Title
Lightweight Fault Tolerance in SRAM Based On-Chip Memories

Author
Alam, Irina

Publication Date
2018

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Lightweight Fault Tolerance in SRAM Based On-Chip Memories

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Electrical and Computer Engineering

by

Irina Alam

2018

ABSTRACT OF THE THESIS

Lightweight Fault Tolerance
in SRAM Based On-Chip Memories

by

Irina Alam
Master of Science in Electrical and Computer Engineering
University of California, Los Angeles, 2018
Professor Puneet Gupta, Chair

The reliability of memory subsystem is fast becoming a concern in computer architecture and system design. From on-chip embedded memories in Internet-of-Things (IoT) devices and on-chip caches to off-chip main memories, they have become the limiting factor in reliability of computing systems. This is because they are primarily designed to maximize bit storage density; this makes memories particularly sensitive to manufacturing process variation, environmental operating conditions, and aging-induced wearout. Addressing these concerns is particularly challenging in on-chip caches or embedded memories like scratchpads in IoT devices as additional area, power and latency overheads of reliability techniques in these memories need to be minimized as much as possible. Hence, this dissertation proposes Lightweight Fault Tolerance in SRAM based scratchpad memories and last level caches.

In the first part of the dissertation we propose *FaultLink*: an approach to deal with known hard faults in software managed scratchpad memories. FaultLink avoids hard faults found during testing by generating a custom-tailored application binary image for each individual chip. During software deployment-time, FaultLink optimally packs small sections of program code and data into fault-free segments of the memory address space and generates a custom linker script for a lazy-linking procedure. The second part proposes two software defined lightweight error detection and correction techniques: Software Defined Error Localization Code (SDELC) and Parity++ to recover from soft errors during run time. SDELC is mostly for embedded memories and uses

novel and inexpensive *Ultra-Lightweight Error-Localizing Codes* (UL-ELCs). These require fewer parity bits than single-error-correcting Hamming codes. Yet our UL-ELCs are more powerful than basic single-error-detecting parity: they localize single-bit errors to a specific chunk of a codeword. SDELC then heuristically recovers from these localized errors using a small embedded C library that exploits observable *side information* (SI) about the application's memory contents. Parity++ is a novel unequal message protection scheme that preferentially provides stronger error protection to certain "special messages". This protection scheme provides Single Error Detection (SED) for all messages and Single Error Correction (SEC) for a subset of special messages. Parity++ can be used in both last level caches and lightweight embedded memories.

The thesis of Irina Alam is approved.

Mani B. Srivastava

Lara Dolecek

Puneet Gupta, Committee Chair

University of California, Los Angeles

2018

To my parents who inspire me,

and without whom none of this would have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Puneet Gupta for the constant support, patience, motivation and guidance. This thesis would not have been possible without his excellent help and foresight. I would also like to thank Prof. Lara Dolecek at UCLA for being an instrumental collaborator who introduced me to the fundamentals of coding theory. I would like to thank Dr. Mark Gottscho (now at Google) for being an excellent mentor and helping me successfully complete my very first project in UCLA. Thanks also to Dr. Greg Wright from Qualcomm Research for inviting me to present at Qualcomm Research (San Diego) in 2017. His feedback and comments on my work have added valuable perspective.

I would like to thank my collaborator, Clayton Schoeny from UCLA for all the discussions and technical help. I would also like to acknowledge the support of my labmates: Dr. Mark Gottscho, Dr. Yasmine Badr, Dr. Shaodi Wang, Wei-Che Wang, Saptadeep Pal, Yoojin Chae, Tianmu Li and Wojciech Romaszkan. As I continue my journey towards PhD., I look forward to spending a few more wonderful years with most of them. Special thanks to all my friends for their support. Thank you Yasmine Badr for helping me settle down and get past my first year at UCLA and Saptadeep Pal for inspiring, motivating and supporting me, for critiquing my work and for proof-reading all my papers.

Lastly, I would like to thank my parents for their unconditional love, support and encouragement. None of my accomplishments would have been possible without them.

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 contains material that was published in [1]. Fundamental concept in Chapter 3 appeared in [2] and the rest is being prepared for publication.

CHAPTER 1

Introduction

Memories are one of the key bottlenecks in the performance, reliability and energy efficiency of most computing systems. As computing systems have scaled over the decades, the need for memory systems where large amount of data can be stored and retrieved efficiently have also risen rapidly. To achieve this, main memory systems have been scaled for maximum information density. Moore's Law has been the primary driver behind the phenomenal advances in computing capability of the past several decades. However, with technology scaling having reached the nanoscale era, integrated circuits, especially memory systems, are becoming increasingly sensitive to process variations leading to reliability and yield concerns.

1.1 Memory Reliability is Becoming a Key Concern

Memories have become the limiting factor in reliability of computing systems [3] because they are primarily designed to maximize bit storage density; this makes memories particularly sensitive to manufacturing process variation, environmental operating conditions, and aging-induced wearout [4, 5]. Unfortunately, errors in computing memories have also increased. In warehouse-scale computers, these errors have become expensive culprits that cause machine crashes, corrupted data, security vulnerabilities, service disruption, and costly repairs and hardware servicing [3, 6]. Google has observed 70000 failures in time (FIT)/Mb in commodity on-chip DRAM memory, with 8% of modules affected per year [3], while Facebook has found that 2.5% of their servers have experienced memory errors per month [7]. The Blue Waters supercomputer had 8.2% of the dual in-line memory modules (DIMMs) (modules that contain multiple RAM chips) encounter an error over the course of a 261 day study [8]. These trends are expected to continue to rise.

Moreover, with IoT devices increasingly becoming part of critical infrastructure and being deployed in failure-intolerant modes (e.g., cars), development of inexpensive fault tolerance schemes for them has become important [9]. Also, with sensing and data-processing being one of the most important use cases for edge devices, these devices are seeing increasing use of large memories. SRAM based scratchpad memories are often the choice of memory architecture used in IoT devices. As demand for higher memory density increases, memory cells are shrunk using advanced technology nodes which in turn makes the memory cells more susceptible to both soft and hard faults. Need for low-power and hence lower operating voltage exacerbates the error rates further. These trends indicate that memory failures are likewise going to be critical for emerging edge/IoT computing devices as well.

1.2 Power/Performance Scaling and Fault Tolerance in On-Chip SRAM Based Memories

Low power density is the key to achieving the vision of both exascale computing and the Internet of Things (IoT) [10]. To achieve that, systems need to adopt intelligent power-saving techniques. Memories, both on-chip and off-chip, consume a significant portion of system power. One way to reduce power consumption in on-chip SRAM based memories is to reduce the supply voltage (VDD). However, as shown in Figure 1.1, scaling the VDD down leads to an exponential rise in hard faults in the memory cells [11]. Not only hard faults, the memories also become more susceptible to radiation-induced soft faults at lower voltages, thus degrading yield at low voltage. Moreover, on-chip embedded memories or caches in high performance computing systems are often the largest consumers of chip area. This further increases the likelihood of defects affecting memory rather than logic and process variations with respect to individual memory cells create a significant impact.

To deal with on-chip memory errors due to manufacturing defects designers traditionally include spare rows and columns in the memory arrays [12] and employ large voltage guardbands [13] to ensure reliable operation. Unfortunately, large guardbands limit the energy proportionality of memory. For unpredictable runtime bit flips, the widely used technique to guarantee reliability of storage devices is using information redundancy in the form of Error Correcting Codes (ECC). In

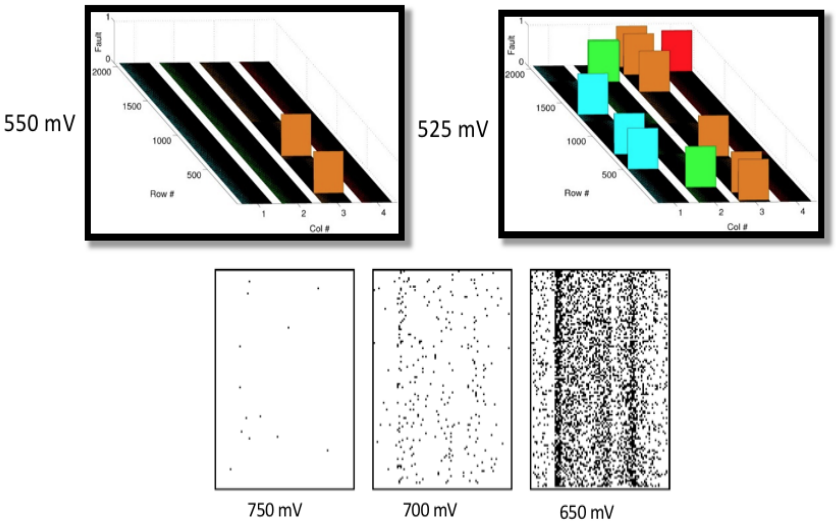


Figure 1.1: Faults in two SRAM based scratchpad memories at different voltages

typical ECCs, extra redundancy bits are added to every row to detect and correct errors. There are additional encoding (while writing data) and decoding (while reading data) procedures required as well. Thus, redundancy in ECC schemes not only incurs area overhead, the encoding and decoding mechanisms also incur additional overheads in terms of latency and energy.

1.3 Thesis Outline

Area, power and latency overheads of fault tolerance techniques are critical considerations for on-chip memories. This thesis primarily focuses on lightweight low overhead solutions for fault tolerance in on-chip scratchpad memories and last level caches . The organization and key contributions of this thesis are as follows:

- In Chapter 2, we develop FaultLink, a fault tolerance technique that tackles the problem of hard faults that appear at low voltages in software managed/scratchpad memories for embedded systems at the edge of the Internet-of-Things (IoT). It is a novel lazy link-time approach that extends the software construction toolchain with new fault-tolerance features for such memories. This approach builds application binary that is custom-tailored for each individual chip based on the chip's memory fault map so that the faulty locations in the

memory are never accessed when it is run at lower voltages.

- In Chapter 2, we also propose Software Defined Error Localization Code (SDELC), a hybrid hardware/software technique that allows the system to heuristically recover from unpredictable single-bit soft faults in instruction and data memories, which cannot be handled using FaultLink. SDELC, along with FaultLink provides a holistic approach in dealing with both hard and soft faults in software managed embedded memories.

- In Chapter 3, we propose Parity++, a novel unequal message protection scheme for last level caches that preferentially provides stronger error protection to certain "special messages". Parity++ sits in between basic Single Error Detecting(SED) parity and a full single error correcting Hamming code. This technique can be used not only for last level caches but can be extended to embedded memories.

- Chapter 4 concludes the thesis.

The techniques proposed in both the chapters can be extended to other system and memory technologies where such lightweight fault tolerance techniques to reduce chip real estate and power consumption are critically needed.

CHAPTER 2

Low Cost Fault Tolerance for IoT Devices

IoT devices need reliable hardware at low cost. It is challenging to efficiently cope with both hard and soft faults in embedded scratchpad memories. To address this problem, we propose a two-step approach: *FaultLink* and *Software-Defined Error-Localizing Codes* (SDELC). FaultLink avoids hard faults found during testing by generating a custom-tailored application binary image for each individual chip. During software deployment-time, FaultLink optimally packs small sections of program code and data into fault-free segments of the memory address space and generates a custom linker script for a lazy-linking procedure. During run-time, SDELC deals with unpredictable soft faults via novel and inexpensive *Ultra-Lightweight Error-Localizing Codes* (UL-ELCs). These require fewer parity bits than single-error-correcting Hamming codes. Yet our UL-ELCs are more powerful than basic single-error-detecting parity: they localize single-bit errors to a specific chunk of a codeword. SDELC then heuristically recovers from these localized errors using a small embedded C library that exploits observable *side information* (SI) about the application's memory contents. SI can be in the form of redundant data (value locality), legal/illegal instructions, etc. Our combined FaultLink+SDELC approach improves min-VDD by up to 440 mV and correctly recovers from up to 90% (70%) of random single-bit soft faults in data (instructions) with just three parity bits per 32-bit word.

Collaborators:

- Mark Gottscho, UCLA/Google
- Clayton Schoeny, UCLA
- Prof. Lara Dolecek, UCLA
- Prof. Puneet Gupta, UCLA

2.1 Introduction

For embedded systems at the edge of the Internet-of-Things (IoT), hardware design is driven by the need for the lowest possible cost and energy consumption, which are both are strongly affected by on-chip memories [14]. Memories consume significant chip area and are particularly susceptible to parameter variations and defects resulting from the manufacturing process [15]. Meanwhile, much of an embedded system's energy is consumed by on-chip SRAM memory, particularly during sleep mode. The embedded systems community has thus increasingly turned to software-managed on-chip memories – also known as *scratchpad memories* (SPMs) [16] – due to their 40% lower energy as well as latency and area benefits compared to hardware-managed caches [17].

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 [18]. However, this causes cell hard fault rates to rise exponentially [11] 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 [12] to deal with manufacturing defects and employ large voltage guardbands [13] to ensure reliable operation. Unfortunately, large guardbands limit the energy proportionality of memory, thus reducing battery life for duty-cycled embedded systems [19], a critical consideration for the IoT. Although many low-voltage solutions have been proposed for caches, fewer have addressed this problem for scratchpads and embedded main memory.

Our goal in this work is to improve embedded software-managed memory reliability at minimal cost; we propose a two-step approach. *FaultLink* first guards applications against known hard faults, which then allows *Software-Defined Error-Localizing Codes* (SDELC) to focus on dealing with unpredictable soft faults. **The key idea** of this work is to first automatically customize an application binary to individually accommodate each chip's unique hard fault map with no disruptions to source code, and second, to deal with single-bit soft faults at run-time using novel *Ultra-Lightweight Error-Localizing Codes* (UL-ELC) with a software-defined error handler that knows about the UL-ELC construction and implements a heuristic data recovery policy. The contributions of this chapter are the following.

