Low-Cost Memory Fault Tolerance for IoT Devices

MARK GOTTSCHO, University of California, Los Angeles
IRINA ALAM, University of California, Los Angeles
CLAYTON SCHOENY, University of California, Los Angeles
LARA DOLECEK, University of California, Los Angeles
PUNEET GUPTA, University of California, Los Angeles

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.

CCS Concepts: • **Computer systems organization → Embedded hardware**; **Embedded software**; **Reliability**; **Processors and memory architectures**; • **Hardware → Process variations**; *Transient errors and upsets*; Aging of circuits and systems; • **Mathematics of computing →** *Coding theory*;

Additional Key Words and Phrases: scratchpad memory, fault tolerance, ECC, IoT, defects, soft errors, approximate computing

ACM Reference format:
Mark Gottscho, Irina Alam, Clayton Schoeny, Lara Dolecek, and Puneet Gupta. 2017. Low-Cost Memory Fault Tolerance for IoT Devices. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (July 2017), 25 pages.
DOI: X.XXX/XXX_X

This article will be presented in the ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) in Seoul, South Korea, October 15-20, 2017, and will appear as part of an ESWEEK special issue in ACM TECS.
This work was supported by the 2016 USA Qualcomm Innovation Fellowship, the 2016 UCLA Dissertation Year Fellowship, and NSF grant numbers CCF-1029030 and CCF-1150212.
Authors' emails: {mgottscho,irina1,cschoeny}@ucla.edu, {dolecek,puneet}@ee.ucla.edu.
Authors' addresses: M. Gottscho, I. Alam, C. Schoeny, L. Dolecek, and P. Gupta, Electrical Engineering Department, University of California at Los Angeles (UCLA), 420 Westwood Plaza, Los Angeles, CA 90095, USA.
M. Gottscho (current address), Google, 1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA.

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 [23]. Memories consume significant chip area and are particularly susceptible to parameter variations and defects resulting from the manufacturing process [32]. 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) [41] – due to their 40% lower energy as well as latency and area benefits compared to hardware-managed caches [10].

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 [25]. However, this causes cell hard fault rates to rise exponentially [57] 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 [51] to deal with manufacturing defects and employ large voltage guardbands [21] to ensure reliable operation. Unfortunately, large guardbands limit the energy proportionality of memory, thus reducing battery life for duty-cycled embedded systems [58], 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. **Our contributions** 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 paper is organized as follows. Background material that is necessary to understand our contributions is presented in Sec. 2. We then describe the high-level ideas of FaultLink and SDELC to achieve low-cost embedded fault-tolerant memory in Sec. 3. FaultLink and SDELC are each described in greater detail in Secs. 4 and 5, respectively. Both FaultLink and SDELC are evaluated in Sec. 6. We provide an overview of related work in Sec. 7 before discussing other considerations and opportunities for future work in Sec. 8. We conclude the paper in Sec. 9.

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.

2.1 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 [10]. In energy and cost-conscious embedded systems, SPMs are increasingly being used for this reason and because they provide more predictable performance. In this paper, FaultLink is used to improve the reliability/min-VDD of SPMs/software-managed main memory.

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 [1]. ELF files contain a header that specifies the ISA, ABI, a list of program sections and memory segments, and various other metadata.

