecture; a large working set uses a large number of cores.

A possible mapping of two virtual machines onto the Checkerboard Architecture is shown in Fig. 8.3.

The Checkerboard Architecture could allow for lightweight or no cache coherence, reduced global communication, and greater many-core scalability. To achieve such a large "datacenter-on-chip," the memory system will likely need to be heterogeneous and variation-tolerant. Opportunistic Memory Systems provides a useful suite of techniques to help address this challenge.

8.3 Outlook

Moving forward, there is significant potential for Opportunistic Memory Systems to be a valuable solution to the problem of hardware variability. Meanwhile, it is likely that Opportunistic Memory Systems will continue to become more relevant as increasing data creates more demand for high-density memory that has low random-access latency, high bandwidth, high reliability, and low cost. The trend of hardware specialization will increase the pressure on memory system design even faster than logic has done before.

As variability continues to worsen in the nanoscale process nodes, adaptive system-level techniques can be used to reduce design and runtime guardbands to save energy and improve reliability. Without such advances, the memory bottleneck will become an even greater challenge to resolve and will limit the future scalability of systems in both the embedded and warehouse-scale computing domains. Even if variability stabilizes at the end of Moore's Law, Opportunistic Memory Systems can still improve system energy efficiency, reliability, and cost, especially if low-overhead techniques are used (like those in this dissertation).

The completed projects in this dissertation do not represent the complete scope of what is possible for Opportunistic Memory Systems in Presence of Hardware Variability. Future work should continue to explore this area of research.

REFERENCES

[1] M. Gottscho, L. A. D. Bathen, N. Dutt, A. Nicolau, and P. Gupta, "ViPZonE: Hardware Power Variability-Aware Memory Management for Energy Savings," *IEEE Transactions on Computers (TC)*, vol. 64, no. 5, pp. 1483–1496, 2015.

[2] M. Gottscho, A. A. Kagalwalla, and P. Gupta, "Power Variability in Contemporary DRAMs," *IEEE Embedded Systems Letters (ESL)*, vol. 4, no. 2, pp. 37–40, 2012.

[3] M. Gottscho, P. Gupta, and A. A. Kagalwalla, "Analyzing Power Variability of DDR3 Dual Inline Memory Modules," University of California, Los Angeles, Tech. Rep., 2011.

[4] L. Bathen, M. Gottscho, N. Dutt, P. Gupta, and A. Nicolau, "ViPZonE: OS-Level Memory Variability-Driven Physical Address Zoning for Energy Savings," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2012, pp. 33–42.

[5] N. Dutt, P. Gupta, A. Nicolau, L. Bathen, and M. Gottscho, "Variability-Aware Memory Management for Nanoscale Computing," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013.

[6] M. Gottscho, "ViPZonE: Exploiting DRAM Power Variability for Energy Savings in Linux x86-64," University of California, Los Angeles, Tech. Rep., 2014.

[7] L. Wanner, L. Lai, A. Rahimi, M. Gottscho, P. Mercati, C.-H. Huang, F. Sala, Y. Agarwal, L. Dolecek, N. Dutt, P. Gupta, R. Gupta, R. Jhala, R. Kumar, S. Lerner, S. Mitra, A. Nicolau, T. Simunic Rosing, M. B. Srivastava, S. Swanson, D. Sylvester, and Y. Zhou, "NSF Expedition on Variability-Aware Software: Recent Results and Contributions," *it - Information Technology*, vol. 57, no. 3, 2015.

[8] M. Gottscho, A. BanaiyanMofrad, N. Dutt, A. Nicolau, and P. Gupta, "DPCS: Dynamic Power/Capacity Scaling for SRAM Caches in the Nanoscale Era," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, p. 26, 2015.

[9] ——, "Power / Capacity Scaling: Energy Savings With Simple Fault-Tolerant Caches," in *Proceedings of the Design Automation Conference (DAC)*, 2014.

[10] N. Dutt, P. Gupta, A. Nicolau, A. BanaiyanMofrad, M. Gottscho, and M. Shoushtari, "Multi-Layer Memory Resiliency," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2014.

[11] M. Gottscho, S. Govindan, B. Sharma, M. Shoaib, and P. Gupta, "X-Mem: A Cross-Platform and Extensible Memory Characterization Tool for the Cloud," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 263–273.

[12] M. Gottscho, M. Shoaib, S. Govindan, B. Sharma, D. Wang, and P. Gupta, "Measuring the Impact of Memory Errors on Application Performance," *IEEE Computer Architecture Letters (CAL)*, 2016.

[13] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software-Defined Error-Correcting Codes," in *The 12th Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2016.

[14] ——, "Software-Defined Error-Correcting Codes," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2016.

[15] J. von Neumann, "The First Draft Report on the EDVAC," Tech. Rep., 1945.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., 2012.

[17] A. W. Burks, H. H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Tech. Rep., 1946.

[18] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/-Software Interface*, 4th ed., 2012.

[19] S. Borkar and A. A. Chien, "The Future of Microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[20] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 9, no. 5, pp. 256–268, 1974.

[21] J. M. Rabaey, A. Chandrakasan, and N. Borivoje, "Designing Memory and Array Structures," in *Digital Integrated Circuits - A Design Perspective*, 2nd ed., 2003, pp. 623–717.

[22] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed., 2011.

[23] N. A. Sherwani, *Algorithms for VLSI Physical Design Automation*, 1995.

[24] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics Magazine*, 1965.

[25] "JEDEC DDR3 SDRAM Standard." [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd-79-3d

[26] "JEDEC DDR4 SDRAM Standard JESD79-4," Tech. Rep., 2012.

[27] C. Kim, D. Burger, and S. W. Keckler, "NUCA: A Non-Uniform Cache Access Architecture for Wire-Delay Dominated On-Chip Caches," *IEEE Micro Top Pics*, 2003.

[28] M. Gorman, *Understanding the Linux Virtual Memory Manager*, 2004.

[29] S. Thompson, M. Armstrong, C. Auth, M. Alavi, M. Buehler, R. Chau, S. Cea, T. Ghani, G. Glass, T. Hoffman, C.-H. Jan, C. Kenyon, J. Klaus, K. Kuhn, Z. Ma, B. Mcintyre, K. Mistry, A. Murthy, B. Obradovic, R. Nagisetty, P. Nguyen, S. Sivakumar, R. Shaheed, L. Shifren, B. Tufts, S. Tyagi, M. Bohr, and Y. El-Mansy, "A 90-nm Logic Technology Featuring Strained-Silicon," *IEEE Transactions on Electron Devices*, vol. 51, no. 11, pp. 1790–1797, 2004.

[30] S. Narasimha, K. Onishi, H. M. Nayfeh, A. Waite, M. Weybright, J. Johnson, C. Fonseca, D. Corliss, C. Robinson, M. Crouse, D. Yang, C.-H. Wu, A. Gabor, T. Adam, I. Ahsan, M. Belyansky, L. Black, S. Butt, J. Cheng, A. Chou, G. Costrini, C. Dimitrakopoulos, A. Domenicucci, P. Fisher, A. Frye, S. Gates, S. Greco, S. Grunow, M. Hargrove, J. Holt, S.-J. Jeng, M. Kelling, B. Kim, W. Landers, G. Larosa, D. Lea, M. Lee, X. Liu, N. Lustig, A. McKnight, L. Nicholson, D. Nielsen, K. Nummy, V. Ontalus, C. Ouyang, X. Ouyang, C. Prindle, R. Pal, W. Rausch, D. Restaino, C. Sheraw, J. Sim, A. Simon, T. Standaert, C. Y. Sung, K. Tabakman, C. Tian, R. Van Den Nieuwenhuizen, H. Van Meer, A. Vayshenker, D. Wehella-Gamage, J. Werking, R. C. Wong, S. Wu, J. Yu, R. Augur, D. Brown, X. Chen, D. Edelstein, A. Grill, M. Khare, Y. Li, S. Luning, J. Norum, S. Sankaran, D. Schepis, R. Wachnik, R. Wise, C. Wann, T. Ivers, and P. Agnello, "High Performance 45-nm SOI Technology with Enhanced Strain, Porous Low-k BEOL, and Immersion Lithography," in *Proceedings of the IEEE International Electron Devices Meeting (IEDM)*, 2006.

[31] K. Mistry, C. Allen, C. Auth, B. Beattie, D. Bergstrom, M. Bost, M. Brazier, M. Buehler, A. Cappellani, R. Chau, C.-H. Choi, G. Ding, K. Fischer, T. Ghani, R. Grover, W. Han, D. Hanken, M. Hattendorf, J. He, J. Hicks, R. Huessner, D. Ingerly, P. Jain, R. James, L. Jong, S. Joshi, C. Kenyon, K. Kuhn, K. Lee, H. Liu, J. Maiz, B. McIntyre, P. Moon, J. Neirynck, S. Pae, C. Parker, D. Parsons, C. Prasad, L. Pipes, M. Prince, P. Ranade, T. Reynolds, J. Sandford, L. Shifren, J. Sebastian, J. Seiple, D. Simon, S. Sivakumar, P. Smith, C. Thomas, T. Troeger, P. Vandervoorn, S. Williams, and K. Zawadzki, "A 45nm Logic Technology with High-k+Metal Gate Transistors, Strained Silicon, 9 Cu Interconnect Layers, 193nm Dry Patterning, and 100% Pb-free Packaging," in *2007 IEEE International Electron Devices Meeting*, 2007, pp. 247–250.