- We present FaultLink, a novel lazy link-time approach that extends the software construction toolchain with new fault-tolerance features for software-managed/scratchpad memories. FaultLink relies on hard fault maps for each software-controlled physical memory region that may be generated during manufacturing test or periodically during run-time using built-in-self-test (BIST).

- We detail an algorithm for FaultLink that automatically produces custom hard fault-aware linker scripts for each individual chip. We first compile the embedded program using specific flags to carve up the typical monolithic sections, e.g., `.text`, `.data`, stack, heap, etc. FaultLink then attempts to optimally pack program sections into memory segments that correspond to contiguous regions of non-faulty addresses.

- We propose SDELC, a hybrid hardware/software technique that allows the system to heuristically recover from unpredictable single-bit soft faults in instruction and data memories, which cannot be handled using FaultLink. SDELC relies on *side information* (SI) about application memory contents, i.e., observable patterns and structure found in both instructions and data. SDELC is inspired by our recently-proposed notion of Software-Defined ECC (SDECC) [20].

- We describe the novel class of *Ultra-Lightweight Error-Localizing Codes* (UL-ELC) that are used by SDELC. UL-ELC codes are stronger than basic single-error-detecting (SED) parity, yet they have lower storage overheads than a single-error-correcting (SEC) Hamming code. Like SED, UL-ELC codes can detect single-bit errors, yet they can additionally *localize* them to a *chunk* of the erroneous codeword. UL-ELC codes can be explicitly designed such that chunks align with meaningful message context, such as the fields of an encoded instruction.

By experimenting with both real and simulated test chips, we find that with no hardware changes, FaultLink enables applications to run correctly on embedded memories using a min-VDD that can be lowered by up to 440 mV. After FaultLink has avoided hard faults (that may include defects as well as voltage-induced faults), our SDELC technique recovers 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). SDELC can even be used to recover up to 70% of errors using a basic SED parity code (3.125% storage overhead). In contrast, a full Hamming SEC code

incurs a storage overhead of 18.75%. *Our combined FaultLink+SDELC approach could thus enable more reliable IoT devices while significantly reducing cost and run-time energy.*

To the best of our knowledge, this is the first work to both (i) customize an application binary on a per-chip basis by lazily linking at software deployment-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 that is necessary to understand our contributions is presented in Sec. 2.2. We then describe the high-level ideas of FaultLink and SDELC to achieve low-cost embedded fault-tolerant memory in Sec. 2.3. FaultLink and SDELC are each described in greater detail in Secs. 2.4 and 2.5, respectively. Both FaultLink and SDELC are evaluated in Sec. 2.6. We provide an overview of related work in Sec. 2.7 before discussing other considerations and opportunities for future work in Sec. 2.8. We conclude the chapter in Sec. 2.9.

2.2 Background

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.

2.2.1 Scratchpad Memories (SPMs)

Scratchpad memories (SPMs) are small on-chip memories that, like caches, can help speed up memory accesses that exhibit spatial and temporal locality. Unlike caches, which are hardware-managed and are thus transparent in the address space, data placement in scratchpads must be orchestrated by software. This requires additional effort from the application programmer, who must – with the help of tools like the compiler and linker – explicitly partition data into physical memory regions that are distinct in the address space. Despite the programming difficulty, SPMs can be more efficient than caches. Banakar et al. showed that SPMs have on average 33% lower area requirements and can reduce energy by 40% compared to equivalently-sized caches [17]. In

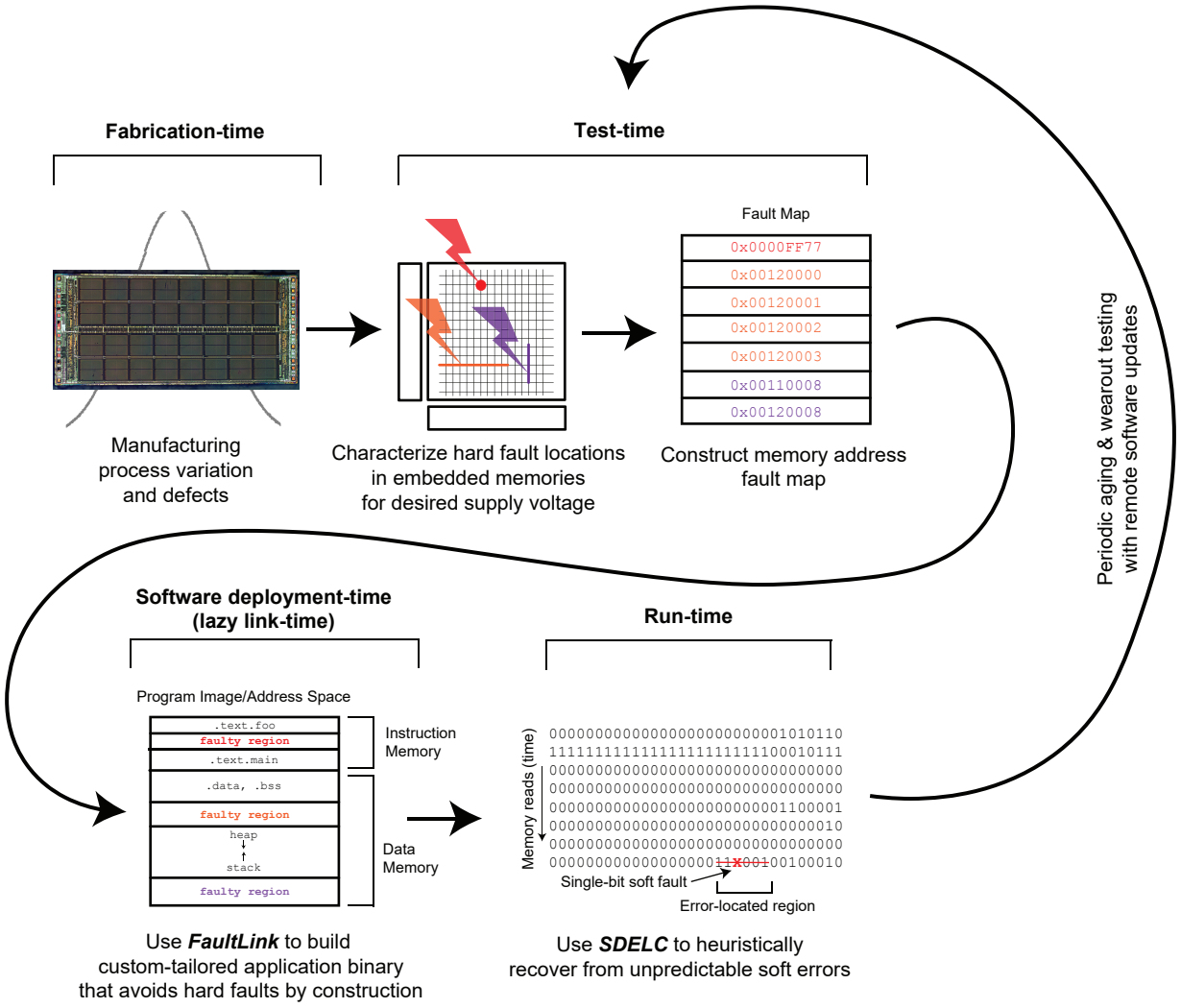


Figure 2.1: Our high-level approach to tolerating both hard (*FaultLink*) and soft (*SDELC*) faults in on-chip scratchpad memories.

energy and cost-conscious embedded systems, SPMs are increasingly being used for this reason and because they provide more predictable performance. In this work, FaultLink is used to improve the reliability/min-VDD of SPMs/software-managed main memory.

2.2.2 Program Sections and Memory Segments

The Executable and Linkable Format (ELF) is ubiquitous on Unix-based systems for representing compiled object files, static and dynamic shared libraries, as well as program executable images in a portable manner [21]. ELF files contain a header that specifies the ISA, ABI, a list of program

sections and memory segments, and various other metadata.

- 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, instruction 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. Manipulating the mapping between program sections and segments is the core focus of FaultLink.

2.2.3 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 [22]. 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 reoccur 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 [23]. Soft faults, if detected and corrected by an *error-correcting code* (ECC), are harmless to the system. In this work, SDELC is used to tolerate single-bit SEUs in a heuristic manner that has significantly lower overheads than a conventional ECC approach, yet can do more than basic SED parity detection.

2.2.4 Error-Correcting Codes (ECCs)

ECCs are mathematical techniques that transform *message* data stored in memory into *codewords* using a hardware encoder to add redundancy for added protection against faults. 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 $\mathbf{G}$ and the complementary decoder implements the parity-check matrix $\mathbf{H}$ to detect/correct errors. To encode a binary message $\vec{m}$, one multiplies its bit-vector by $\mathbf{G}$ to obtain the codeword $\vec{c}$: $\vec{m}\mathbf{G} = \vec{c}$. To decode, one multiplies 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 to the end of the message. This makes it easy to directly extract message data in the common case when no errors occur.

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 [24] have unacceptable overheads and are not suited for embedded memories. In this work, we propose novel ECC constructions that have very low overheads, making them suitable for low-cost IoT devices that may experience occasional single-bit SEUs.

2.2.5 Error-Localizing Codes

In 1963, Wolf et al. introduced *error-localizing codes* (ELC) that attempt to detect errors and identify the erroneous fixed-length chunk of the codeword. Wolf established some fundamental bounds [25] and studied how to create them using the tensor product of the parity-check matrices of an error-detecting and an error-correcting code [26]. ELC has been adapted to byte-addressable memory systems [27] but until now, they had not gained any traction in the systems community.

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 ELC* (UL-ELC) that lies in this regime and apply specific constructions to recover from a majority of single-bit soft faults.

2.3 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 SPMs.

The high-level concept is illustrated in Fig. 2.1. 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 that is stored in a database for later. When the system developer later deploys the application software onto the devices, FaultLink is used to customize the binary for each individual chip in a way that avoids its unique hard fault locations. Finally, at run-time, unpredictable soft faults are detected, localized, and recovered heuristically using SDELC.

Note that FaultLink is not heuristic and therefore does not induce errors. On the other hand, SDELC has a chance of introducing silent data corruption (SDC) if recovery turns out to be incorrect; this consideration will be revisited later in the discussion. We briefly explain the approaches of the FaultLink and SDELC steps before going into greater detail for each.

2.3.1 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 software-managed memories are characterized to identify memory addresses that contain hard

faults.

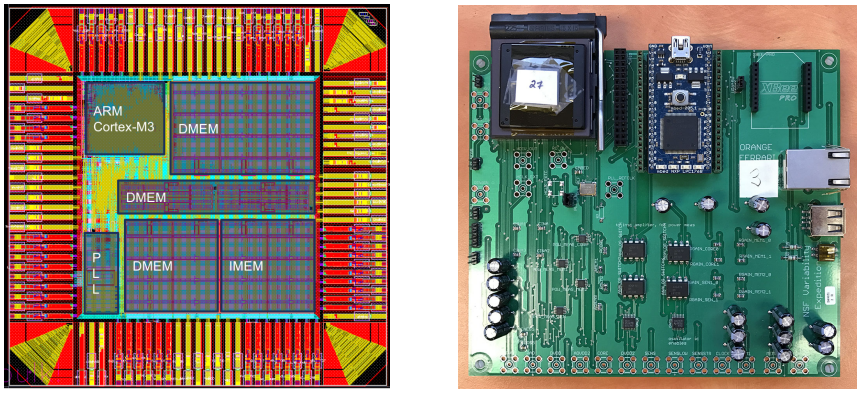
At software deployment time – i.e., when the application is actually programmed onto a particular device – FaultLink customizes the application binary image to work correctly on that particular chip given the fault map as an input. FaultLink does this by linking the program to guarantee that no hard-faulty 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.

2.3.2 Software-Defined Error-Localizing Codes (SDELC): Recovering Soft Faults at Run-Time

Typically, either basic SED parity is used to detect random single-bit errors or a Hamming SEC 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 by other techniques it cannot correct any errors.

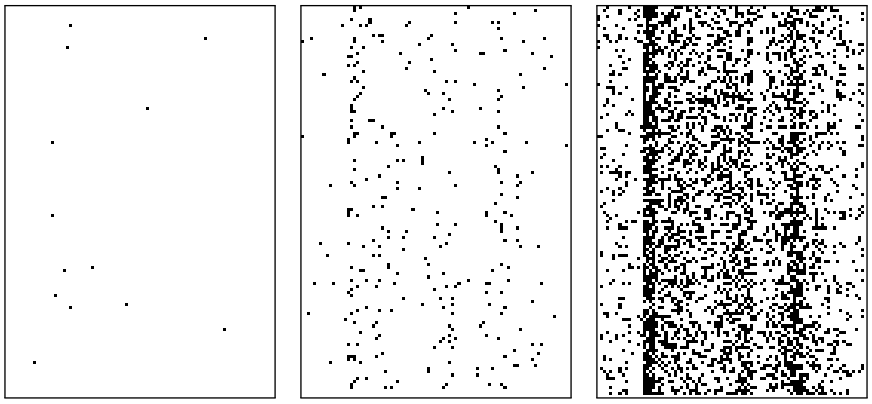
SDELC is a novel solution that lies in between these regimes. A key component is the new class of *Ultra-Lightweight Error-Localizing Codes* (UL-ELCs). UL-ELCs have lower storage overheads than Hamming codes: they can detect and then *localize* any single-bit error to a chunk of a memory codeword. We 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 a basic SED parity code. Unlike our recent preliminary work on general-purpose Software-Defined ECC (SDECC) [20], SDELC focuses on heuristic error recovery that is suitable for microcontroller-class IoT devices.