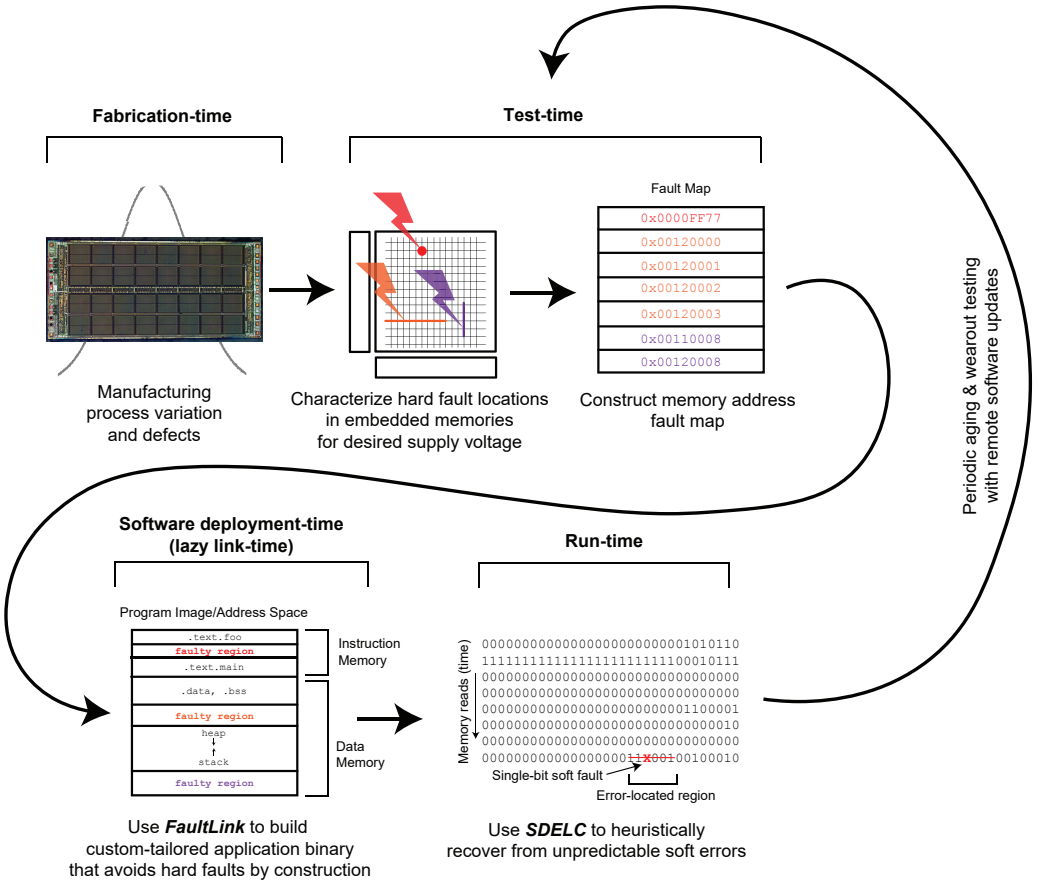


Fig. 1. Our high-level approach to tolerating both hard (*FaultLink*) and soft (*SDELC*) faults in on-chip scratchpad memories.

- 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.

2.3 Tolerating SRAM Faults

There are several types of SRAM faults. In this paper, 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 [15]. 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 [13]. Soft faults, if detected and corrected by an *error-correcting code* (ECC), are harmless to the system. In this paper, 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.

2.4 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 [14] 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.

2.5 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 [63] and studied how to create them using the tensor product of the parity-check matrices of an error-detecting and an error-correcting code [62]. ELC has been adapted to byte-addressable memory systems [17] 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.

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. 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.

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.

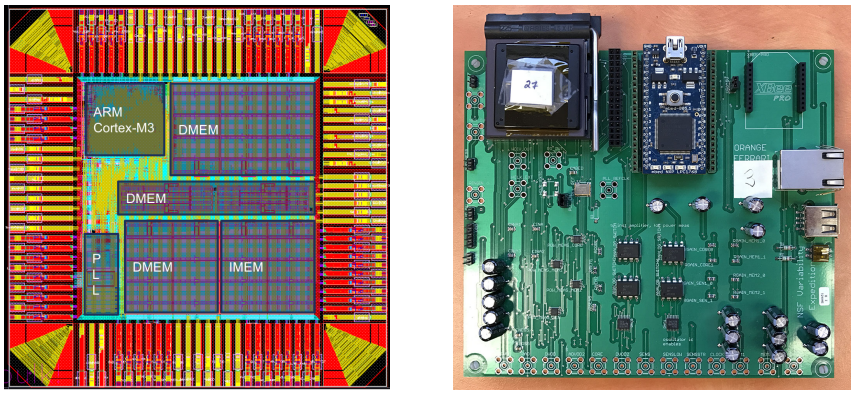
At software deployment time – i.e., when the application is actually programmed onto a particular device – FaultLink customizes the application binary image to work correctly on that particular chip given the fault map as an input. FaultLink does this by linking the program to guarantee that no hard-faulty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

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.