[32] C.-H. Jan, U. Bhattacharya, R. Brain, S.-J. Choi, G. Curello, G. Gupta, W. Hafez, M. Jang, M. Kang, K. Komeyli, T. Leo, N. Nidhi, L. Pan, J. Park, K. Phoa, A. Rahman, C. Staus, H. Tashiro, C. Tsai, P. Vandervoorn, L. Yang, J.-Y. Yeh, and P. Bai, "A 22nm SoC platform technology featuring 3-D tri-gate and high-k/metal gate, optimized for ultra low power, high performance and high density SoC applications," in *2012 International Electron Devices Meeting*, 2012, pp. 3.1.1–3.1.4.

[33] O. Mutlu, J. Meza, and L. Subramanian, "The Main Memory System: Challenges and Opportunities," *Communications of the Korean Institute of Information Scientists and Engineers*, 2015.

[34] "Report to the National Science Foundation on the Workshop for Energy Efficient Computing," NSF and SRC, Tech. Rep., 2015.

[35] S. Herbert and D. Marculescu, "Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors," in *Proceedings of the IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2007, pp. 38–43.

[36] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing Power in High-Performance Microprocessors," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 1998, pp. 732–737.

[37] Q. Wu, M. Pedram, and X. Wu, “Clock-Gating and Its Application to Low Power Design of Sequential Circuits,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications (TCAS-I)*, vol. 47, no. 3, pp. 415–420, 2000.

[38] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, “Microarchitectural Techniques for Power Gating of Execution Units,” in *Proceedings of the IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2004, pp. 32–37.

[39] L. A. Barroso and U. Hölzle, “The Case for Energy-Proportional Computing,” *IEEE Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[40] ——, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2009, vol. 4, no. 1.

[41] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. Keller, “Energy Management for Commercial Servers,” *IEEE Computer*, vol. 36, no. 12, pp. 39–48, 2003.

[42] M. Pedram, D. Brooks, and T. Pinkston, “Report for the NSF Workshop on Cross-layer Power Optimization and Management,” NSF, Tech. Rep., 2012.

[43] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger, “Dark Silicon and the End of Multicore Scaling,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[44] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, “Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance,” in *Proceedings of the 31st annual international symposium on Computer architecture*, 2004.

[45] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, “Architecture Support for Accelerator-Rich CMPs,” in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2012, pp. 843–849.

[46] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural Acceleration for General-Purpose Approximate Programs,” *Communications of the ACM*, vol. 58, no. 1, pp. 105–115, 2014.

[47] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, “Profiling a Warehouse-Scale Computer,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2015, pp. 158–169.

[48] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014, pp. 13–24.

[49] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable Computing Architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[50] J. Oh, G. Kim, J. Park, I. Hong, S. Lee, and H.-J. Yoo, "A 320mW 342GOPS Real-Time Moving Object Recognition Processor for HD 720p Video Streams," in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2012, pp. 220–222.

[51] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng, "FPGA Implementation of GZIP Compression and Decompression for IDC Services," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, 2010, pp. 265–268.

[52] S. K. Mathew, F. Sheikh, M. Kounavis, S. Gueron, A. Agarwal, S. K. Hsu, H. Kaul, M. A. Anders, and R. K. Krishnamurthy, "53 Gbps Native GF(2^4)^2 Composite-Field AES-Encrypt/Decrypt Accelerator for Content-Protection in 45 nm High-Performance Microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 4, pp. 767–776, 2011.

[53] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–269–269–284–284–284, 2014.

[54] B. Grigorian, N. Farahpour, and G. Reinman, "BRAINIAC: Bringing Reliable Accuracy into Neurally-Implemented Approximate Computing," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 615–626.

[55] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2015.

[56] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A Cloud-Scale Acceleration Architecture," in *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2016.

[57] A. Putnam, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. M. Caulfield, A. Smith, J. Thong, P. Y. Xiao, D. Burger, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, and G. P. Gopal, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *Communications of the ACM*, vol. 59, no. 11, pp. 114–122, 2016.

[58] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-L. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. Mackean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan,

R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit TM," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[59] "Amazon EC2 F1 Instances: Run Custom FPGAs in the AWS Cloud." [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[60] M. Wutting, "Phase-change materials: Towards a universal memory?" *Nature Materials*, vol. 4, no. 4, pp. 265–266, 2005.

[61] S. S. P. Parkin, M. Hayashi, and L. Thomas, "Magnetic Domain-Wall Racetrack Memory," *Science*, vol. 320, no. 5873, pp. 190–194, 2008.

[62] S. Sugahara and N. Junsaku, "Spin-Transistor Electronics: An Overview and Outlook," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2124–2154, 2010.

[63] S. A. Wolf, J. Lu, M. R. Stan, E. Chen, and D. M. Treger, "The Promise of Nanomagnetics and Spintronics for Future Logic and Universal Memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2155–2168, 2010.

[64] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li, "Emerging Non-Volatile Memories," in *Proceedings of the ACM/IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011, pp. 325–334.

[65] K. L. Wang, J. G. Alzate, and P. Khalili Amiri, "Low-Power Non-Volatile Spintronic Memory: STT-RAM and Beyond," *Journal of Physics D: Applied Physics*, vol. 46, no. 7, 2013.

[66] C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari, "Bridging the Processor-Memory Performance Gap with 3D IC Technology," *IEEE Design Test of Computers*, vol. 22, no. 6, pp. 556–564, 2005.

[67] S. Borkar, "3D Integration for Energy Efficient System Design," in *48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 214–219.

[68] G. Van der Plas, P. Limaye, I. Loi, A. Mercha, H. Oprins, C. Torregiani, S. Thijs, D. Linten, M. Stucchi, G. Katti, D. Velenis, V. Cherman, B. Vandevelde, V. Simons, I. De Wolf, R. Labie, D. Perry, S. Bronckers, N. Minas, M. Cupac, W. Ruythooren, J. Van Olmen, A. Phommahaxay, M. de Potter de ten Broeck, A. Opdebeeck, M. Rakowski, B. De Wachter, M. Dehan, M. Nelis, R. Agarwal, A. Pullini, F. Angiolini, L. Benini, W. Dehaene, Y. Travaly, E. Beyne, and P. Marchal, "Design Issues and Considerations for Low-Cost 3-D TSV IC Technology," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 293–307, 2011.

[69] B. K. Kaushik, M. K. Majumder, and V. R. Kumar, "Carbon Nanotube Based 3-D Interconnects - A Reality or a Distant Dream," *IEEE Circuits and Systems Magazine*, vol. 14, no. 4, pp. 16–35, 2014.

[70] M.-C. F. Chang, J. Cong, A. Kaplan, C. Liu, M. Naik, J. Premkumar, G. Reinman, E. Socher, and S.-W. Tam, "Power Reduction of CMP Communication Networks via RF-Interconnects," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2008, pp. 376–387.

[71] C. Xiao, M.-C. Frank Chang, J. Cong, M. Gill, Z. Huang, C. Liu, G. Reinman, and H. Wu, "Stream Arbitration: Towards Efficient Bandwidth Utilization for Emerging On-Chip Interconnects," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 1–27, 2013.

[72] "The International Technology Roadmap for Semiconductors." [Online]. Available: http://www.itrs.net

[73] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter Variations and Impact on Circuits and Microarchitecture," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2003, pp. 338–342.

[74] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[75] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. K. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. S. Rosing, M. B. Srivastava, S. Swanson, and D. Sylvester, "Underdesigned and Opportunistic Computing in Presence of Hardware Variability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 32, no. 1, pp. 8–23, 2013.

[76] I. Koren and Z. Koren, "Defect Tolerance in VLSI Circuits: Techniques and Yield Analysis," *Proceedings of the IEEE*, vol. 86, no. 9, pp. 1819–1838, 1998.

[77] K. A. Bowman, S. G. Duvall, and J. D. Meindl, "Impact of Die-to-Die and Within-Die Parameter Fluctuations on the Maximum Clock Frequency Distribution for Gigascale Integration," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 37, no. 2, pp. 183–190, 2002.

[78] A. A. Kagalwalla, "Computational Methods for Design-Assisted Mask Flows," Ph.D. Dissertation, University of California, Los Angeles, 2014.

[79] R. S. Ghaida, "Design Enablement and Design-Centric Assessment of Future Semiconductor Technologies," Ph.D. Dissertation, University of California, Los Angeles, 2012.

[80] X. Tang, V. K. De, and J. D. Meindl, "Intrinsic MOSFET Parameter Fluctuations due to Random Dopant Placement," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 5, no. 4, pp. 369–376, 1997.

[81] A. Asenov, "Random Dopant Induced Threshold Voltage Lowering and Fluctuations in Sub-0.1 μm MOSFET's: A 3-D "Atomistic" Simulation Study," *IEEE Transactions on Electron Devices (TED)*, vol. 45, no. 12, pp. 2505–2513, 1998.