We now discuss FaultLink in greater depth before revisiting the details of SDELC in Sec. 2.5.



(a) Chip floorplan **(b)** Board

Figure 2.2: Test chip and board used to collect hard fault maps for FaultLink.



(a) 750 mV **(b)** 700 mV **(c)** 650 mV

Figure 2.3: Measured voltage-induced hard fault maps of the 176 KB data memory for one test chip. Black pixels represent faulty byte locations.

2.4 FaultLink

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 solves.

2.4.1 Test Chip Experiments

To motivate FaultLink, we characterized the voltage scaling-induced fault maps for eight microcontroller test chips. Each chip contains a single ARM Cortex-M3 core, 176 KB of on-chip data memory, 64 KB of instruction memory. They were fabricated in a 45nm SOI technology with dual-Vth libraries [28, 29, 30]; the chip floorplan and test board are shown in Fig. 2.2. The locations of voltage-induced SRAM hard faults in the data memory for one chip are shown in Fig. 2.3 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.

2.4.2 Toolchain

FaultLink utilizes the standard GNU tools for C/C++ without modification. The overall procedure is depicted in Fig. 2.4. The programmer compiles code into object files but does not 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. Our FaultLink tool then uses the ELFIO C++ library [31] 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.

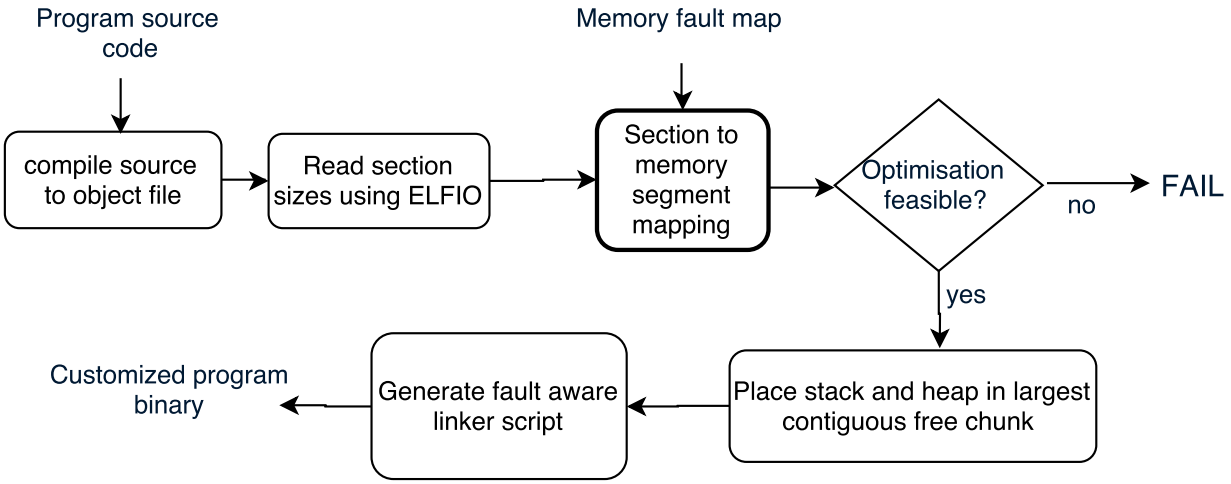


Figure 2.4: FaultLink procedure: given program source code and a memory fault map, produce a per-chip custom binary executable that will work in presence of known hard fault locations in the SPMs.

2.4.3 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.

2.4.3.1 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. 2.5, 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 to segment j, otherwise let it be 0. Then 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 use an objective that minimizes the number of packed segments because the solution naturally avoids memory regions that have higher fault densities. The constraints ensure that every program section gets packed in the non-faulty segments of the memory and the total size of all the sections packed in one non-faulty segment is no more than the size of that particular segment. (Note that other objectives will produce equally-valid section-packing solutions in terms of correctness; the important fault-avoidance constraints are fixed.) 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. If a faster solution is needed, a greedy ILP relaxation can be used.

2.4.3.2 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, and that most FaultLink section-packing failures occur when the largest program section is larger than all non-faulty memory segments.

Therefore, we estimate the FaultLink success rate based on the probability distribution of the longest consecutive sequences of coin flips as provided by Schilling [32]. Let L_k be a random variable representing the length of the largest run of heads in k independent flips of a biased coin

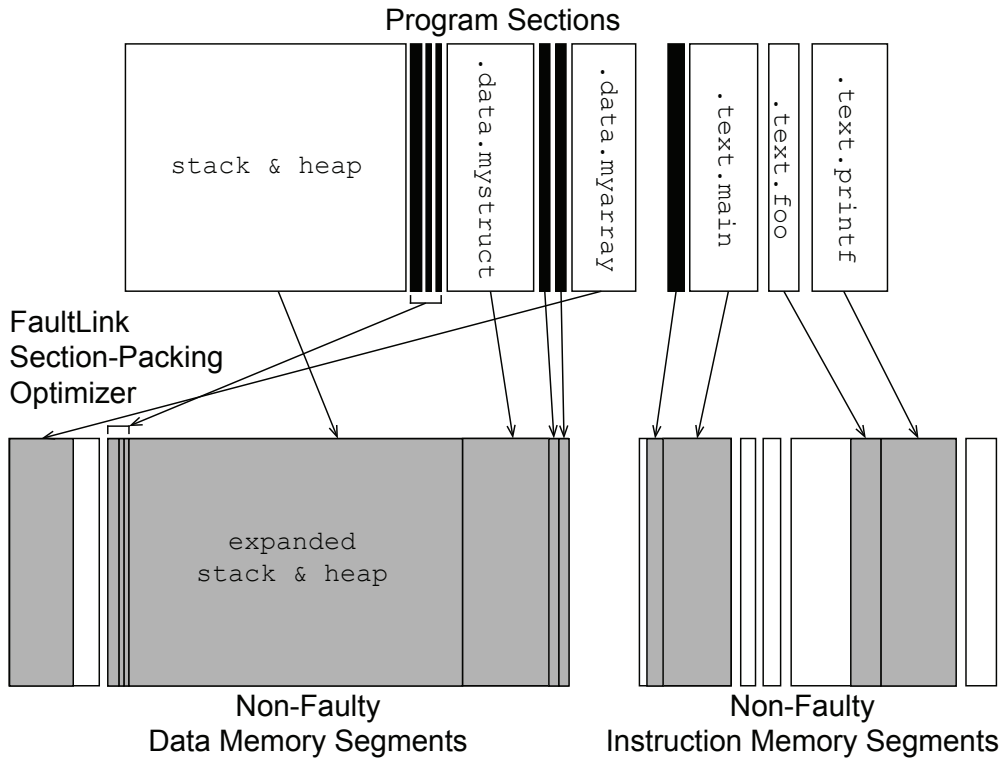


Figure 2.5: FaultLink attempts to pack contiguous program sections into contiguous disjoint segments of non-faulty memory. Gray memory segments are occupied by mapped sections, while white segment areas are free space. The depicted gaps between some of the gray/white boxes indicate faulty memory regions that are not available for section-packing.

(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 $k(1-p) \gg 1$ [32]:

$$P(L_k < x) \approx e^{-p^{(x-\log_{p^{-1}}(k(1-p)))}}. \tag{2.1}$$

We apply Schilling's above 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. 2.1, we plug in $p = s$, $k = \texttt{size}/4$, and $x = m_{\max}/4$. Then, we can approximate the probability of there *not* being a memory segment that is large enough to store the largest program section:

$$P\left(L_{\texttt{size}/4} < \frac{m_{\max}}{4}\right) \approx e^{-s^{\left(\frac{m_{\max}}{4} - \log_{s^{-1}}\left(\frac{\texttt{size}}{4}(1-s)\right)\right)}}. \tag{2.2}$$

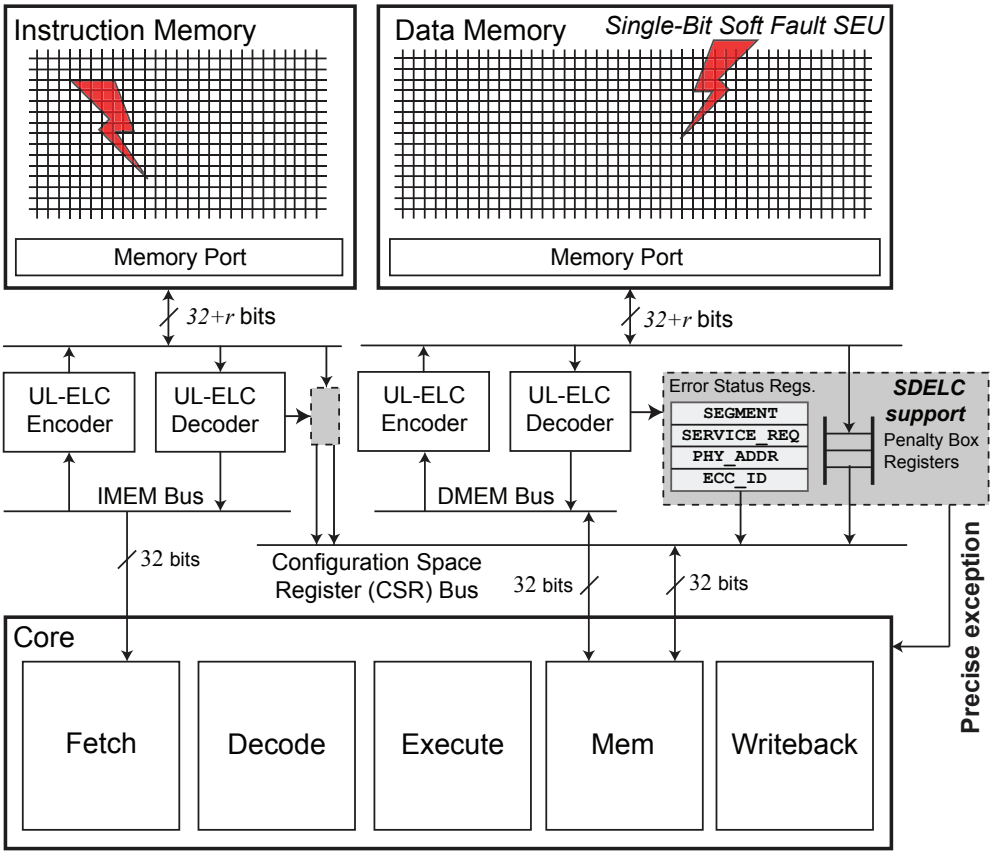


Figure 2.6: Architectural support for SDELC on an microcontroller-class embedded system. Hard faults that would be managed by FaultLink are not shown.

This formula will be used in the evaluation to estimate FaultLink yield and min-VDD.

2.5 SDELC

We describe the SDELC architecture, the concept of UL-ELC codes, and two SDELC recovery policies for instruction and data memory.

2.5.1 Architecture

The SDELC architecture is illustrated in Fig. 2.6 for a system with split on-chip instruction and data SPMs (each with its own UL-ELC code) and a single-issue core that has an in-order pipeline. We assume that hard faults are already mitigated using FaultLink.

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.

2.5.2 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 segmented 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 and 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. (A minimum codeword distance of two bits is required for SED, while three bits are needed for SEC, etc.)

For *an example* of an UL-ELC construction, consider the following $\mathbf{H}_{\text{example}}$ parity-check matrix with nine message bits and $r = 3$ parity bits:

$$\mathbf{H}_{\text{example}} = \begin{array}{c} \\ \\ c_1 \\ c_2 \\ c_3 \end{array} \begin{array}{c} \begin{array}{cccccccccccc} S_1 & S_2 & S_3 & S_4 & S_4 & S_5 & S_6 & S_6 & S_7 & S_5 & S_6 & S_7 \\ d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 & p_1 & p_2 & p_3 \end{array} \\ \left[\begin{array}{cccccccccccc} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \end{array},$$

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 S_l enumerates the distinct error-localizing chunk that a given bit belongs to. Because $r = 3$, there are $N = 7$ chunks. Bits $d_1, d_2,$ and d_3 each have the SEC property because no other bits are in their respective chunks. Bits d_4 and d_5 make up an unshared chunk S_4 because no parity bits are included in S_4. The remaining data bits belong to shared chunks because each of them also

Table 2.1: Proposed 7-Chunk UL-ELC Construction with $r = 3$ for Instruction Memory (RV64G ISA v2.0)

bit →	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

includes at least one parity bit. Notice that any data or parity bits that belong to the same chunk S_l have identical columns of $\mathbf{H}$, e.g., d_7, d_8, and p_2 all belong to S_6 and have the column $[0;1;0]$.

The two key properties of UL-ELC (that do not apply to generalized ELC codes) 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.

2.5.3 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 SDELC implementation targets the open-source and free 64-bit RISC-V (RV64G) ISA [33], 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 work.

Our UL-ELC construction for instruction memory has seven chunks that align to the finest-grain

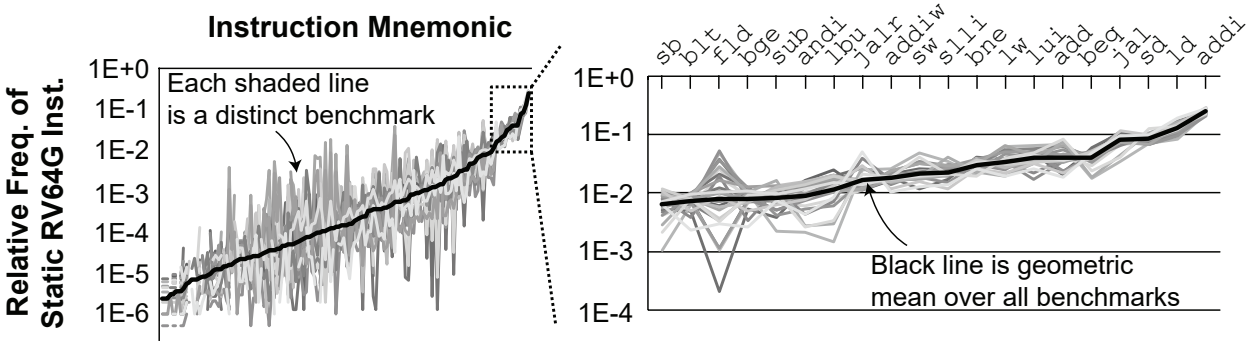


Figure 2.7: The relative frequencies of static instructions roughly follow power law distributions. Results shown are for RISC-V with 20 SPEC CPU2006 benchmarks; we observed similar trends for MIPS and Alpha, as well as dynamic instructions.

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 2.1. 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 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 2.1, 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 bit 7 (the least-significant bit of chunk C_6/`rd`) is flipped by a fault. Assume the original instruction stored in memory was `0x0000beef`, 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.
```

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). The instruction recovery policy consists of three steps.

- Step 1. We apply a software-implemented instruction decoder to filter out any candidate messages that are illegal instructions. Most bit patterns decode to illegal instructions in three RISC ISAs we characterized: 92.33% for RISC-V, 72.44% for MIPS, and 66.87% for Alpha. This can be used to dramatically improve the chances of a successful SDELC recovery.
- 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. We find that the relative frequencies of legal instructions follow power-law distribution, as shown by Fig. 2.7. This is used to favor more common instructions.
- 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. This is because in many instructions, the MSBs represent immediate values (as shown in Table 2.1). These MSBs are usually low-magnitude signed integers or they represent 0-dominant function codes.

If the SI is strong, then we would expect to have a high chance of correcting the error by choosing the right candidate.

2.5.4 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 [34].

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* [35] that has been used in numerous works for cache and memory compression [36, 37, 38]. The Hamming distance is a good measure of data correlation, as shown later in Fig. 2.13.

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. 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 [39, 40, 41, 42, 43, 44].

2.6 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 software-managed memories.

2.6.1 Hard Fault Avoidance using FaultLink

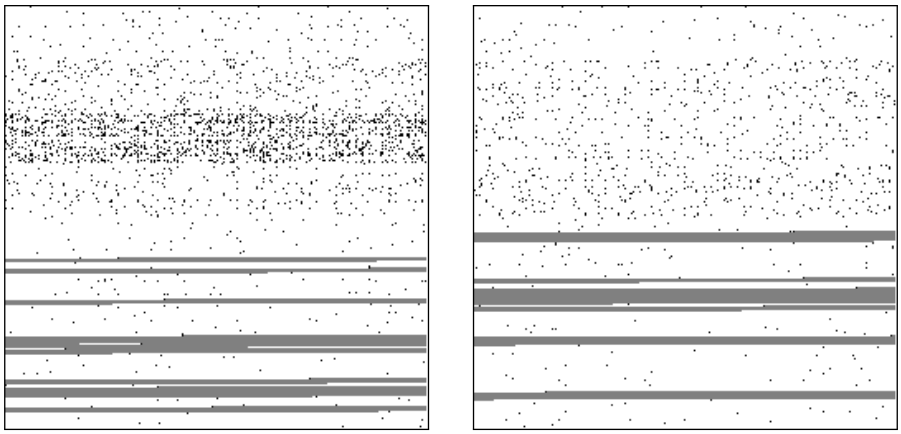
We first demonstrate how applications can run on real test chips at low voltage with many hard faults in on-chip memory using FaultLink, and then evaluate the yield benefits at low voltage for a synthetic population of chips.

2.6.1.1 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 [45] 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 [46] 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. 2.8 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. 2.8. 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. 2.8a) also appears to have a cluster of weak rows in the first instruction bank. Because FaultLink chooses a



(a) Chip 1 **(b)** Chip 2

Figure 2.8: Result from applying FaultLink to the `sha` benchmark for two real test chips' 64 KB instruction memory at 650 mV.

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. FaultLink did not require more than 14 seconds on our machine to optimally section-pack any program for any chip at any voltage.

2.6.1.2 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 [44] that are too big to fit on the real test chips: `blackscholes`, `fft`, `inversek2j`, `jmeint`, `jpeg`, and `sobel`. These AxBench benchmarks were compiled for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [33] and privileged specification v1.7 using the official tools. This is because

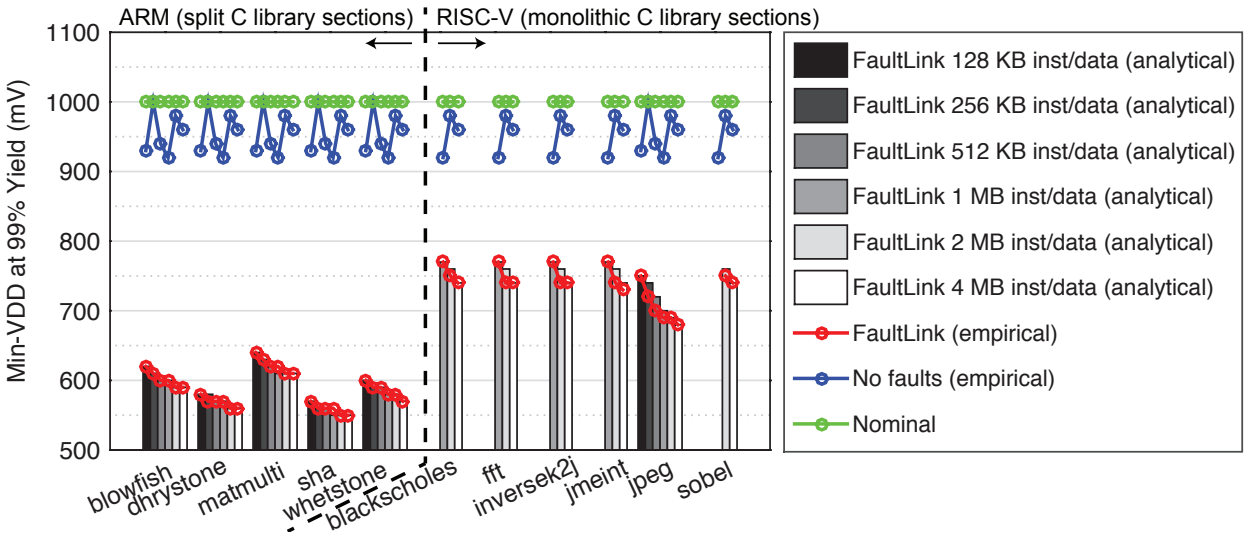


Figure 2.9: Achievable min-VDD for FaultLink at 99% yield. Bars represent the analytical lower bound from Eqn. 2.2 and circles represent our actual results using Monte Carlo simulation for 100 synthetic fault maps.

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. 2.9. The vertical bars represent our analytical estimates calculated using Eqn. 2.2. 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.

The voltage-scaling limits are nearly always determined by the length of the longest program

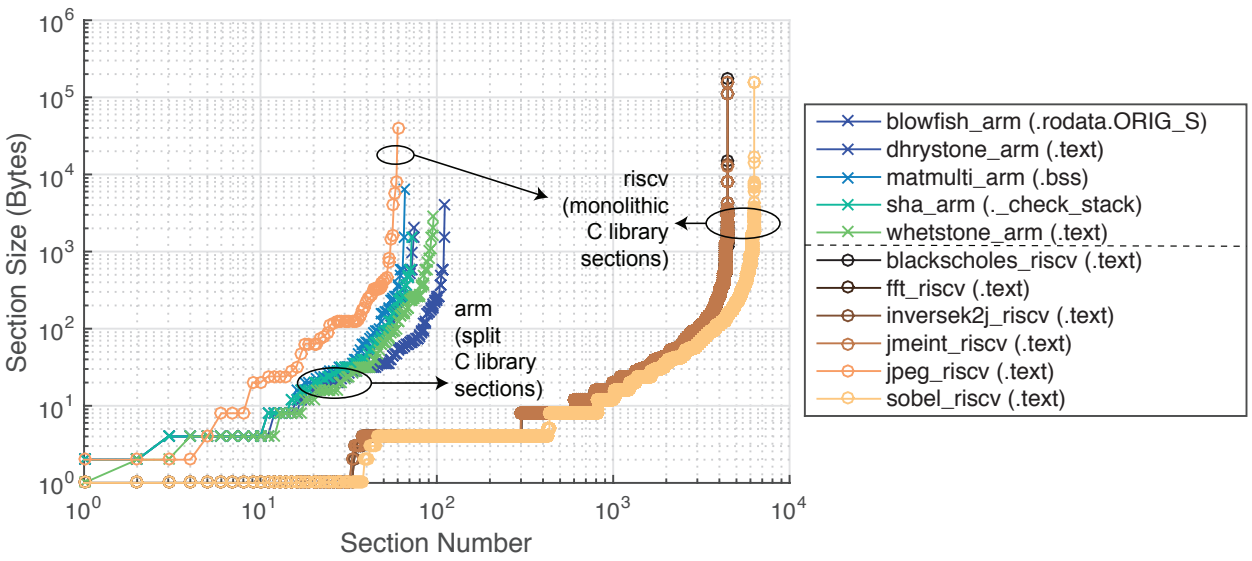


Figure 2.10: Distribution of program section sizes. Packing the largest section into a non-faulty contiguous memory segment is the most difficult constraint for FaultLink to satisfy and limits min-VDD.

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. 2.10. 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. 2.9. 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. For instance, `jpeg` compiled on RISC-V achieves a min-VDD of 750mV for 128 KB memory, while on ARM (not depicted) it achieves a min-VDD of 660mV.

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.

2.6.2 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.

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 2.1). A *successful recovery* for SDELC occurs when the policy corrects the error; otherwise, it fails by accidentally mis-correcting.

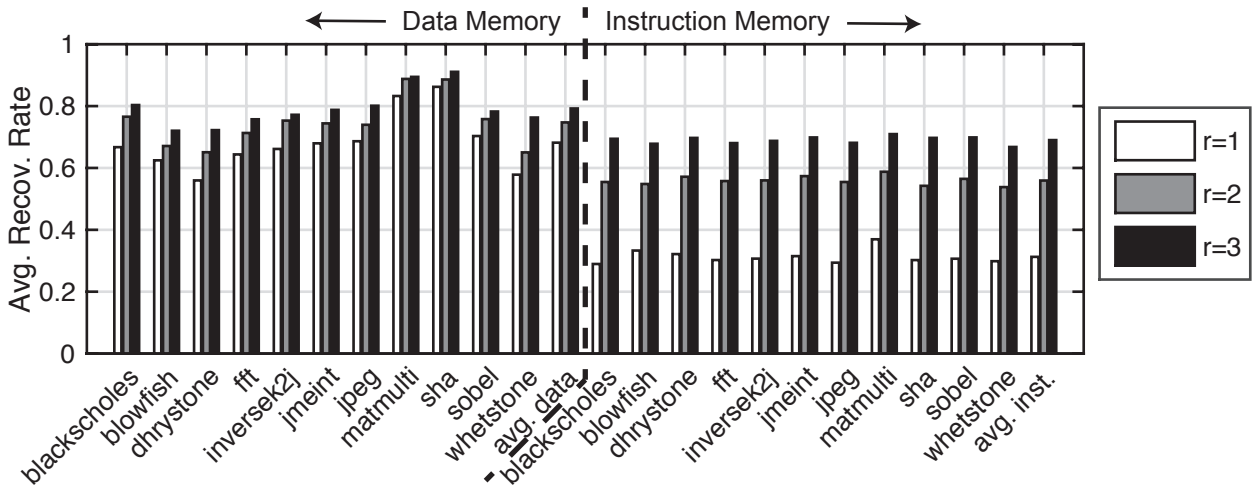


Figure 2.11: Average rate of recovery using SDELC from single-bit soft faults in data and instruction memory. Benchmarks have already been protected against known hard fault locations using FaultLink. r is the number of parity bits in our UL-ELC construction.

2.6.2.1 Overall Results

The overall SDELC results are presented in Fig. 2.11. 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 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 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.

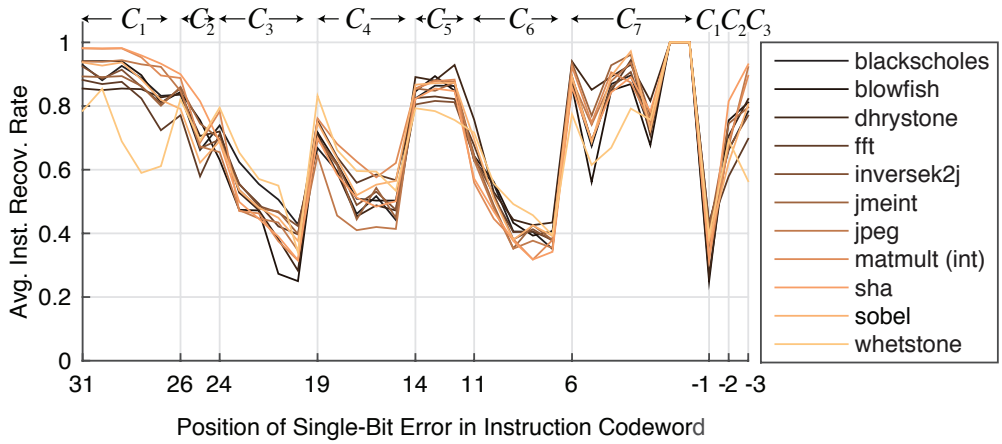
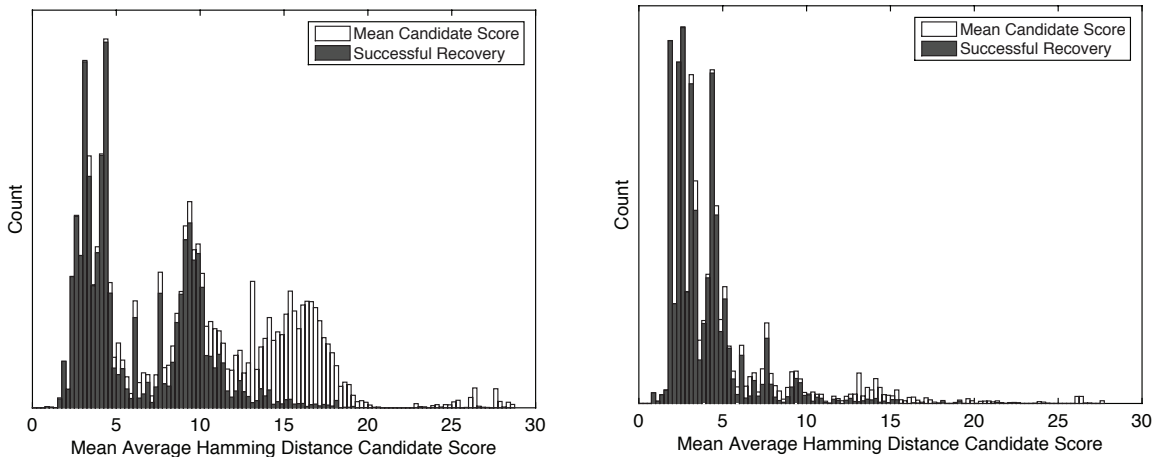


Figure 2.12: Sensitivity of SDELC instruction recovery to the actual position of the single-bit fault with the $r = 3$ UL-ELC construction.

2.6.2.2 Recovery Policy Analysis

The average instruction recovery rate as a function of bit error position for all benchmarks is shown in Fig. 2.12. Error positions -1, -2, and -3 correspond to the three parity bits in our UL-ELC construction from Table 2.1.

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



(a) `blowfish` **(b)** `sha`

Figure 2.13: Sensitivity of SDELC data recovery to the mean candidate Hamming distance score for two benchmarks and $r = 1$ parity code.

C_1 which often represents the most-significant bits of an encoded immediate value.

The sensitivity of SDELC data recovery to the mean candidate Hamming distance score for two benchmarks is shown in Fig.2.13. 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 [35] in memory, we thus recover correctly in a majority of cases. On the other hand, when data has very low correlation – essentially random information — SDELC does not recover any better than taking a random guess of the bit-error position within the localized chunk, as expected.

2.7 Related Work

We summarize and differentiate our contributions from related work on fault-tolerant caches and scratchpads, as well as error-localizing and unequal error protection codes.

2.7.1 Fault-Tolerant Caches

There is an abundance of prior work on fault-tolerant and/or low-voltage caches. Examples include PADded Cache [47], Gated-VDD [48], Process-Tolerant Cache [49], Variation-Aware Caches [50], Bit Fix/Word Disable [51], ZerehCache [52], Archipelago [53], FFT-Cache [54], VS-ECC [55], Correctable Parity Protected Cache (CPPC) [56], FLAIR [57], Macho [58], DPCS [59], DARCA [60], and others (see related surveys by Mittal [61, 4]). These fault-tolerant cache techniques tolerate hard faults/save energy by sacrificing capacity or remapping physical data locations. This affects the software-visible memory address space and hence they cannot be readily applied to SPMs.

Although they are cache-specific, some of the above techniques can be roughly compared with FaultLink in terms of min-VDD. For instance, DPCS [59] achieves a similar min-VDD to FaultLink of around 600 mV, while FLAIR [57] achieves a lower min-VDD (485 mV). We emphasize that the above techniques cannot be applied to SPMs and are therefore not a valid comparison.

Similar to SDELC, CPPC [56] can recover random soft faults using SED parity. However, CPPC requires additional hardware bookkeeping mechanisms that are in the critical path whenever data is added, modified, or removed from the cache (and again, their method is not applicable to SPMs).

2.7.2 Fault-Tolerant Scratchpads

The community has proposed various methods for tolerating variability and faults in SPMs that relate closely to this work. Traditional fault avoidance techniques like dynamic bit-steering [62] and strong ECC codes are too costly for small embedded memories. Meanwhile, spare rows and columns cannot scale to handle many faults that arise from deep voltage scaling.

E-RoC [41] is a SPM fault-tolerance scheme that aims to dynamically allocate scratchpad space to different applications on a multi-core embedded SoC using a virtual memory approach. However, it requires extensive hardware and run-time support. Several works [63, 40, 42, 43] propose to use OS-based virtual memory to directly manage memory variations and/or hard faults, but they are not feasible in low-cost IoT devices that lack support for virtual memory; nor do they guarantee avoidance of known hard faults at software deployment time. Others have proposed to add small fault-tolerant buffers that assist SPM checkpoint/restore [64], re-compute corrupted data upon

detection [65], build radiation-tolerant SPMs using hybrid non-volatile memory [66] and duplicate data storage to guard against soft errors [67]; these are each orthogonal to this work.

There are several other prior works that relate closely to SDELC, although ours is the first to propose heuristic recovery that lies in the regime between SED and SEC capabilities. Farbeh et al. [68] propose to recover from soft faults in instruction memory by leveraging basic SED parity combined with a software recovery handler that leverages duplicated instructions in memory. On the other hand, our approach does not add any storage overhead to recover from errors (although ours is heuristic). Volpato et al. [69] proposed a post-compilation binary patching approach to improve energy efficiency in SPMs that closely resembles the FaultLink procedure. However, that work did not deal with faults in the SPMs. Sayadi et al. [65] uses SED parity to dynamically recompute critical data that that is affected by single-bit soft faults. SDELC completely subsumes that approach: the embedded SDELC library can heuristically recover data, recompute it if possible, or opt to panic according to the application's needs.

Unlike all known prior work, our combined FaultLink+SDELC approach can simultaneously deal with both hard and soft SPM faults with minimal hardware changes compared to existing IoT systems. Our low-cost approach can be used today with off-the-shelf microcontrollers (minor changes are needed to implement UL-ELC codes, however), and can improve yield and min-VDD.

2.8 Discussion

We highlight several considerations and beneficial use cases for FaultLink and SDELC and outline directions for future work.

2.8.1 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 instructions, which takes a few μs on a typical microcontroller (the number of instructions 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.

2.8.2 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 [70, 71, 72, 73].

2.8.3 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 memory 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.

2.8.4 Risk of SDCs from SDELC

SDELC introduces a risk of mis-correcting single-bit soft faults that cannot be avoided unless one resorts to 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× compared to Hamming while still recovering most single-bit faults. Similar to observations by others [74], 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., an SDC with more than 10% output error [44]. The rest of the faults are either successfully corrected, benign, or cause crashes/hangs. The latter are no worse than crashes from commonly-used SED parity. Current SED-based systems' reliability could be improved with remote software updates to incorporate our techniques.

2.8.5 Directions for Future Work