[82] P. Oldiges, Q. Lin, K. Petrillo, M. Sanchez, M. Ieong, and M. Hargrove, "Modeling Line Edge Roughness Effects in sub 100 Nanometer Gate Length Devices," in *Proceedings of the IEEE International Conference on Simulation Semiconductor Processes and Devices (SISPAD)*, 2000, pp. 131–134.

[83] C. H. Diaz, H.-J. Tao, Y.-C. Ku, A. Yen, and K. Young, “An Experimentally Validated Analytical Model for Gate Line-Edge Roughness (LER) Effects on Technology Scaling,” *IEEE Electron Device Letters (EDL)*, vol. 22, no. 6, pp. 287–289, 2001.

[84] A. Asenov, S. Kaya, and A. R. Brown, “Intrinsic Parameter Fluctuations in Decananometer MOSFETs Introduced by Gate Line Edge Roughness,” *IEEE Transactions on Electron Devices (TED)*, vol. 50, no. 5, pp. 1254–1260, 2003.

[85] L. Lai, “Cross-Layer Approaches for Monitoring, Margining and Mitigation of Circuit Variability,” Ph.D. Dissertation, University of California, Los Angeles, 2015.

[86] M. S. Gupta, J. L. Oatley, R. Joseph, G.-Y. Wei, and D. M. Brooks, “Understanding Voltage Variations in Chip Multiprocessors using a Distributed Power-Delivery Network,” in *2007 Design, Automation & Test in Europe Conference & Exhibition*, 2007, pp. 1–6.

[87] M. Sadri, A. Bartolini, and L. Benini, “Temperature Variation Aware Multi-Scale Delay, Power and Thermal Analysis at RT and Gate Level,” *Elsevier Integration, the VLSI Journal*, vol. 49, pp. 35–48, 2014.

[88] L. Lai, C.-H. Chang, and P. Gupta, “Exploring Total Power Saving from High Temperature of Server Operations,” University of California, Los Angeles, Tech. Rep., 2014.

[89] V. Chandra and R. Aitken, “Impact of Voltage Scaling on Nanoscale SRAM Reliability,” in *Design, Automation, and Test in Europe (DATE)*, 2009, pp. 387–392.

[90] V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. D. Smith, G.-Y. Wei, and D. Brooks, “Voltage Smoothing: Characterizing and Mitigating Voltage Noise in Production Processors via Software-Guided Thread Scheduling,” in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 77–88.

[91] Q. Xie, M. J. Dousti, and M. Pedram, “Therminator: A Thermal Simulator for Smartphones Producing Accurate Chip and Skin Temperature Maps,” in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2014.

[92] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer, “High-Performance CMOS Variability in the 65-nm Regime and Beyond,” *IBM Journal of Research and Development*, vol. 50, no. 4.5, pp. 433–449, 2006.

[93] M. Namaki-Shoushtari, A. Rahimi, N. Dutt, P. Gupta, and R. K. Gupta, “ARGO: Aging-Aware GPGPU Register File Allocation,” in *Proceedings of the ACM/IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.

[94] B. Ebrahimi, R. Asadpour, A. Afzali-Kusha, and M. Pedram, “A FinFET SRAM Cell Design with BTI Robustness at High Supply Voltages and High Yield at Low Supply Voltages,” *International Journal of Circuit Theory and Applications*, 2015.

[95] C.-H. Ho, S. Y. Kim, and K. Roy, “Ultra-Thin Dielectric Breakdown in Devices and Circuits: A Brief Review,” *Elsevier Microelectronics Reliability*, 2014.

[96] X. Wang, J. Keane, T. T.-H. Kim, P. Jain, Q. Tang, and C. H. Kim, “Silicon Odometers: Compact In Situ Aging Sensors for Robust System Design,” *IEEE Micro*, vol. 34, no. 6, pp. 74–85, 2014.

[97] J. Lienig, “Electromigration and Its Impact on Physical Design in Future Technologies,” in *Proceedings of the ACM International Symposium on Physical Design (ISPD)*, 2013, pp. 33–40.

[98] “Intel’s e-DRAM Shows Up in the Wild,” 2014. [Online]. Available: http://semimd.com/chipworks/2014/02/07/intels-e-dram-shows-up-in-the-wild/

[99] J. Cong, H. Duwe, R. Kumar, and S. Li, “Better-Than-Worst-Case Design: Progress and Opportunities,” *Journal of Computer Science and Technology*, vol. 29, no. 4, pp. 656–663, 2014.

[100] F. Wang, C. Nicopoulos, X. Wu, Y. Xie, and N. Vijaykrishnan, “Variation-Aware Task Allocation and Scheduling for MPSoC,” in *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, 2007, pp. 598–603.

[101] J. Sartori, A. Pant, R. Kumar, and P. Gupta, “Variation-Aware Speed Binning of Multi-core Processors,” in *Proceedings of the IEEE International Symposium on Quality Electronic Design (ISQED)*, 2010, pp. 307–314.

[102] H. Hanson, K. Rajamani, J. Rubio, S. Ghiasi, and F. Rawson, “Benchmarking for Power and Performance,” in *SPEC Benchmark Workshop*, 2007.

[103] L. Wanner, C. Apte, R. Balani, P. Gupta, and M. Srivastava, “A Case for Opportunistic Embedded Sensing in Presence of Hardware Power Variability,” *Workshop on Power Aware Computers and Systems (HotPower)*, 2010.

[104] A. Pant, P. Gupta, and M. van der Schaar, “Software Adaptation in Quality Sensitive Applications to Deal with Hardware Variability,” in *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI*, 2010, pp. 85–90.

[105] L. Wanner, R. Balani, S. Zahedi, C. Apte, P. Gupta, and M. Srivastava, “Variability-Aware Duty Cycle Scheduling in Long Running Embedded Sensing Systems,” in *Design, Automation, and Test in Europe (DATE)*, 2011, pp. 1–6.

[106] K. Meng and R. Joseph, “Process Variation Aware Cache Leakage Management,” in *Proceedings of the IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2006, pp. 262–267.

[107] X. Liang, R. Canal, G.-Y. Wei, and D. Brooks, “Process Variation Tolerant 3T1D-Based Cache Architectures,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 15–26.

[108] M. Mutyam, F. Wang, R. Krishnan, V. Narayanan, M. Kandemir, Y. Xie, and M. J. Irwin, “Process-Variation-Aware Adaptive Cache Architecture and Management,” *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 865–877, 2009.

[109] A. Sasan, H. Homayoun, A. Eltawil, and F. Kurdahi, "A Fault Tolerant Cache Architecture for Sub 500mV Operation: Resizable Data Composer Cache (RDC-Cache)," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2009, pp. 251–260.

[110] L. A. D. Bathen and N. D. Dutt, "E-RoC: Embedded RAIDs-on-Chip for Low Power Distributed Dynamically Managed Reliable Memories," in *Design, Automation, and Test in Europe (DATE)*, 2011.

[111] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power Aware Page Allocation," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 35, no. 11, 2000, pp. 105–116.

[112] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Scheduler-Based DRAM Energy Management," in *Proceedings of the Design Automation Conference (DAC)*, 2002, pp. 697–702.

[113] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic Tracking of Page Miss Ratio Curve for Memory Management," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004, pp. 177–188.

[114] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu, "Mini-Rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency," in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, 2008, pp. 210–221.

[115] J. H. Ahn, J. Leverich, R. Schreiber, and N. Jouppi, "Multicore DIMM: an Energy Efficient Memory Module with Independently Controlled DRAMs," *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 5–8, 2008.

[116] L. A. D. Bathen, N. D. Dutt, A. Nicolau, and P. Gupta, "VaMV: Variability-Aware Memory Virtualization," in *Design, Automation, and Test in Europe (DATE)*, 2012, pp. 284–287.

[117] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the ACM/IEEE/IFIP International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 72–81.

[118] H. S. Stone, *High-Performance Computer Architecture*, 3rd ed., 1993.

[119] G. J. Burnett and E. G. Coffman Jr, "A Study of Interleaved Memory Systems," in *Proceedings of the Spring Joint Computer Conference*, 1970, pp. 467–474.

[120] B. R. Rau, "Interleaved Memory Bandwidth in a Model of a Multiprocessor Computer System," *IEEE Transactions on Computers (TC)*, vol. 100, no. 9, pp. 678–681, 1979.

[121] P. Budnik and D. J. Kuck, "The Organization and Use of Parallel Memories," *IEEE Transactions on Computers*, vol. C-20, no. 12, pp. 1566–1569, 1971.

[122] D. J. Kuck and R. A. Stokes, "The Burroughs Scientific Processor (BSP)," *IEEE Transactions on Computers*, vol. C-31, no. 5, pp. 363–376, 1982.

[123] G. S. Sohi, "High-Bandwidth Interleaved Memories for Vector Processors – A Simulation Study," *IEEE Transactions on Computers (TC)*, vol. 42, no. 1, pp. 34–44, 1993.

[124] M. B. Junior and J. P. Shen, "Organization of Array Data for Concurrent Memory Access," in *Proceedings of the Workshop on Microprogramming and Microarchitecture*, 1988, pp. 97–99.

[125] R. Love, *Linux Kernel Development*, 3rd ed., 2010.

[126] D. Bovet, M. Cesati, and A. Oram, *Understanding the Linux Kernel*, 2002. [Online]. Available: files/1870/citation.html

[127] M. Gorman, *Understanding the Linux Virtual Memory Manager*, 2007.