The FaultLink and SDELC approaches can be further improved upon. One could extend FaultLink to accommodate hard faults within packed sections to reduce min-VDD and increase reliability. For FaultLink with instruction memory, one approach could be to insert unconditional jump instructions to split up basic blocks, similar to a recent cache-based approach [75]. For FaultLink with data memory, one could use smaller split stacks [76] and design a fault-aware `malloc()`. For SDELC, one could design more sophisticated recovery policies using stronger forms of SI, and use profiling methods to automatically annotate program regions that are likely to experience faults.

2.9 Conclusion

We proposed FaultLink and SDELC, two complementary techniques to 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 with SDELC.

CHAPTER 3

Parity++: Lightweight Error Correction for Last Level Caches

As the size of on-chip SRAM caches is increasing rapidly and the physical dimension of the SRAM devices is decreasing, reliability of caches is becoming a growing concern. This is because with increased size of caches, the likelihood of radiation-induced soft faults also increases. As a result, information redundancy in the form of Error Correcting Codes (ECC) is becoming extremely important, especially to protect the larger sized last level caches (LLCs). In typical ECCs, extra redundancy bits are added to every row to detect and correct errors. There is additional encoding (while writing data) and decoding (while reading data) procedures required as well. In caches, these additional area, power and latency overheads need to be minimized as much as possible. To address this problem, we present in this chapter Parity++: a novel unequal message protection scheme for last level caches that preferentially provides stronger error protection to certain "special messages". This protection scheme provides Single Error Detection (SED) for all messages and Single Error Correction (SEC) for a subset of messages. Thus, it is stronger than just a basic SED parity and has $\sim$9% lower storage overhead and much lower error detection energy than a traditional Single Error Correcting, Double Error Detecting (SECDED) code. We also propose a memory speculation procedure that can be used with any ECC scheme to hide the decoding latency while reading messages when there are no errors.

Collaborators:

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

3.1 Introduction

As demand and size of on-chip caches is increasing rapidly and the physical dimension and noise margins are decreasing, reliability of caches is increasingly becoming an important issue. As given in [77, 78], the vulnerability of SRAM caches to soft errors grows with increase in size. Also with reduction in physical dimensions of these devices, the critical charge required to flip the content of a cell due to a particle strike decreases. As a result, the soft error rate is higher for large capacity caches. The widely used technique to guarantee reliability of storage devices is using information redundancy in the form of Error Correcting Codes (ECC). In typical ECCs, extra redundancy bits are added to every row to detect and correct errors. There are additional encoding (while writing data) and decoding (while reading data) procedures required as well. Thus ECCs come with encoding and decoding mechanisms that incur additional overheads in terms of latency and energy. Both these overheads are critical for caches and hence, ECC protection was not widely used in caches till recently. However, due to the increased reliability concerns of large capacity caches and processor performance degradation due to occurrence of errors, cache protection using ECC schemes is becoming increasingly popular. Nevertheless, these additional area, power and latency overheads need to be minimized in caches as much as possible.

We present Parity++: a novel unequal message protection scheme for last level caches that preferentially provides stronger error protection to certain "special messages". As the name suggests, this coding scheme requires one extra bit above a simple parity Single Error Detection (SED) code while providing SED for all messages and Single Error Correction (SEC) for a subset of messages. Thus, it is stronger than just basic SED parity and has $\sim$9% lower storage overhead than a traditional Single Error Correcting, Double Error Detecting (SECDED) code. Error detection circuitry often lies on the critical path and is generally more critical than error correction circuitry as error occurrences are rare even with an increasing soft error rate. Our coding scheme has a much simpler error detection circuitry that incurs lower energy and latency costs than the traditional SECDED code. Thus, Parity++ is a lightweight ECC code that is ideal for large capacity last level caches. We also propose a memory speculation procedure that can be generally applied to any ECC protected cache to hide the decoding latency while reading messages when there are no errors.

3.2 Background and Related Work

3.2.1 Error Correcting Codes

Error-correcting codes (ECCs) increase the resiliency of communication and storage systems by adding redundant bits (or symbols, but in this work we focus on the binary regime). A code $\mathscr{C}$ can be thought of as an injective mapping of *messages* of length k to *codewords* of length n. Let r be the number of redundant bits, i.e., $r = n - k$. A binary code is considered *linear* if the sum of any two codewords in $\mathscr{C}$ is also a codeword in $\mathscr{C}$.

A linear block code is described by either its $(k \times n)$ *generator matrix* $\mathbf{G}$ or its $(r \times n)$ *parity-check matrix* $\mathbf{H}$, with the relation $\mathbf{G}\mathbf{H}^T = \mathbf{0}$. A particular message $\mathbf{m}$ is encoded to its corresponding codeword $\mathbf{c}$ by multiplying it with the generator matrix as follows: $\mathbf{m}\mathbf{G} = \mathbf{c}$. Each row of $\mathbf{H}$ is a parity-check equation that all codewords must suffice, thus $\mathbf{H}\mathbf{c}^{\mathrm{T}} = \mathbf{0}$. We define the *received vector* at the output of the channel as $\mathbf{y} = \mathbf{c} + \mathbf{e}$, in which $\mathbf{e}$ is the error-vector representing which bits have been flipped. The receiver calculates the *syndrome*, $\mathbf{s} = \mathbf{H}\mathbf{y}^{\mathrm{T}}$, and if $\mathbf{s} \neq \mathbf{0}$, then it is known that the received vector is not a valid codeword. At this point, the decoder can either attempt to determine the most likely originally transmitted codeword or it can simply raise a flag that an error was detected (depending on the system goals and design). We say a code is *systematic* if a message is directly embedded in the codeword, i.e., each message bit is equal to a specific codeword bit.

A useful parameter of a linear code is its *minimum distance*, d_{min}, which is the minimum Hamming distance between any two (non-identical) codewords. Additionally, since a linear code must include the $\mathbf{0}$ codeword, the minimum distance of a linear code is simply the minimum weight of any (non-zero) codeword in the code:

$$d_{min} = \min_{\substack{\mathbf{c_1},\mathbf{c_2} \in \mathscr{C}; \\ \mathbf{c_1} \neq \mathbf{c_2}}} [d_H(\mathbf{c_1}, \mathbf{c_2})] = \min_{\substack{\mathbf{c} \in \mathscr{C}; \\ \mathbf{c} \neq \mathbf{0}}} [wt(\mathbf{c})].$$

A linear code guarantees correction of up to $t = \lfloor \frac{1}{2}(d_{min} - 1) \rfloor$ bit-errors, or detection of up to $(d_{min} - 1)$ bit-errors (without any correction guarantees). For even values of d_{min}, a linear code simultaneously guarantees correction of up to t bit-errors and detection of up to $(t+1)$ bit-errors. Further explanation of the fundamental properties of codes can be found in classic textbooks [79, 80].

3.2.2 SRAM Reliability and Error Detection and Correction in Caches

As mentioned before, SRAM reliability concerns are growing. Although the soft error rate of SRAM cell has almost been constant at 10^{-3} FIT/bit [81, 82], the likelihood of a particle striking the array is increasing with increase in size. Most of the recent processors with large capacity caches have ECC protected L2 and/or L3 caches. Some of the common and recent examples include Qualcomm's Centriq 2400 processor [83], AMD's Athlon [84] and Opteron [85] processors as well as IBM Power 4 [86] processors. Most of the commercially available processors use traditional (72,64) SECDED [87] code on each 64-bit word in the cache line. A lot of past works have suggested decoupling error detection and correction mechanisms so as to reduce the complexity and overhead of error detection since that is more critical than error correction. In [88], the authors suggest using SRAM for only error detection and storing the ECC correction bits within the memory hierarchy to reduce the overhead. In another work on ECC in caches, the authors of [89] suggest protecting only those cache lines that have been recently used. Thus, they trade-off protection with area and energy. Some past works like [90] have also focused on ECC protection schemes for L1 cache.

3.2.3 Application Characteristics

Data or instructions in applications are generally very structured. Frequencies of instructions in most applications follow power law distribution [91]. This means that some instructions get more frequently accessed than the rest. If the opcode (that primarily determines the action taken by the instruction) in a certain instruction set architecture (ISA) is, for example, the first x bits, then the relative frequency of the opcodes of the common instructions are high. This means most instructions in the memory would have the same prefix of x-bits. Table 3.1 shows the fraction of the two most frequently occurring opcode over each of the benchmark suites. The benchmarks were compiled for 32-bit RISC-V (RV32G) [33] instruction set v2.0 were the least significant 7 bits are designated as the opcode. This is true not just for instructions but also for data. In most applications, the data in the memory is usually low-magnitude signed data of a certain data type. However, these values get represented inefficiently, for e.g., 4-byte integer type used to represent values that usually need only 1-byte. Thus, in most cases, the MSBs would be a leading-pad of 0s or 1s. Table 3.1 shows that, for a

wide range of data sets, most stored data starts with a leading pad of zeros. Our approach of utilizing these characteristics in applications complements recent research on data compression in cache and main memory systems such as frequent value/pattern compression [92, 93], base-delta-immediate compression [94] and bit-plane compression [95]. However, our main goal is to provide stronger error protection to these special messages that are chosen based on the knowledge of data patterns in context.

Table 3.1: Fraction of Special Messages per Benchmark Within Suite

	Top Two Most Freq Opcodes (Data Memory)		**First 6 bits are 0 (Instruction Memory)**	
Benchmark Suite	**Max**	**Mean**	**Max**	**Mean**
AxBench	0.51	0.46	0.92	0.86
SPEC CPU2006	0.56	0.37	0.99	0.89

3.3 Lightweight Error Correction Code

3.3.1 Theory

The code we developed in this work, which we call Parity++, is a type of *unequal message protection* code, in that we *a priori* designate specific messages to have extra protection against errors. As in [96], there are two classes of messages, normal and special, and they are mapped to normal and special codewords, respectively. When dealing with the importance or frequency of the underlying data, we refer to the messages; when discussing error detection/correction capabilities we refer to the codewords.

Codewords in Parity++ have the following error protection guarantees: normal codewords have single-error detection; special codewords have single-error correction. Let us partition the codewords in our code $\mathscr{C}$ into two sets, $\mathscr{N}$ and $\mathscr{S}$, representing the normal and special codewords, respectively. The minimum distance properties necessary for the aforementioned error protection

guarantees of Parity++ are as follows:

$$\min_{\mathbf{u},\mathbf{v}\in\mathscr{N},\mathbf{u}\neq\mathbf{v}} d_H(\mathbf{u},\mathbf{v}) \geq 2, \tag{3.1}$$

$$\min_{\mathbf{u}\in\mathscr{N},\mathbf{v}\in\mathscr{S}} d_H(\mathbf{u},\mathbf{v}) \geq 3, \tag{3.2}$$

$$\min_{\mathbf{u},\mathbf{v}\in\mathscr{S},\mathbf{u}\neq\mathbf{v}} d_H(\mathbf{u},\mathbf{v}) \geq 3. \tag{3.3}$$

A second defining characteristic of the Parity++ code, is that the length of a codeword is only two bits longer than a message, i.e., $n = k+2$. Comprehensive comparisons between Parity++ and other popular ECCs are included in some of the subsequent sections.

For the context of this work, let us assume that our Parity++ always has message length k as a power of 2. The overall approach to constructing our code is to create a Hamming subcode of a SED code [97]; when an error is detected, we decode to the neighboring special codeword. The overall code has $d_{min} = 2$, but a block in $\mathbf{G}$, corresponding to the special messages, has $d_{min} \geq 3$. For the sake of notational convenience, we will go through the steps of constructing the $(34,32)$ Parity++ code (as opposed to the generic $(k+2,k)$ Parity++ code).

We begin by creating the generating matrix for the Hamming code whose message length is at least as large as the message length in the desired Parity++ code; in our case, we use the $(63,57)$ Hamming code. Let α be a primitive element of GF(2^6) such that $1+x+x^6 = 0$, then our generator polynomial is simply $g_S(x) = 1+x+x^6$ (and we construct our generator matrix using the usual polynomial coding methods). We then shorten this code to $(32,26)$ by expurgating and puncturing (i.e., deleting) the right and bottom 30 columns and rows. Now, we add a column of 1s to the end, resulting in a generator matrix, which we denote as $\mathbf{G}_S$, for a $(33,26)$ code with $d_{min} = 4$.

For the next step in the construction of the generating matrix of our $(34,32)$ Parity++ code, we add $\mathbf{G_N}$ on top of $\mathbf{G}_S$, where $\mathbf{G_N}$ is the first 6 rows of the generator matrix using the generator polynomial $g_N(x) = 1+x$, with an appended row of 0s at the end. Note that $\mathbf{G_N}$ is the generator polynomial of a simple parity-check code. By using this polynomial subcode construction, we have built a generator matrix with overall $d_{min} = 2$, with the submatrix $\mathbf{G}_S$ having $d_{min} = 4$. At this point, notice that messages that begin with 6 0s only interact with $\mathbf{G}_S$; these messages will be our special messages. Note that Conditions 3.1 and 3.3 are satisfied; however, Condition 3.2 is not satisfied. To

meet the requirement, we add a single non-linear parity-bit that is a NOR of the bits corresponding to $\mathbf{G_N}$, in our case, the first 6 bits.

The final step is to convert $\mathbf{G}_S$ to systematic form via elementary row operations. Note that these row operations preserve all 3 of the required minimum distance properties of Parity++. As a result, the special codewords (with the exception of the known prefix) are in systematic form. For example, in our $(34,32)$ Parity++ code, the first 26 bits of a special codeword are simply the 26 bits in the message (not including the leading run of 6 0s).

At the encoding stage of the process, when the message is multiplied by $\mathbf{G}$, the messages denoted as special must begin with a leading run of $\log_2(k)+1$ 0's. However, the original messages we deem to be special do not have to follow this pattern as we can simply apply a pre-mapping before the encoding step, and a post-mapping after the decoding step.

In our $(34,32)$ Parity++ code, observe that there are 2^{26} special messages. Generalizing, it is easy to see that for a $(k+2,k)$ Parity++ code, there are $2^{k-\log_2(k)-1}$ special messages.

3.3.2 Error Detection and Correction

We separate the received–possibly erroneous–vector $\mathbf{y}$ into two parts, $\bar{\mathbf{c}}$ and η, with $\bar{\mathbf{c}}$ being the first $k+1$ bits of the codeword and η the additional nonlinear redundancy bit ($\eta = 0$ for special messages and $\eta = 1$ for normal messages). There are three possible scenarios at the decoder: no (detectable) error, correctable error, or detected but uncorrectable error.

First, due to the Parity++ construction, every valid codeword has even weight. Thus, if $\bar{\mathbf{c}}$ has even weight, then the decoder concludes no error has occurred, i.e., $\bar{\mathbf{c}}$ was the original codeword. Second, if $\bar{\mathbf{c}}$ has odd weight and $\eta = 0$, the decoder attempts to correct the error. Since $\mathbf{G}_S$ is in systematic form, we can easily retrieve $\mathbf{H}_S$, its corresponding parity-check matrix. The decoder calculates the syndrome $\mathbf{s_1} = \mathbf{H}_S^T\bar{\mathbf{c}}$. If $\mathbf{s_1}$ is equal to a column in $\mathbf{H}_S$, then that corresponding bit in $\bar{\mathbf{c}}$ is flipped. Third, if $\bar{\mathbf{c}}$ has odd weight and either $\mathbf{s_1}$ does not correspond to any column in $\mathbf{H}_S$ or $\eta = 1$, then the decoder declares a DUE.

The decoding process described above guarantees that any single-bit error in a special codeword will be corrected, and any single-bit error in a normal codeword will be detected (even if the bit in

error is η).

Let's take a look at two concrete examples for the $(10, 8)$ Parity++ code. Without any premapping, a special message begins with $\log_2(3) + 1 = 4$ zeros. Let our original message be $\mathbf{m} = (00001011)$, which is encoded to $\mathbf{c} = (1011010110)$. Note that the first 4 bits of $\mathbf{c}$ is the systematic part of the special codeword. After passing through the channel, let the received vector be $\mathbf{y} = (1001010110)$, divided into $\bar{\mathbf{c}} = (1001010110)$ and $\eta = 0$. Since the weight of c is odd and $\eta = 0$, the decoder attempts to correct the error. The syndrome is equal to the 3rd column in $\mathbf{H}_S$, thus the decoder correctly flips the 3rd bit of $\bar{\mathbf{c}}$.

For the second example, let us begin with $\mathbf{m} = (11010011)$, which is encoded to (0011111101). After passing through the channel, the received vector is $\mathbf{y} = (0011011101)$. Since the weight of $\bar{\mathbf{c}}$ is odd and $\eta = 1$, the decoder declares a DUE. Note that for both normal and special codewords, if the only bit in error is η itself, then it is implicitly corrected since $\bar{\mathbf{c}}$ has even weight and will be correctly mapped back to $\mathbf{m}$ without any error detection or correction required.

3.3.3 Architecture

Figure 3.1 shows the flow of a normal read operation in a cache with any ECC protection scheme. Due to the protection mechanism, there is additional error detection/correction latency. Error detection latency is more critical than error correction as occurrence of an error is a rare event when compared to the processor cycle time and doesn't fall in the critical path. The data/instruction being read from the cache goes through the ECC error detection engine first. If there are no errors then the decoded message moves ahead. In case of an error, the received message goes through an additional correction engine to retrieve the correct message and then the message can be used in the rest of the computation flow.

When using Parity++, the flow almost remains the same. Parity++ can detect all single bit errors but has correction capability for "special messages". When a single bit flip occurs on a message, the error detection engine first detects the error and stalls the pipeline. If the non-linear bit says it is a "special message"(non-linear bit is '0'), the received message goes through the Parity++ error correction engine which outputs the corrected message. This marks the completion of the cache

access. If the non-linear bit says it is a non-special message (non-linear bit is '1'), it is checked if the cache line is clean. If so, the cache line is simply read back from the lower level cache or the memory and the cache access is completed. However, if the cache line is dirty and there are no other copies of that particular cache line, it leads to a crash or a roll back to checkpoint. Note that both Parity++ and SECDED have equal decoding latency of one cycle that is incurred during every read operation from an ECC protected cache. The encoding latency during write operation does not fall in the critical path and hence, is not considered in our analyses.

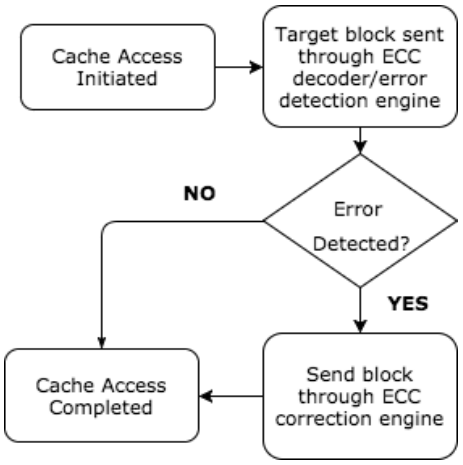


Figure 3.1: Flow of a read operation in a cache with ECC protection

Next in this chapter, we present a memory speculation scheme that helps to hide the latency incurred by the error detection engine when there are no errors.

3.3.3.1 Memory Speculation

Figure 3.2 shows the flow of a read operation when the memory speculation scheme is used. The basic idea behind this speculation scheme is to predict the original message from the encoded codeword without having to go through the decoding/error detection circuitry in order to hide the additional latency incurred by the decoding/detection mechanism. While the decoding happens, the predicted instruction/data can move forward to the next stages in the pipeline. If the predicted value is correct, then no action is required and pipeline goes ahead as usual without any additional stalls. In case an error is detected, the mis-predicted instruction or all the dependent instructions that received the mis-predicted data needs to be squashed. This prediction scheme for ECC protected

caches is similar to what was proposed in [98] for stronger error protection in on-chip memories.

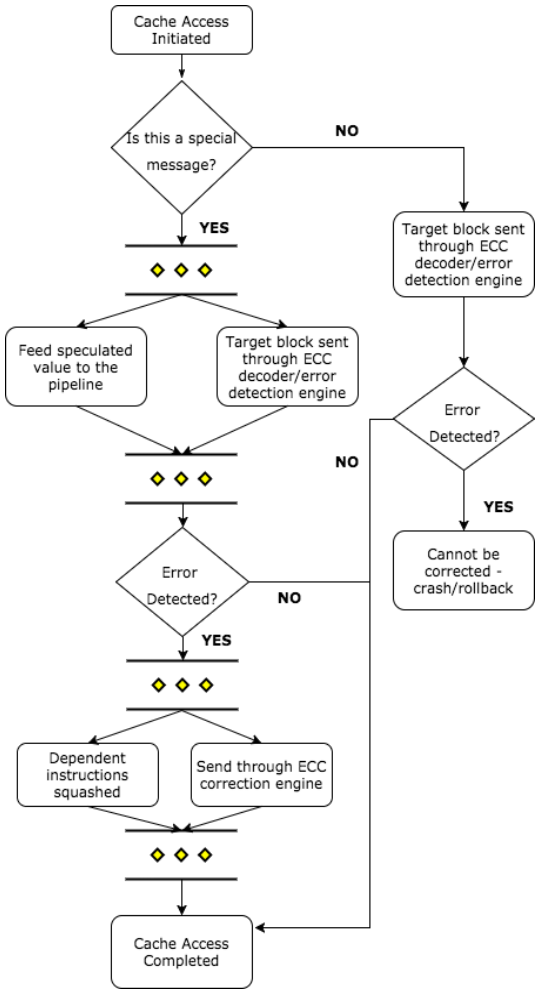


Figure 3.2: Flow of read operation in cache with memory speculation and Parity++ protection schemes

This speculation scheme is most effective when the encoded ECC codewords are systematic. When systematic, the original message can be easily retrieved by truncating the additional redundant bits that are generally added to the end of the actual message in case of no errors in the received codeword. Instead of waiting for the decoding to get done, the original message can be speculated by truncating the redundant bits. Thus, the computation moves ahead with the predicted data/instruction without any stalls while the decoding for error detection happens in parallel. A major difference between SECDED and our scheme, Parity++ is that all codewords under SECDED are systematic while only the special messages for Parity++ are systematic. As a result, for Parity++, speculation is

used only if the message is special. If not, computation is stalled for one cycle while decoding/error detection happens. Special messages can be distinguished from non-special messages using the non-linear bit.

3.3.3.2 Additional Cache Support for Speculation

Figure 3.3 depicts the additional circuitry that needs to be added to a traditional cache to support the memory speculation scheme with Parity++.

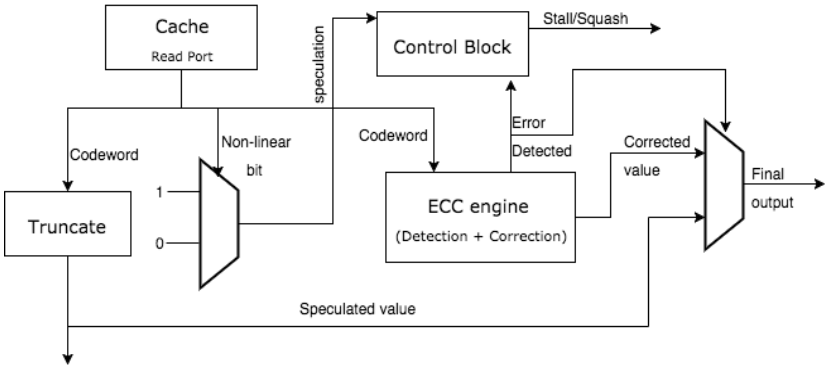


Figure 3.3: Cache architecture to implement Parity++ with memory speculation

The non linear bit is first checked. If it is a special message, then speculation is triggered and the speculated value is forwarded to the next stage. This speculated value comprises of the lower 26-bits of the received codeword to which the special prefix is separately appended. Meanwhile, the decoding and the error detection circuitry works in parallel. If an error is detected, the control module initiates a squash operation to squash all the dependant instructions that used the mis-predicted data and the ECC correction engine provides the correct output. The control module also stalls the pipeline when the non linear bit indicates that the message is not special and hence, the codeword is not systematic. Therefore, speculation cannot be used and the pipeline needs to be stalled for one cycle till the original message is decoded. The stall latency is, of course, greater than one cycle when an error is detected and the ECC correction engine needs to be triggered. This additional control module is simple and has minimal overhead in terms of area and energy.

3.3.4 Coverage and Overheads

3.3.4.1 Detection/Correction Coverage