[128] "The Linux Kernel 3.2," 2012. [Online]. Available: www.kernel.org

[129] "GLIBC, The GNU C Library," 2012. [Online]. Available: www.gnu.org/s/libc

[130] L. Cheng, P. Gupta, C. J. Spanos, K. Qian, and L. He, "Physically Justifiable Die-Level Modeling of Spatial Variation in View of Systematic Across Wafer Variability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 3, pp. 388–401, 2011.

[131] L.-T. Pang, K. Qian, C. J. Spanos, and B. Nikolic, "Measurement and Analysis of Variability in 45 nm Strained-Si CMOS Technology," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 8, pp. 2233–2243, 2009.

[132] S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tschanz, V. Erraguntla, and N. Borkar, "Within-Die Variation-Aware Dynamic-Voltage-Frequency Scaling Core Mapping and Thread Hopping for an 80-Core Processor," in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2010, pp. 174–175.

[133] K. Rajamani, C. Lefurgy, S. Ghiasi, J. C. Rubio, H. Hanson, and T. Keller, "Power Management for Computer Systems and Datacenters," 2008.

[134] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller, "Improving Energy Efficiency by Making DRAM Less Randomly Accessed," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2005, pp. 393–398.

[135] "Memtest86+." [Online]. Available: http://www.memtest.org/

[136] "34410A/11A 6-1/2 Digit Multimeter User's Guide," 2012.

[137] K. Itoh, *VLSI Memory Chip Design*, 1st ed., 2001.

[138] "TN-41-01: Calculating Memory System Power for DDR3," 2007.

[139] J. Hezavei, N. Vijaykrishnan, and M. J. Irwin, "A Comparative Study of Power Efficient SRAM Designs," in *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2000, pp. 117–122.

[140] H.-W. Lee, K.-H. Kim, Y.-K. Choi, J.-H. Sohn, N.-K. Park, K.-W. Kim, C. Kim, Y.-J. Choi, and B.-T. Chung, "A 1.6 V 1.4 Gbp/s/pin Consumer DRAM With Self-Dynamic Voltage Scaling Technique in 44 nm CMOS Technology," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, pp. 131–140, 2012.

[141] M. Gottscho, "ViPZonE Source Code," 2013. [Online]. Available: http://github.com/nanocad-lab/

[142] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2009, pp. 664–669.

[143] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," *Communications of the ACM*, vol. 54, no. 2, pp. 100–107, 2011.

[144] V. Sridharan and D. Liberty, "A Field Study of DRAM Errors," AMD, Tech. Rep., 2012.

[145] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy Caches: Simple Techniques for Reducing Leakage Power," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002, pp. 148–157.

[146] A. Kumar, J. Rabaey, and K. Ramchandran, "SRAM Supply Voltage Scaling: A Reliability Perspective," in *Proceedings of the International Symposium on Quality Electronic Design (ISQED)*, 2009, pp. 782–787.

[147] J. Wang and B. H. Calhoun, "Minimum Supply Voltage and Yield Estimation for Large SRAMs Under Parametric Variations," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 19, no. 11, pp. 2120–2125, 2011.

[148] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the Spring Joint Computer Conference*, 1967, p. 483.

[149] A. BanaiyanMofrad, H. Homayoun, and N. Dutt, "FFT-Cache: A Flexible Fault-Tolerant Cache Architecture for Ultra Low Voltage Operation," in *Proceedings of the ACM/IEEE International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011, pp. 95–104.

[150] S. Mittal, "A Survey of Architectural Techniques for Improving Cache Power Efficiency," *Sustainable Computing: Informatics and Systems*, vol. 4, no. 1, pp. 33–43, 2014.

[151] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories," in *Proceedings of the IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2000, pp. 90–95.

[152] H.-Y. Cheng, M. Poremba, N. Shahidi, I. Stalev, M. J. Irwin, M. Kandemir, J. Sampson, and Y. Xie, "EECache: Exploiting Design Choices in Energy-Efficient Last-Level Caches for Chip Multiprocessors," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2014, pp. 303–306.

[153] A. Bardine, M. Comparetti, P. Foglia, and C. A. Prete, "Evaluation of Leakage Reduction Alternatives for Deep Submicron Dynamic Nonuniform Cache Architecture Caches," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 22, no. 1, pp. 185–190, 2014.

[154] P. P. Shirvani and E. J. McCluskey, "PADded Cache: A New Fault-Tolerance Technique for Cache Memories," in *Proceedings of the VLSI Test Symposium*, 1999, pp. 440–445.

[155] A. Agarwal, B. C. Paul, H. Mahmoodi, A. Datta, and K. Roy, "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 1, pp. 27–38, 2005.

[156] S. Ozdemir, D. Sinha, G. Memik, J. Adams, and H. Zhou, "Yield-Aware Cache Architectures," in *In Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2006, pp. 15–25.

[157] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe, "Multi-Bit Error Tolerant Caches Using Two-Dimensional Error Coding," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2007, pp. 197–209.

[158] C.-K. Koh, W.-F. Wong, Y. Chen, and H. Li, "The Salvage Cache: A Fault-Tolerant Cache Architecture for Next-Generation Memory Technologies," in *Proceedings of the International Conference on Computer Design (ICCD)*, 2009, pp. 268–274.

[159] A. Ansari, S. Gupta, S. Feng, and S. Mahlke, "ZerehCache: Armoring Cache Architectures in High Defect Density Technologies," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009, pp. 100–110.

[160] D. Rossi, N. Timoncini, M. Spica, and C. Metra, "Error Correcting Code Analysis for Cache Memory High Reliability and Performance," in *Design, Automation, and Test in Europe (DATE)*, 2011, pp. 1–6.

[161] A. R. Alameldeen, Z. Chishti, C. Wilkerson, W. Wu, and S.-L. Lu, "Adaptive Cache Design to Enable Reliable Low-Voltage Operation," *IEEE Transactions on Computers (TC)*, vol. 60, no. 1, pp. 50–63, 2011.

[162] A. R. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu, "Energy-Efficient Cache Design Using Variable-Strength Error-Correcting Codes," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011, pp. 461–471.

[163] M. K. Qureshi and Z. Chishti, "Operating SECDED-Based Caches at Ultra-Low Voltage with FLAIR," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[164] L. Chang, D. M. Fried, J. Hergenrother, J. W. Sleight, R. H. Dennard, R. K. Montoye, L. Sekaric, S. J. McNab, A. W. Topol, C. D. Adams, K. W. Guarini, and W. Haensch, "Stable SRAM Cell Design for the 32 nm Node and Beyond," in *Symposium on VLSI Technology Digest of Technical Papers*, 2005, pp. 128–129.

[165] B. H. Calhoun and A. Chandrakasan, “A 256kb Sub-threshold SRAM in 65nm CMOS,” in *IEEE International Solid State Circuits Conference (ISSCC) Digest of Technical Papers*, 2006, pp. 2592–2601.

[166] M. A. Hussain and M. Mutyam, “Block Remap with Turnoff: A Variation-Tolerant Cache Design Technique,” in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2008, pp. 783–788.

[167] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, “Trading off Cache Capacity for Reliability to Enable Low Voltage Operation,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2008, pp. 203–214.

[168] J. Abella, J. Carretero, P. Chaparro, X. Vera, and A. González, “Low Vccmin Fault-Tolerant Cache with Highly Predictable Performance,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009, pp. 111–121.

[169] A. Ansari, S. Feng, S. Gupta, and S. Mahlke, “Archipelago: A Polymorphic Cache Design for Enabling Robust Near-Threshold Operation,” in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 539–550.

[170] A. Sasan, H. Homayoun, K. Amiri, A. Eltawil, and F. Kudahi, “History & Variation Trained Cache (HVT-Cache): A Process Variation Aware and Fine Grain Voltage Scalable Cache with Active Access History Monitoring,” in *Proceedings of the International Symposium on Quality Electronic Design (ISQED)*, 2012, pp. 498–505.

[171] M. Zhang, V. M. Stojanovic, and P. Ampadu, “Reliable Ultra-Low-Voltage Cache Design for Many-Core Systems,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 59, no. 12, pp. 858–862, 2012.

[172] Y. Han, Y. Wang, H. Li, and X. Li, “Enabling Near-Threshold Voltage (NTV) Operation in Multi-VDD Cache for Power Reduction,” in *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, 2013, pp. 337–340.

[173] A. Chakraborty, H. Homayoun, A. Khajeh, N. Dutt, A. Eltawil, and F. Kurdahi, “Multicopy Cache: A Highly Energy-Efficient Cache Architecture,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 5s, 2014.

[174] T. Mahmood, S. Hong, and S. Kim, “Ensuring Cache Reliability and Energy Scaling at Near-Threshold Voltage with Macho,” *IEEE Transactions on Computers (TC)*, vol. 64, no. 6, pp. 1694–1706, 2014.

[175] S. Kim and M. R. Guthaus, “SEU-Aware Low-Power Memories Using a Multiple Supply Voltage Array Architecture,” in *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, vol. 418, 2013, pp. 181–195.