Single-bit parity detects any single-bit error. Our Parity++ scheme keeps this single-bit error detection guarantee, and additionally provides single-bit error correction for special messages. Also, any 2-bit error on a special message in our Parity++ scheme is guaranteed detectable.

The coverage of SECDED and DECTED codes can be understood from their names. SECDED codes guarantee correction of any single bit error and detection of any double bit error; DECTED codes guarantee correction of any double bit error and detection of any triple bit error.

Table 3.2: Error Detection and Correction Coverage for Parity++ along with some widely used ECC schemes

ECC scheme	Error Bits Detected	Error Bits Corrected
Parity- Single Error Detecting (SED)	1	0
Parity++	Special Messages - 2 Non-Special Messages - 1	Special Messages - 1 Non-Special Messages - 0
SECDED	2	1
DECTED	3	2

3.3.4.2 Storage Overhead

Single-error detection requires only a single parity bit; our Pairty++ scheme adds an additional parity-bit for a total of 2. The most efficient SEC code is the Hamming code. Assuming our message length, k, is a power of 2, then the number of redundancy bits required for the (shortened) Hamming code is $\log(k)+1$. Since the Hamming code has a minimum distance of 3, we can create a SECDED code—the extended Hamming code—with the addition of a single parity bit, yielding a total of $\log(k)+2$ redundancy bits. Similarly, we can use a (shortened) extended BCH code as a DECTED code, with $2\log(k)+3$ redundancy bits.

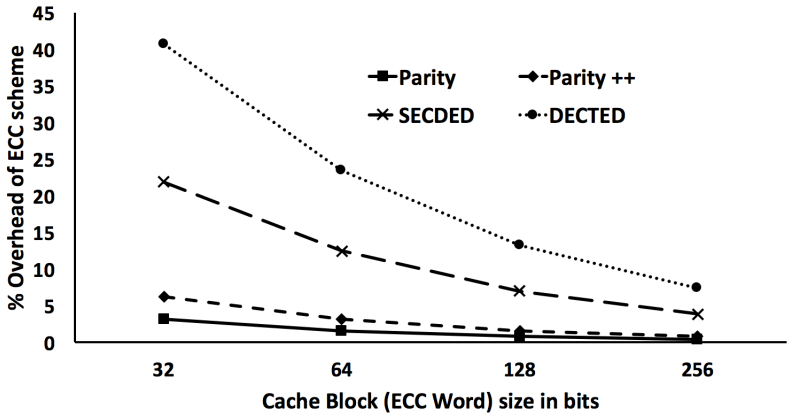


Figure 3.4: Storage overhead of different commonly used ECC schemes along with our scheme Parity++

3.3.4.3 Latency and Energy Overhead

The encoding and decoding latencies when writing to/reading from the memory are almost identical for Parity++ and SECDED. They would both require an additional one cycle for each of the two operations. Error correction in case of Parity++ requires an extra matrix multiplication. However, this latency is not critical as occurrence of errors is a rare event compared to the cycle time of the processor. With the proposed memory speculation scheme, SECDED incurs no additional decoding latency when there are no errors. For Parity++ the one cycle extra decoding latency happens only when it is a non special message (only 20-25% of messages are typically non-special).

The encoding energy overhead is almost similar for both Parity++ and SECDED. The decoding energy overheads are slightly different. For SECDED, the original message can be retrieved from the received codeword by simply truncating the additional ECC redundant bits. However, all received codewords need to be multiplied with the H-matrix to detect if any errors have occurred. For Parity++, the original message can be retrieved using truncation when it is a special messages. For the 20-25% non special messages, the non-systematic received codeword needs to be multiplied with a decoder matrix to get the original message. This decoder matrix multiplication has $\sim$4x higher energy overhead than the H-matrix multiplication of SECDED since the Parity++ decoder is larger than the SECDED H-matrix. However, for Parity++, the error detection scheme is much simpler. It is just a chain of XOR gates and hence consumes $\sim$10x lower energy than the H-matrix

multiplication of SECDED required for error detection. In short, for SECDED, every message needs to be multiplied with H-matrix for error detection even though all original messages can be retrieved through truncation of received codewords. For Parity++, all messages go through the chain of XOR gates for error detection and only the non special messages need to be multiplied with the decoder matrix to retrieve the original message. Since the error detection in Parity++ is much cheaper in terms of energy overhead than SECDED and the non special messages only constitute about 20-25% of the total messages, the overall read energy in Parity++ turns out to be much lesser than SECDED. Also, with reduced array size for caches with Parity++ due to lower storage overhead, the leakage energy is also less than that in caches with SECDED.

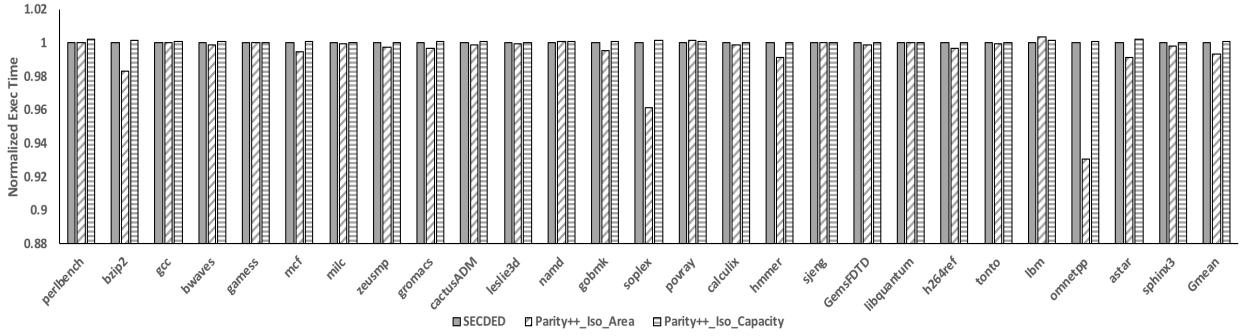


Figure 3.5: Comparing Normalized Execution Time of Processor-I with SECDED and Parity++ (with memory speculation)

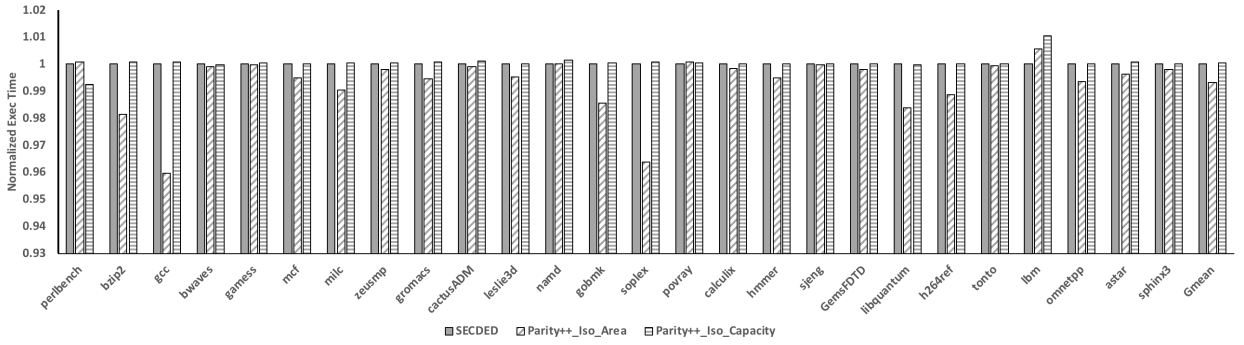


Figure 3.6: Comparing Normalized Execution Time of Processor-II with SECDED and Parity++ (with memory speculation)

3.4 Experimental Methodology

We evaluated Parity++ over applications from the SPEC 2006 benchmark suite. Two sets of core micro-architectural parameters (provided in Table 3.3) were chosen to understand the performance benefits in both a lightweight in-order(InO) processor and a larger out-of-order(OoO) core. Performance simulations were run using Gem5 [99], fast forwarding for 1 billion instructions and executing for 2 billion instructions.

The first processor is a lightweight single in-order core architecture with a 32kB L1 cache for instruction and 64kB L1 cache for data. Both the instruction and data caches are 4-way associative. The LLC is a unified 1MB L2 cache which is also 8-way associative. The second processor is a dual core out-of-order architecture. The L1 instruction and data caches have the same configuration as the previous processor. The LLC comprises of both L2 and L3 caches. The L2 is a shared 512kB SRAM based cache while the L3 is a shared 2MB cache which is 16-way associative. For both the baseline processors it is assumed that the LLCs (L2 for the InO processor and L2 and L3 for the OoO processor) have SECDED ECC protection.

The performance evaluation was done only for cases where there are no errors. Thus, latency due to error detection is taken into consideration but not error correction as correction is rare when compared to the processor cycle time and doesn't fall in the critical path. In order to compare the performance of the systems with Parity++ against the baseline cases with SECDED ECC protection, the size of the LLCs were increased by $\sim$10% due to the lower storage overhead of Parity as provided in Section 3.3.4. We call this iso-area since the additional area coming from reduction in redundancy is used to increase the total capacity of the SRAM. The iso-area evaluation was done for both with and without memory speculation. The analysis was also done for the iso-capacity where the memory capacity of the systems with Parity++ and SECDED remain same and their performances are measured. As mentioned before, SECDED allows speculation in all cases and thus, incurs no additional read latency due to error detection when there is no error. But for Parity++, only the special messages are systematic and thus, for all non-special messages, there is an additional one cycle read latency due to the error detection circuitry. This additional latency for non-special messages was also taken into consideration for our simulations.

Table 3.3: Core Micro-architectural Parameters

	Processor-1	**Processor-2**
Cores	1, InO (@ 2GHz)	2, OoO (@ 2GHz)
L1 Cache per core	32KB I$ 64KB D$ 4-way	32KB I$ 64kB D$ 4-way
L2 Cache	1MB (unified) 8-way	512KB (shared, unified) 8-way
L3 Cache	-	2MB (shared) 16-way
Cache Line Size	64B	64B
Memory Configuration	4GB of 2133MHz DDR3	8GB of 2133MHz DDR3
Nominal Voltage	1V	1V

3.5 Results and Discussion

In this section we discuss the performance results obtained from the Gem5 simulations (as mentioned in Section 3.4). Figures 3.5 and 3.6 show the comparative results for the two different sets of core micro-architectures across a variety of benchmarks from the SPEC2006 suite when using memory speculation. In both the evaluations, performance of the system with Parity++ was compared against that with SECDED. The evaluation was further split into iso-area and iso-capacity as explained in Section 3.4.

For both the core configurations, the observations for the iso-area case are almost similar. With memory speculation it is seen that with additional memory capacity for iso-area, the system with Parity++ has upto $\sim$4% better performance (lower execution time) than the one with SECDED. This improvement in performance happens in spite of the additional one cycle latency incurred on non special messages in the case of Parity++. The applications showing higher performance benefits are mostly memory intensive. Hence, additional cache capacity with Parity++ reduces overall miss rate to an extent such that the slight increase in average LLC hit time gets offset. For most of these applications, this performance gap widens as the LLC size increases for Processor-II. The applications showing roughly similar performances on both the systems are the ones which already have a considerably lower LLC miss rate. As a result, increase in LLC capacity due to

Parity++ doesn't lead to a significant improvement in performance. The same evaluation was also done for the case where there is no memory speculation, i.e., both Parity++ and SECDED protected caches have additional hit latency of one cycle for all read operations. The results show that with the exact same hit latency, Parity++ has upto 7% lower execution time than SECDED due to additional memory capacity.

A more significant result is the iso-capacity case with memory speculation. It is seen that even with additional one cycle latency for non special messages in Parity++, the performance of the system with Parity++ is at par with that of SECDED. This means that by using our lightweight error correction scheme, we manage to save about 5-9% last level cache area (excluding decoder and peripheral circuit area) with negligible hit in performance. Since the LLCs constitute more than 30% of the processor chip area, the cache area savings translate to a considerable amount of reduction in the chip size. This additional area benefit can either be utilized to make an overall smaller sized chip or it can be used to pack in more compute tiles to increase the overall performance of the system.

The iso-capacity results also imply that Parity++ can be used in SRAM based scratchpad memories used in embedded systems at the edge of the Internet-of-Things (IoT) where hardware design is driven by the need for low area, cost and energy consumption. Since Parity++ helps in reducing area (in turn reducing SRAM leakage energy) and also has lower error detection energy, it provides a better protection mechanism than SECDED in such devices.

3.6 Conclusion

In this work, we present a novel lightweight error protection scheme, Parity++, for last level caches based on unequal message protection. From our analysis, we find that about 80% of messages/words have same prefix bits (leading 0's) and we denote these as special messages. For a 64 bit word, Parity++ uses only 2 additional redundant bits and provides SECDED protection for these special messages while providing only SED for the non-special messages. In iso-area evaluations, up to about 4% performance benefit can be obtained, while iso-capacity evaluations showed almost negligible ($<$0.2% in all but one case) performance degradation with $\sim$9% lower storage overhead than a traditional SECDED scheme which translates to about 5% cache area savings.

CHAPTER 4

Conclusion

This chapter reviews the key contributions of this thesis and outlines directions for future work.

4.1 Overview of Contributions

A series of techniques were proposed to cope with hardware variability and errors in on-chip memories. With increase in size of on-chip memories and decrease in physical dimensions of cells, memory reliability is becoming a growing concern. The challenge with these memories is that the fault tolerance techniques need to effective but with minimal overhead.

4.1.1 FaultLink and SDELC