[176] H. R. Ghasemi, S. C. Draper, and N. S. Kim, “Low-voltage on-chip cache architecture using heterogeneous cell sizes for high-performance processors,” in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 38–49.

[177] P.-H. Wang, W.-C. Cheng, Y.-H. Yu, T.-C. Kao, C.-L. Tsai, P.-Y. Chang, T.-J. Lin, J.-S. Wang, and T.-F. Chen, "Variation-Aware and Adaptive-Latency Accesses for Reliable Low Voltage Caches," in *Proceedings of the International Conference on Very Large Scale Integration (VLSI-SoC)*, 2013, pp. 358–363.

[178] F. Hijaz and O. Khan, "NUCA-L1: A Non-Uniform Access Latency Level-1 Cache Architecture for Multicores Operating at Near-Threshold Voltages," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, pp. 1–28, 2014.

[179] B. S. Mohammad, H. Saleh, and M. Ismail, "Design Methodologies for Yield Enhancement and Power Efficiency in SRAM-Based SoCs," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 23, no. 10, pp. 2054–2064, 2014.

[180] A. Ferreron, D. Suarez-Gracia, J. Alastruey, T. Monreal, and V. Vinals, "Block Disabling Characterization and Improvements in CMPs Operating at Ultra-low Voltages," in *Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2014.

[181] X. Fan, C. S. Ellis, and A. R. Lebeck, "The Synergy Between Power-Aware Memory Systems and Processor Voltage Scaling," *Lecture Notes in Computer Science*, vol. 3164, pp. 164–179, 2005.

[182] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, "MemScale: Active Low-Power Modes for Main Memory," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 46, no. 3, 2011, pp. 225–238.

[183] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory Power Management via Dynamic Voltage/Frequency Scaling," in *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2011, pp. 31–40.

[184] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "CoScale: Coordinating CPU and Memory System DVFS in Server Systems," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012, pp. 143–154.

[185] L. Lai and P. Gupta, "Accurate and Inexpensive Performance Monitoring for Variability-Aware Systems," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 467–473.

[186] Y. Agarwal, A. Bishop, T.-B. Chan, M. Fotjik, P. Gupta, A. B. Kahng, L. Lai, P. Martin, M. Srivastava, D. Sylvester, L. Wanner, and B. Zhang, "RedCooper: Hardware Sensor Enabled Variability Software Testbed for Lifetime Energy Constrained Application," University of California, Los Angeles, Tech. Rep., 2014. [Online]. Available: http://www.escholarship.org/uc/item/1c21g217

[187] S. Hamdioui, A. J. van de Goor, and M. Rodgers, "March SS: A Test for All Static Simple RAM Faults," in *International Workshop on Memory Technology, Design, and Testing (MTDT)*, 2002, pp. 95–100.

[188] B. Bowhill, B. Stackhouse, N. Nassif, Z. Yang, A. Raghavan, C. Morganti, C. Houghton, D. Krueger, O. Franza, J. Desai, J. Crop, D. Bradley, C. Bostak, S. Bhimji, and M. Becker, "The Xeon Processor E5-2600 v3: A 22nm 18-Core Product Family," in *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, 2015.

[189] J. Pyo, Y. Shin, H.-J. Lee, S.-i. Bae, M.-s. Kim, K. Kim, K. Shin, Y. Kwon, H. Oh, J. Lim, D.-w. Lee, J. Lee, I. Hong, K. Chae, H.-H. Lee, S.-W. Lee, S. Song, C.-H. Kim, J.-S. Park, H. Kim, S. Yun, U.-R. Cho, J. C. Son, and S. Park, "20nm High-k Metal-Gate Heterogeneous 64b Quad-Core CPUs and Hexa-Core GPU for High-Performance and Energy-Efficient Mobile Application Processor," in *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, 2015.

[190] M. E. Sinangil, H. Mair, and A. P. Chandrakasan, "A 28nm High-Density 6T SRAM with Optimized Peripheral-Assist Circuits for Operation Down to 0.6V," in *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, 2011, pp. 260–262.

[191] N. Verma and A. P. Chandrakasan, "A 256 kb 65 nm 8T Subthreshold SRAM Employing Sense-Amplifier Redundancy," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 43, no. 1, pp. 141–149, 2008.

[192] J. Singh, D. K. Pradhan, S. Hollis, and S. P. Mohanty, "A Single Ended 6T SRAM Cell Design for Ultra-Low-Voltage Applications," *IEICE Electronics Express*, vol. 5, no. 18, pp. 750–755, 2008.

[193] M.-F. Chang, C.-F. Chen, T.-H. Chang, C.-C. Shuai, Y.-Y. Wang, and H. Yamauchi, "A 28nm 256kb 6T-SRAM with 280mV Improvement in Vmin Using a Dual-Split-Control Assist Scheme," in *International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, 2015.

[194] Y. Li, S. Makar, and S. Mitra, "CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns," in *Design, Automation, and Test in Europe (DATE)*, 2008, pp. 885–890.

[195] N. Balasubramonian, R. Muralimanohar, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep., 2009.

[196] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[197] D. Sánchez, Y. Sazeides, J. M. Cebrián, J. M. García, and J. L. Aragón, "Modeling the Impact of Permanent Faults in Caches," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, 2013.

[198] A. S. Dhodapkar and J. E. Smith, "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002, pp. 233–244.

[199] ——, "Comparing Program Phase Detection Techniques," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2003, p. 217.

[200] A. M. Dani, B. Amrutur, and Y. N. Srikant, "Toward a Scalable Working Set Size Estimation Method and Its Application for Chip Multiprocessors," *IEEE Transactions on Computers*, vol. 63, no. 6, pp. 1567–1579, 2014.

[201] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, and A. Rabkin, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[202] L. Columbus, "Roundup of Cloud Computing Forecasts and Market Estimates," 2015. [Online]. Available: http://www.forbes.com/sites/louiscolumbus/2015/01/24/roundup-of-cloud-computing-forecasts-and-market-estimates-2015

[203] J. D. Gelas, "Server Buying Decisions: Memory." [Online]. Available: http://www.anandtech.com/print/7479/server-buying-decisions-memory

[204] W. L. Bircher and L. K. John, "Complete System Power Estimation Using Processor Performance Events," *IEEE Transactions on Computers*, vol. 61, no. 4, pp. 563–577, 2012.

[205] K. Sudan, S. Srinivasan, R. Balasubramonian, and R. Iyer, "Optimizing Datacenter Power with Memory System Levers for Guaranteed Quality-of-Service," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 117–126.

[206] "Microsoft Azure." [Online]. Available: http://azure.microsoft.com

[207] "Amazon EC2." [Online]. Available: http://aws.amazon.com/ec2/

[208] "Scaleway." [Online]. Available: http://www.scaleway.com

[209] "TryStack." [Online]. Available: http://www.trystack.org

[210] X. Fan, W.-D. Weber, and L. A. Barroso, "Power Provisioning for a Warehouse-Sized Computer," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 13–23, 2007.

[211] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating System Implications of Fast, Cheap, Non-Volatile Memory," in *USENIX Hot Topics on Operating Systems (HotOS)*, 2011.

[212] J. Zhao, C. Xu, P. Chi, and Y. Xie, "Memory and Storage System Design with Nonvolatile Memory Technologies," *IPSJ Transactions on System LSI Design Methodology*, vol. 8, pp. 2–11, 2015.

[213] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common Case," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[214] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud," in *In Proceedings of the International Conference on Cloud Computing Technology and Science*, 2010, pp. 159–168.

[215] S. K. Barker and P. Shenoy, "Empirical Evaluation of Latency-Sensitive Application Performance in the Cloud," in *Proceedings of the Conference on Multimedia Systems (MMSys)*, 2010, pp. 35–46.

[216] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, "More for Your Money: Exploiting Performance Heterogeneity in Public Clouds," in *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2012.

[217] M. Ferdman, B. Falsafi, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, and A. Ailamaki, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 47, no. 4, 2012, pp. 37–47.

[218] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server Engineering Insights for Large-Scale Online Services," *IEEE Micro*, vol. 30, no. 4, pp. 2–13, 2010.

[219] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing Public Cloud Providers," in *Proceedings of the Annual Conference on Internet Measurement (IMC)*, 2010.

[220] E. Blem, J. Menon, and K. Sankaralingam, "A Detailed Analysis of Contemporary ARM and x86 Architectures," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[221] R. Kotla, A. Devgan, S. Ghiasi, T. Keller, and F. Rawson, "Characterizing the Impact of Different Memory-Intensity Levels," in *In Proceedings of the International Workshop on Workload Characterization (WWC)*, 2004, pp. 3–10.

[222] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt, "Predicting Performance Impact of DVFS for Realistic Memory Systems," in *In Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012, pp. 155–165.

[223] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, vol. 28, no. 2, 2000, pp. 128–138.

[224] V. Cuppu and B. Jacob, "Concurrency, Latency, or System Overhead: Which Has the Largest Impact on Uniprocessor DRAM-System Performance?" in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2001, pp. 62–71.

[225] C. Natarajan, B. Christenson, and F. Briggs, "A Study of Performance Impact of Memory Controller Features in Multi-Processor Server Environment," in *Workshop on Memory Performance Issues*, 2004, pp. 80–87.

[226] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "MultiScale: Memory System DVFS With Multiple Memory Controllers," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2012, pp. 297–302.

[227] F. Ercan, N. A. Gazala, and H. David, "An Integrated Approach to System-Level CPU and Memory Energy Efficiency on Computing Systems," in *In Proceedings of the International Conference on Energy Aware Computing*, 2012.

[228] K. Chandrasekar, S. Goossens, C. Weis, M. Koedam, B. Akesson, N. Wehn, and K. Goossens, "Exploiting Expendable Process-Margins in DRAMs for Run-Time Performance Optimization," in *Design, Automation, and Test in Europe (DATE)*, 2014.

[229] J. D. McCalpin, "The STREAM Benchmark." [Online]. Available: http://www.cs.virginia.edu/stream/

[230] ——, "The STREAM2 Benchmark." [Online]. Available: http://www.cs.virginia.edu/stream/stream2

[231] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in *Proceedings of the USENIX Annual Technical Conference*, 1996.

[232] S. Siamashka, "TinyMemBench." [Online]. Available: https://github.com/ssvb/tinymembench

[233] V. Viswanathan, K. Kumar, and T. Willhalm, "Intel Memory Latency Checker v2." [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[234] "X-Mem: The eXtensible MEMory Characterization Tool." [Online]. Available: https://nanocad-lab.github.io/X-Mem/

[235] M. Gottscho, "X-Mem Source Code," 2015. [Online]. Available: https://github.com/Microsoft/X-Mem

[236] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," pp. 19–25, 1995.

[237] ——, "A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers," 1995. [Online]. Available: http://www.cs.virginia.edu/{~}mccalpin/papers/balance/index.html

[238] L. Peng, J.-K. Peir, T. K. Prakash, C. Staelin, Y.-K. Chen, and D. Koppelman, "Memory Hierarchy Performance Measurement of Commercial Dual-Core Desktop Processors," *Journal of Systems Architecture*, vol. 54, no. 8, pp. 816–828, 2008.

[239] V. Babka and P. Tma, "Investigating Cache Parameters of x86 Family Processors," in *Computer Performance Evaluation and Benchmarking*, ser. Lecture Notes in Computer Science, 2009, vol. 5419, pp. 77–96.

[240] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009, pp. 261–270.

[241] L. Bergstrom, "Measuring NUMA Effects With the STREAM Benchmark," University of Chicago, Tech. Rep., 2010.

[242] R. Murphy, "On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance," in *In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2007, pp. 35–43.

[243] "AVX-512 Instructions." [Online]. Available: https://software.intel.com/en-us/blogs/2013/avx-512-instructions

[244] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014.

[245] "Project Zero: Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges." [Online]. Available: http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

[246] "Intel 64 and IA-32 Architectures Software Developer Manuals." [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[247] "SCons." [Online]. Available: http://www.scons.org

[248] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2009, pp. 14–23.

[249] M. S. Benkmann, "The Lean Mean C++ Option Parser." [Online]. Available: optionparser.sourceforge.net

[250] "PandaBoard." [Online]. Available: http://www.pandaboard.org

[251] "Intel Performance Counter Monitor." [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[252] A. L. Shimpi, "Intel's Sandy Bridge Architecture Exposed," 2010. [Online]. Available: http://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed

[253] "Open Compute Project." [Online]. Available: http://www.opencompute.org

[254] K. V. Vishwanath and N. Nagappan, “Characterizing Cloud Computing Hardware Reliability,” in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010, pp. 193–203.

[255] B. Maurer, “Fail at Scale,” *Communications of the ACM*, vol. 58, no. 11, pp. 44–49, 2015.

[256] X. Li, M. C. Huang, K. Shen, and L. Chu, “A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility,” 2010.

[257] B. Schroeder, E. Pinheiro, and W.-D. Weber, “DRAM Errors in the Wild: A Large-Scale Field Study,” in *Proceedings of the International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009, pp. 193–204.

[258] V. Sridharan and D. Liberty, “A Study of DRAM Failures in the Field,” in *Proceedings of the IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[259] A. A. Hwang, I. A. 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 the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 40, no. 1, 2012, p. 111.

[260] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, “Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015.

[261] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, “Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[262] P. Nikolaou, Y. Sazeides, L. Ndreu, and M. Kleanthous, “Modeling the Implications of DRAM Failures and Protection Techniques on Datacenter TCO,” in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2015.

[263] S. Mittal, “A Survey of Architectural Techniques for Managing Process Variation,” *ACM Computing Surveys*, vol. 48, no. 4, 2016.

[264] A. Rahimi, L. Benini, and R. K. Gupta, “Variability Mitigation in Nanometer CMOS Integrated Systems: A Survey of Techniques From Circuits to Software,” *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1410–1448, 2016.

[265] S. Mittal and J. Vetter, “A Survey of Techniques for Modeling and Improving Reliability of Computing Systems,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 4, pp. 1226–1238, 2016.

[266] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-Power Through Critical Data Partitioning," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 213–224.

[267] S. Li, K. Chen, M.-Y. Hsieh, N. Muralimanohar, C. D. Kersey, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi, "System Implications of Memory Reliability in Exascale Computing," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[268] L. Chen, Y. Cao, and Z. Zhang, "E3CC: A Memory Error Protection Scheme with Novel Address Mapping for Subranked and Low-Power memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, pp. 1–22, 2013.

[269] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults," in *Proceedings of the IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[270] N. DeBardeleben, S. Blanchard, V. Sridharan, S. Gurumurthi, J. Stearley, K. B. Ferreira, and J. Shalf, "Extra Bits on SRAM and DRAM Errors - More Data from the Field," in *Workshop on Silicon Errors in Logic – System Effects (SELSE)*, 2014.

[271] E. Baseman, N. DeBardeleben, K. Ferreira, S. Levy, S. Raasch, V. Sridharan, T. Siddiqua, and Q. Guan, "Improving DRAM Fault Characterization Through Machine Learning," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2016.

[272] B. Delgado and K. L. Karavanic, "Performance Implications of System Management Mode," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2013, pp. 163–173.

[273] "MCA Enhancements in Future Intel Xeon Processors," 2013.

[274] "Advanced Configuration and Power Interface (ACPI) Specification." [Online]. Available: www.uefi.org

[275] "Microsoft WinDbg." [Online]. Available: https://msdn.microsoft.com/en-us/windows/hardware/hh852365.aspx

[276] A. Leon-Garcia, *Probability, Statistics, and Random Processes for Electrical Engineering*, 3rd ed., 2008.

[277] A. C. Cameron and P. K. Trivedi, *Regression Analysis of Count Data*, 1998.

[278] G. Rasch, "The Poisson Process as a Model for a Diversity of Behavioral Phenomena," in *The International Congress of Psychology*, 1963.

[279] J. Daigle, "The Basic M/G/1 Queueing System," in *Queueing Theory with Applications to Packet Telecommunication*, 2005, pp. 159–223.

[280] J. Haigh, *Probability Models*, 2nd ed., 2002.

[281] J. Tanner, “A Derivation of the Borel Distribution,” *Biometrika*, vol. 48, no. 1-2, pp. 222–224, 1961.

[282] J. F. C. Kingman, “The first Erlang centuryand the next,” *Queueing Systems*, vol. 63, no. 1-4, pp. 3–12, 2009.

[283] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, “Memory Errors in Modern Systems: The Good, The Bad, and The Ugly,” in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[284] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, “Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014, pp. 610–621.

[285] D. Tiwari, S. Gupta, and S. S. Vazhkudai, “Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014, pp. 25–36.

[286] R. W. Hamming, “Error Detecting and Error Correcting Codes,” *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[287] M. Y. 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.

[288] S. Kaneda and E. Fujiwara, “Single Byte Error Correcting – Double Byte Error Detecting Codes for Memory Systems,” *IEEE Transactions on Computers (TC)*, vol. C-31, no. 7, pp. 596–602, 1982.

[289] T. J. Dell, “A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory,” IBM Microelectronics Division, Tech. Rep., 1997.

[290] S. Lin and D. J. Costello, *Error Control Coding*, 2004.

[291] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, “LOT-ECC: LOcalized and Tiered Reliability Mechanisms for Commodity Memory Systems,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2012.

[292] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar, “Low-Power, Low-Storage-Overhead Chipkill Correct via Multi-line Error Correction,” in *Proceedings of the IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013, pp. 1–12.

[293] X. Jian, J. Sartori, H. Duwe, and R. Kumar, "High Performance, Energy Efficient Chipkill Correct Memory with Multidimensional Parity," *IEEE Computer Architecture Letters (CAL)*, vol. 12, no. 2, pp. 39–42, 2013.

[294] X. Jian, V. Sridharan, and R. Kumar, "Parity Helix: Efficient Protection for Single-Dimensional Faults in Multi-Dimensional Memory Systems," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 555–567.

[295] A. Davydov and L. Tombak, "An Alternative to the Hamming Code in the Class of SEC-DED Codes in Semiconductor Memory," *IEEE Transactions on Information Theory*, vol. 37, no. 3, pp. 897–902, 1991.

[296] R. Bose and D. Ray-Chaudhuri, "On a Class of Error Correcting Binary Group Codes," *Information and Control*, vol. 3, no. 1, pp. 68–79, 1960.