FaultLink and SDELC provided a holistic virtualization-free fault tolerance methodology to deal with hard and soft faults in software managed embedded memories in IoT devices. Hardware design in most of these IoT devices is driven by the need for low cost and low power. One way to reduce power consumption is to lower the supply voltage. But as the VDD is lowered, some of the weak SRAM memory cells begin to fail. Hence, low cost protection against hard faults in memory is required if these devices have to be run at low voltage. FaultLink does exactly that with almost no hardware overhead. In software managed memories, data placement in memory is orchestrated by the software. Thus, application programmers, with the help of tools like compiler and linker, explicitly partition data into physical memory regions that are distinct in the address space. FaultLink utilizes exactly that property of software managed memories and makes loading application in faulty memory plausible. It takes a pre-compiled binary of an application and links it to the memory in such a way that the application would not access the bad locations. Thus, the

application is compiled once but the final linked binary image is unique for every chip. SDELC, on the other hand, helps to recover from unpredictable single bit flips in the memory that occur during runtime. It helps to localize the error to a smaller chunk in a 32/64-bit message and then tries to heuristically recover from it using software defined policies that leverage on the available side information about memory contents to choose the most likely candidate codeword. Overall, FaultLink and SDELC together opportunistically copes with memory errors in low-cost IoT devices and helps in improving the longevity of these devices.

4.1.2 Parity++

Parity++, like SDELC, is another lightweight error recovery scheme. But Parity++ tackles the problem of miscorrections that might occur with SDELC during the heuristic recovery. Instead of trying to recover from errors heuristically, Parity++ preferentially provides stronger protection to certain "special messages". While it provides stronger protection than a basic Single Error Detection parity, it has lower overhead than a full single error correcting Hamming code. With just two additional bits of redundancy per message, this code is ideal for last level caches. We also propose a memory speculation scheme that can be used to further hide the decoding latency that comes with using any error correcting code. Parity++ can be extended to embedded scratchpad memories as it has much lower area overhead and can be opportunistically used to reduce the chip area.

4.2 Directions for Future Work

There are many future possibilities for Lightweight Fault Tolerance in Memory Systems. Firstly a FaultLink-compatible remote software update mechanism for IoT devices in the field need to be designed and new failure modes with SDELC need to be supported. Also for FaultLink, the stack and heap in applications were not split. A split stack and heap would lead to smaller program sections, which, in turn, would allow the user to tolerate more faults and thus run at ever lower supply voltages. Parity++ can be extended to server class systems with large sized last level caches where the chip area savings would be considerable and can be utilized to increase the number of

cores or the size of the memory to improve overall system performance. Alongside these extensions to the techniques proposed in the thesis, a possible direction for future work would be fault tolerance in emerging memory devices that have been suggested as potential replacements for SRAM based on-chip embedded memories and lasrger sized last level caches. Reliability is the current biggest concern facing these non-volatile memories (NVM) that can potentially eclipse the density and energy benefits these technologies promise. A lot of work needs to be done in this area before these emerging random access NVMs such as STT-MRAM, MeRAM, ReRAM, etc. start replacing on-chip or off-chip memories. The stochastic bit failures in NVMs is similar to the radiation induced soft errors in DRAM and SRAM and occur without any warning and hence both these techniques can be extended to deal with these random bit errors. For example, a stronger version of Parity++, which can provide double error correcting to certain "special messages" with just one extra bit as compared to the commonly used SECDED code can be used in these NVMs with high bit error rate.

REFERENCES

[1] M. Gottscho, I. Alam, C. Schoeny, L. Dolecek, and P. Gupta, "Low-cost memory fault tolerance for iot devices," *ACM Trans. Embed. Comput. Syst.*, vol. 16, pp. 128:1–128:25, Sept. 2017.

[2] C. Schoeny, F. Sala, M. Gottscho, I. Alam, P. Gupta, and L. Dolecek, "Context-aware resiliency: Unequal message protection for random-access memories," in *2017 IEEE Information Theory Workshop (ITW)*, pp. 166–170, Nov 2017.

[3] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," *Communications of the ACM*, vol. 54, pp. 100–107, Feb. 2011.

[4] S. Mittal, "A Survey of Architectural Techniques for Managing Process Variation," *ACM Computing Surveys*, vol. 48, no. 4, 2016.

[5] 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.

[6] B. Schroeder and G. A. Gibson, "Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?," in *Proc. 5th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2007.

[7] 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," June 2015.

[8] 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," June 2014.

[9] V. Narayanan and Y. Xie, "Reliability Concerns in Embedded System Designs," *Computer*, vol. 39, no. 1, pp. 118–120, 2006.

[10] NSF and SRC, "Report to the National Science Foundation on the Workshop for Energy Efficient Computing," tech. rep., 2015.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] S.-L. Lu, Q. Cai, and P. Stolt, "Memory Resiliency," *Intel Technology Journal*, vol. 17, no. 1, 2013.

[16] P. R. Panda, N. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. 1999.

[17] 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.

[18] 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 Systems (TVLSI)*, vol. 12, no. 2, pp. 167–184, 2004.

[19] 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 Systems (TVLSI)*, vol. 21, no. 6, pp. 1000–1012, 2013.

[20] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software-Defined Error-Correcting Codes," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2016.

[21] "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (Version 1.2)," 1995.

[22] 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.

[23] 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.

[24] T. J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," tech. rep., IBM Microelectronics Division, 1997.

[25] 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.

[26] J. K. Wolf, "On an Extended Class of Error-Locating Codes," *Information and Control*, vol. 8, no. 2, pp. 163–169, 1965.

[27] 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*, 1993.

[28] 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," *De Gruyter Information Technology (it)*, vol. 57, no. 3, 2015.

[29] L. Lai, *Cross-Layer Approaches for Monitoring, Margining and Mitigation of Circuit Variability*. Ph.d. dissertation, University of California, Los Angeles (UCLA), 2015.

[30] 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," tech. rep., University of California, Los Angeles (UCLA), 2014.

[31] S. Lamikhov-Center, "ELFIO: C++ Library for Reading and Generating ELF Files," 2016.

[32] M. F. Schilling, "The Surprising Predictability of Long Runs," *Mathematics Magazine*, vol. 85, no. 2, pp. 141–149, 2012.

[33] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.0," 2014.

[34] 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.

[35] 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.

[36] J. Yang, Y. Zhang, and R. Gupta, "Frequent Value Compression in Data Caches," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pp. 258–265, 2000.

[37] A. Alameldeen and D. Wood, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches," tech. rep., University of Wisconsin, Madison, 2004.

[38] 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.

[39] 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)*, 2011.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A Multiplatform Benchmark Suite for Approximate Computing," *IEEE Design and Test*, vol. 34, no. 2, pp. 60–68, 2017.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] M. Mutyam and V. Narayanan, "Working with Process Variation Aware Caches," in *Design, Automation, and Test in Europe (DATE)*, 2007.

[51] 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.

[52] A. Ansari, S. Gupta, S. Feng, and S. Mahlke, "ZerehCache: Armoring Cache Architectures in High Defect Density Technologies," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2009.

[53] A. Ansari, S. Feng, S. Gupta, and S. Mahlke, "Archipelago: A Polymorphic Cache Design for Enabling Robust Near-Threshold Operation," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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, 2015.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] R. van Rein, "BadRAM: Linux Kernel Support for Broken RAM Modules," 2016.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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)*, pp. 127–132, 2010.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt, "Cache scrubbing in microprocessors: myth or necessity?," in *10th IEEE Pacific Rim International Symposium on Dependable Computing, 2004. Proceedings.*, pp. 37–42, March 2004.

[78] J. Yan and W. Zhang, "Evaluating instruction cache vulnerability to transient errors," in *Proceedings of the 2006 Workshop on MEmory Performance: DEaling with Applications, Systems and Architectures*, MEDEA '06, (New York, NY, USA), pp. 21–28, ACM, 2006.

[79] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes*. Elsevier, 1977.

[80] S. Lin and D. J. Costello, *Error control coding*, vol. 2. Prentice Hall Englewood Cliffs, 2004.

[81] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of multi-bit soft error events in advanced srams," in *IEEE International Electron Devices Meeting 2003*, pp. 21.4.1–21.4.4, Dec 2003.

[82] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Transactions on Device and Materials Reliability*, vol. 5, pp. 397–404, Sept 2005.

[83] "Qualcomm Centriq 2400 Processor."

[84] J. Huynh, "White Paper: The AMD Athlon MP Processor with 512KB L2 Cache," tech. rep., May 2003.

[85] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, pp. 66–76, March 2003.

[86] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, pp. 5–25, Jan 2002.

[87] M. Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, pp. 395–401, July 1970.

[88] D. H. Yoon and M. Erez, "Memory mapped ecc: Low-cost error protection for last level caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 116–127, ACM, 2009.

[89] S. Kim and A. K. Somani, "Area efficient architectures for information integrity in cache memories," *SIGARCH Comput. Archit. News*, vol. 27, pp. 246–255, May 1999.

[90] N. N. Sadler and D. J. Sorin, "Choosing an error protection scheme for a microprocessor's l1 data cache," in *2006 International Conference on Computer Design*, pp. 499–505, Oct 2006.

[91] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software-defined ECC: Heuristic recovery from uncorrectable memory errors," tech. rep., University of California, Los Angeles, Oct. 2017.

[92] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pp. 258–265, 2000.

[93] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," 2004.

[94] 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 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 377–388, ACM, 2012.

[95] 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 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 329–340, IEEE Press, 2016.

[96] C. Schoeny, F. Sala, M. Gottscho, I. Alam, P. Gupta, and L. Dolecek, "Context-aware resiliency: Unequal message protection for random-access memories," in *Proc. IEEE Information Theory Workshop*, (Kaohsiung, Taiwan), pp. 166–170, Nov. 2017.

[97] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[98] H. Duwe, X. Jian, and R. Kumar, "Correction prediction: Reducing error correction latency for on-chip memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 463–475, Feb 2015.

[99] 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," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.