[297] J. D. Gelas, "The Intel Xeon E5 v4 Review: Testing Broadwell-EP With Demanding Server Workloads," 2016. [Online]. Available: http://www.anandtech.com/show/10158/the-intel-xeon-e5-v4-review

[298] S. Mitra and K. Kim, "X-Compact: An Efficient Response Compaction Technique," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 23, no. 3, pp. 421–432, 2004.

[299] J. C. Smolens, "Fingerprinting: Hash-Based Error Detection in Microprocessors," Ph.D. Dissertation, Carnegie Mellon University, 2008.

[300] J. Yang, Y. Zhang, and R. Gupta, "Frequent Value Compression in Data Caches," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2000, pp. 258–265.

[301] A. Alameldeen and D. Wood, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches," University of Wisconsin, Madison, Tech. Rep., 2004.

[302] S. Mittal and J. Vetter, "A Survey of Architectural Approaches for Data Compression in Cache and Main Memory Systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 5, pp. 1524–1536, 2015.

[303] G. Pekhimenko, V. Seshadri, O. Mutlu, 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 ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[304] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2013, pp. 172–184.

[305] J. Kim, M. Sullivan, E. Choukse, and M. Erez, “Bit-Plane Compression: Transforming Data for Better Compression in Many-core Architectures,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[306] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value Locality and Load Value Prediction,” in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[307] J. S. Miguel, M. Badr, and N. E. Jerger, “Load Value Approximation,” in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2014, pp. 127–139.

[308] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, “Mitigating the Memory Bottleneck with Approximate Load Value Prediction,” *IEEE Design & Test*, vol. 33, no. 1, pp. 32–42, 2016.

[309] J. S. Miguel, J. Albericio, N. E. Jerger, and A. Jaleel, “The Bunker Cache for Spatio-Value Approximation,” in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2016.

[310] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, “The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.0,” 2014. [Online]. Available: https://riscv.org

[311] Q. Nguyen, “RISC-V Tools (GNU Toolchain, ISA Simulator, Tests) – git commit 816a252.” [Online]. Available: https://github.com/riscv/riscv-tools

[312] A. Waterman, “RISC-V Proxy Kernel – git commit 85ae17a.” [Online]. Available: https://github.com/riscv/riscv-pk/commit/85ae17a

[313] A. Waterman and Y. Lee, “Spike, a RISC-V ISA Simulator – git commit 3bfc00e.” [Online]. Available: https://github.com/riscv/riscv-isa-sim

[314] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, “AxBench: A Multiplatform Benchmark Suite for Approximate Computing,” *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, 2017.

[315] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, “Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design,” in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[316] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, “Characterizing the Impact of Intermittent Hardware Faults on Programs,” *IEEE Transactions on Reliability (TR)*, vol. 64, no. 1, pp. 297–310, 2015.

[317] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, “ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-layer Resilience Analysis,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[318] R. Naseer and J. Draper, “Parallel Double Error Correcting Code Design to Mitigate Multi-Bit Upsets in SRAMs,” *Proceedings of the IEEE European Solid-State Circuits Conference (ESSCIRC)*, pp. 222–225, 2008.

[319] D. H. Yoon and M. Erez, “Virtualized and Flexible ECC for Main Memory,” in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, no. 3, 2010.

[320] S. Li, D. H. Yoon, K. Chen, and J. Zhao, “MAGE : Adaptive Granularity and ECC for Resilient and Power Efficient Memory Systems,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[321] N. Oh, P. Shirvani, and E. McCluskey, “Error Detection by Duplicated Instructions in Super-Scalar Processors,” *IEEE Transactions on Reliability (TR)*, vol. 51, no. 1, pp. 63–75, 2002.

[322] ——, “Control-Flow Checking by Software Signatures,” *IEEE Transactions on Reliability (TR)*, vol. 51, no. 1, pp. 111–122, 2002.

[323] Z. Ming, X. L. Yi, and L. H. Wei, “New SEC-DED-DAEC codes for multiple bit upsets mitigation in memory,” in *Proceedings of the IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC)*, 2011, pp. 254–259.

[324] S. Cha and H. Yoon, “Single-Error-Correction and Double-Adjacent-Error-Correction Code for Simultaneous Testing of Data Bit and Check Bit Arrays in Memories,” *IEEE Transactions on Device and Materials Reliability (TDMR)*, vol. 14, no. 1, pp. 529–535, 2014.

[325] P. Reviriego, J. Martinez, S. Pontarelli, and J. A. Maestro, “A Method to Design SEC-DED-DAEC Codes With Optimized Decoding,” *IEEE Transactions on Device and Materials Reliability (TDMR)*, vol. 14, no. 3, pp. 884–889, 2014.

[326] K. Namba, S. Pontarelli, M. Ottavi, and F. Lombardi, “A Single-Bit and Double-Adjacent Error Correcting Parallel Decoder for Multiple-Bit Error Correcting BCH Codes,” *IEEE Transactions on Device and Materials Reliability (TDMR)*, vol. 14, no. 2, pp. 664–671, 2014.

[327] M. Jung, É. Zulian, D. M. Mathew, M. Herrmann, C. Brugger, C. Weis, and N. Wehn, “Omitting Refresh: A Case Study for Commodity and Wide I/O DRAMs,” in *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, 2015, pp. 85–91.

[328] C. Weis, M. Jung, P. Ehses, C. Santos, P. Vivet, S. Goossens, M. Koedam, and N. Wehn, “Retention Time Measurements and Modelling of Bit Error Rates of WIDE I/O DRAM in MPSoCs,” in *Design, Automation, and Test in Europe (DATE)*, 2015.

[329] M. Jung, C. C. Rheinländer, C. Weis, and N. Wehn, “Reverse Engineering of DRAMs: Row Hammer with Crosshair,” in *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, 2016, pp. 471–476.

[330] S. Khan, C. Wilkerson, D. Lee, A. R. Alameldeen, and O. Mutlu, “A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM,” *IEEE Computer Architecture Letters (CAL)*, 2016.

[331] X. Li, M. C. Huang, K. Shen, and L. Chu, “A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility,” in *USENIX Annual Technical Conference (ATC)*, 2012.

[332] S. Wang, H. C. Hu, H. Zheng, and P. Gupta, “MEMRES: A Fast Memory System Reliability Simulator,” *IEEE Transactions on Reliability (TR)*, vol. 65, no. 4, pp. 1783–1797, 2016.

[333] F. Sala, H. Duwe, L. Dolecek, and R. Kumar, “A Unified Framework for Error Correction in On-chip Memories,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2016, pp. 268–274.

[334] D. H. Yoon and M. Erez, “Memory Mapped ECC: Low-Cost Error Protection for Last Level Caches,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2009.

[335] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, “Reducing Cache Power with Low-Cost, Multi-Bit Error-Correcting Codes,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2010.

[336] W. Wen, M. Mao, X. Zhu, S. H. Kang, D. Wang, and Y. Chen, “CD-ECC: Content-Dependent Error Correction Codes for Combating Asymmetric Nonvolatile Memory Operation Errors,” in *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, 2013.

[337] X. Jian and R. Kumar, “ECC Parity: A Technique for Efficient Memory Error Resilience for Multi-Channel Memory Systems,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 1035–1046.

[338] H. Duwe, X. Jian, and R. Kumar, “Correction Prediction: Reducing Error Correction Latency for On-Chip Memories,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 463–475.

[339] J. Kim, M. Sullivan, and M. Erez, “Bamboo ECC: Strong, Safe, and Flexible Codes for Reliable Computer Memory,” in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[340] P. J. Nair, V. Sridharan, and M. K. Qureshi, “XED: Exposing On-Die Error Detection Information for Strong Memory Reliability,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[341] D. W. Kim and M. Erez, “RelaxFault Memory Repair,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[342] J. Kim, M. Sullivan, S. Lym, and M. Erez, “All-Inclusive ECC: Thorough End-to-End Protection for Reliable Computer Memory,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[343] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, “Use ECP, not ECC, for Hard Failures in Resistive Memories,” *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, vol. 38, no. 3, p. 141, 2010.

[344] 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 *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 466–477.

[345] M. K. Qureshi, "Pay-As-You-Go: Low-Overhead Hard-Error Correction for Phase Change Memories," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2011.

[346] H. Duwe, X. Jian, D. Petrisko, and R. Kumar, "Rescuing Uncorrectable Fault Patterns in On-Chip Memories through Error Pattern Transformation," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016, pp. 634–644.

[347] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "Citadel: Efficiently Protecting Stacked Memory from TSV and Large Granularity Failures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, 2016.

[348] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *Proceedings of the ACM SIGMETRICS Conference*, 2016, pp. 323–336.

[349] K. Chandrasekar, C. Weis, B. Akesson, N. Wehn, and K. Goossens, "Towards Variation-Aware System-Level Power Estimation of DRAMs," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2013.

[350] M. Shoushtari, A. BanaiyanMofrad, and N. Dutt, "Exploiting Partially-Forgetful Memories for Approximate Computing," *IEEE Embedded Systems Letters (ESL)*, vol. 7, no. 1, pp. 19–22, 2015.

[351] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan, "Approximate Storage for Energy Efficient Spintronic Memories," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2015.

[352] D. Jeong, Y. H. Oh, J. W. Lee, and Y. Park, "An eDRAM-Based Approximate Register File for GPUs," *IEEE Design & Test*, vol. 33, no. 1, pp. 23–31, 2016.

[353] M. Jung, D. M. Mathew, C. Weis, and N. Wehn, "Efficient Reliability Management in SoCs - An Approximate DRAM Perspective," in *Proceedings of the ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016, pp. 390–394.

[354] A. Rahmati, M. Hicks, D. E. Holcomb, and K. Fu, "Probable Cause: The Deanonymizing Effects of Approximate DRAM," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2015, pp. 604–615.

[355] P. Elias, "List Decoding for Noisy Channels," Massachusetts Institute of Technology (MIT), Tech. Rep., 1957.

[356] J. Wozencraft, "List Decoding," *Quarterly Progress Report*, 1958.

[357] M. Sudan, “List Decoding: Algorithms and Applications,” in *Springer Lecture Notes in Computer Science*, 2001, pp. 25–41.

[358] V. Guruswami, “List Decoding of Error-Correcting Codes,” Ph.D. Dissertation, Massachusetts Institute of Technology (MIT), 2001.

[359] M. Sudan, “Decoding of Reed Solomon Codes beyond the Error-Correction Bound,” pp. 180–193, 1997.

[360] F. Parvaresh and A. Vardy, “Correcting Errors Beyond the Guruswami-Sudan Radius in Polynomial Time,” in *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS’05)*, 2005, pp. 285–294.

[361] R. Gallager, “Low-density parity-check codes,” *IEEE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[362] W. Kang, W. Zhao, L. Yang, J.-O. Klein, Y. Zhang, and D. Ravclosona, “One-step majority-logic-decodable codes enable STT-MRAM for high speed working memories,” in *2014 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2014, pp. 1–6.

[363] S. Hamdioui, G. Gaydadjiev, and A. J. van de Goor, “The State-of-art and Future Trends in Testing Embedded Memories,” in *International Workshop on Memory Technology, Design and Testing (MTDT)*, 2004, pp. 54–59.

[364] S.-L. Lu, Q. Cai, and P. Stolt, “Memory Resiliency,” *Intel Technology Journal*, vol. 17, no. 1, 2013.

[365] P. R. Panda, N. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*, 1999.

[366] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, “Scratchpad Memory: A Design Alternative for Cache On-Chip Memory in Embedded Systems,” in *Proceedings of the ACM/IEEE International Symposium on Hardware/Software Codesign (CODES)*, 2002.

[367] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge, “Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 2, pp. 167–184, 2004.

[368] S. E. Schuster, “Multiple Word/Bit Line Redundancy for Semiconductor Memories,” *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 13, no. 5, pp. 698–703, 1978.

[369] L. Wanner, C. Apte, R. Balani, P. Gupta, and M. Srivastava, “Hardware Variability-Aware Duty Cycling for Embedded Sensors,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 6, pp. 1000–1012, 2013.

[370] “Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (Version 1.2),” 1995.

[371] R. C. Baumann, “Radiation-Induced Soft Errors in Advanced Semiconductor Technologies,” *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[372] M. Mutyam and V. Narayanan, “Working with Process Variation Aware Caches,” in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE ’07*, 2007, pp. 1–6.

[373] M. Mavropoulos, G. Keramidas, and D. Nikolos, “A Defect-Aware Reconfigurable Cache Architecture for Low-Vccmin DVFS-Enabled Systems,” in *Design, Automation, and Test in Europe (DATE)*, 2015, pp. 417–422.

[374] M. Manoochehri, M. Annavaram, and M. Dubois, “CPPC: Correctable Parity Protected Cache,” in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011.

[375] R. van Rein, “BadRAM: Linux Kernel Support for Broken RAM Modules.” [Online]. Available: http://rick.vanrein.org/linux/badram/

[376] M. M. Sabry, D. Atienza, and F. Catthoor, “OCEAN: An Optimized HW/SW Reliability Mitigation Approach for Scratchpad Memories in Real-Time SoCs,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, 2014.

[377] H. Sayadi, H. Farbeh, A. M. H. Monazzah, and S. G. Miremadi, “A Data Recomputation Approach for Reliability Improvement of Scratchpad Memory in Embedded Systems,” in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2014, pp. 228–233.

[378] A. M. H. Monazzah, H. Farbeh, S. G. Miremadi, M. Fazeli, and H. Asadi, “FTSPM: A Fault-Tolerant ScratchPad Memory,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[379] H. Farbeh, M. Fazeli, F. Khosravi, and S. G. Miremadi, “Memory Mapped SPM: Protecting Instruction Scratchpad Memory in Embedded Systems against Soft Errors,” in *Proceedings of the European Dependable Computing Conference (EDCC)*, 2012, pp. 218–226.

[380] F. Li, G. Chen, M. Kandemir, and I. Kolcu, “Improving Scratch-Pad Memory Reliability Through Compiler-Guided Data Block Duplication,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2005, pp. 1002–1005.

[381] D. P. Volpato, A. K. Mendonca, L. C. dos Santos, and J. L. Güntzel, “A Post-Compiling Approach that Exploits Code Granularity in Scratchpads to Improve Energy Efficiency,” in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2010, pp. 127–132.

[382] F. J. Aichelmann, “Fault-Tolerant Design Techniques for Semiconductor Memory Applications,” *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 177–183, 1984.

[383] J. K. Wolf and B. Elspas, “Error-Locating Codes – A New Concept in Error Control,” *IEEE Transactions on Information Theory*, vol. 9, no. 2, pp. 113–117, 1963.

[384] J. Wolf, "On Codes Derivable from the Tensor Product of Check Matrices," *IEEE Transactions on Information Theory*, vol. 11, no. 2, pp. 281–284, 1965.

[385] J. K. Wolf, "On an Extended Class of Error-Locating Codes," *Information and Control*, vol. 8, no. 2, pp. 163–169, 1965.

[386] E. Fujiwara and M. Kitakami, "A Class of Error Locating Codes for Byte-Organized Memory Systems," in *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, 1993, pp. 110–119.

[387] M. Kitakami and E. Fujiwara, "A Class of Error Locating Codes: SEC S e/b EL Codes," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 78, no. 9, pp. 1086–1091, 1995.

[388] S. Borade, B. Nakiboglu, and Lizhong Zheng, "Some Fundamental Limits of Unequal Error Protection," in *IEEE International Symposium on Information Theory (ISIT)*, 2008, pp. 2222–2226.

[389] N. Rahnavard, B. N. Vellambi, and F. Fekri, "Rateless Codes With Unequal Error Protection Property," *IEEE Transactions on Information Theory*, vol. 53, no. 4, pp. 1521–1532, 2007.

[390] N. Thomos, N. V. Boulgouris, and M. G. Strintzis, "Wireless Image Transmission Using Turbo Codes and Optimal Unequal Error Protection," *IEEE Transactions on Image Processing*, vol. 14, no. 11, pp. 1890–1901, 2005.

[391] U. Horn, K. Stuhlmüller, M. Link, and B. Girod, "Robust Internet Video Transmission Based on Scalable Coding and Unequal Error Protection," *Elsevier Signal Processing: Image Communication*, vol. 15, no. 1, pp. 77–94, 1999.

[392] B. Masnick and J. Wolf, "On Linear Unequal Error Protection Codes," *IEEE Transactions on Information Theory*, vol. 13, no. 4, pp. 600–607, 1967.

[393] L. A. Dunning and W. Robbins, "Optimal Encodings of Linear Block Codes for Unequal Error Protection," *Information and Control*, vol. 37, no. 2, pp. 150–177, 1978. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0019995878904928

[394] S. Lamikhov-Center, "ELFIO: C++ Library for Reading and Generating ELF Files." [Online]. Available: http://elfio.sourceforge.net/

[395] M. F. Schilling, "The Surprising Predictability of Long Runs," *Mathematics Magazine*, vol. 85, no. 2, pp. 141–149, 2012.

[396] P. Deheuvels, "Upper Bounds for k-th Maximal Spacings," *Zeitschrift fuer Wahrscheinlichkeitstheorie und verwandte Gebiete*, vol. 62, no. 4, pp. 465–474, 1983.

[397] J. Song, G. Bloom, and G. Palmer, "SuperGlue: IDL-Based, System-Level Fault Tolerance for Embedded Systems," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[398] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, “EnerJ: Approximate Data Types for Safe and General Low-Power Computation,” in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, vol. 46, no. 6, 2011.

[399] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “MiBench: A Free, Commercially Representative Embedded Benchmark Suite,” in *Proceedings of the IEEE International Workshop on Workload Characterization (IWWC)*, 2001, pp. 3–14.

[400] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, “Approximate Storage in Solid-State Memories,” in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 25–36.

[401] C. Yan and R. Joseph, “Enabling Deep Voltage Scaling in Delay Sensitive L1 Caches,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[402] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, “Architecture Support for Defending Against Buffer Overflow Attacks,” in *Workshop on Evaluating and Architecting Systems for Dependability*, 2002.

[403] P. J. Nair, “Architectural Techniques to Enable Reliable and Scalable Memory Systems,” Ph.D. Dissertation, Georgia Institute of Technology, 2017. [Online]. Available: http://arxiv.org/abs/1704.03991