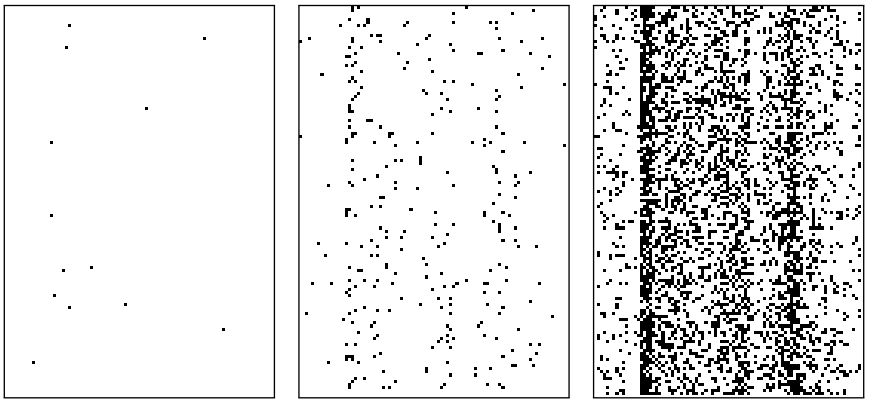
SDELC is a novel solution that lies in between these regimes. A key component is the new class of *Ultra-Lightweight Error-Localizing Codes* (UL-ELCs). UL-ELCs have lower storage overheads than Hamming codes: they can detect and then *localize* any single-bit error to a chunk of a memory codeword. We 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. 5.



(a) Chip floorplan (b) Board

Fig. 2. Test chip and board used to collect hard fault maps for FaultLink.



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

Fig. 3. Measured voltage-induced hard fault maps of the 176 KB data memory for one test chip. Black pixels represent faulty byte locations.

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.

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 [3, 26, 59]; the chip floorplan and test board are shown in Fig. 2. The locations of voltage-induced SRAM hard faults in the data memory for one chip are shown in Fig. 3 as black dots. Its byte-level fault address map appears as follows:

```
0x200057D6
```

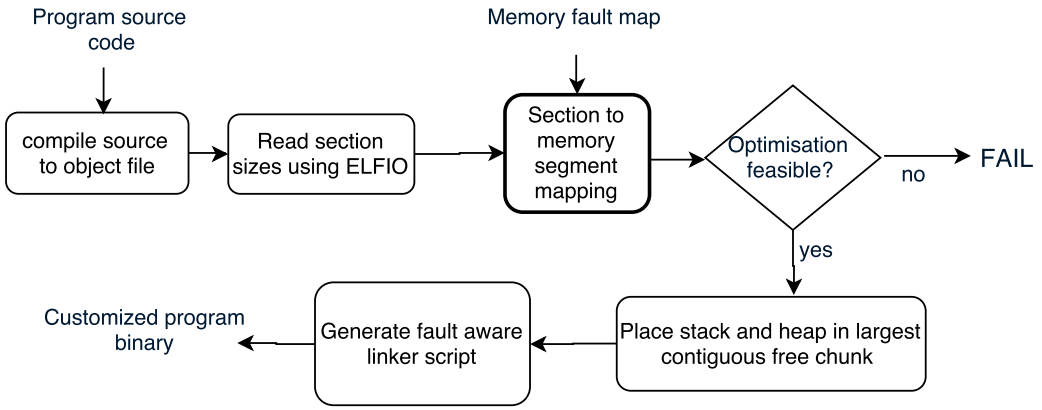


Fig. 4. FaultLink procedure: given program source code and a memory fault map, produce a per-chip custom binary executable that will work in presence of known hard fault locations in the SPMs.

```
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.

4.2 Toolchain

FaultLink utilizes the standard GNU tools for C/C++ without modification. The overall procedure is depicted in Fig. 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 [27] 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.

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.

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. 5, with the program sections on the top and fault-free memory regions on the bottom.

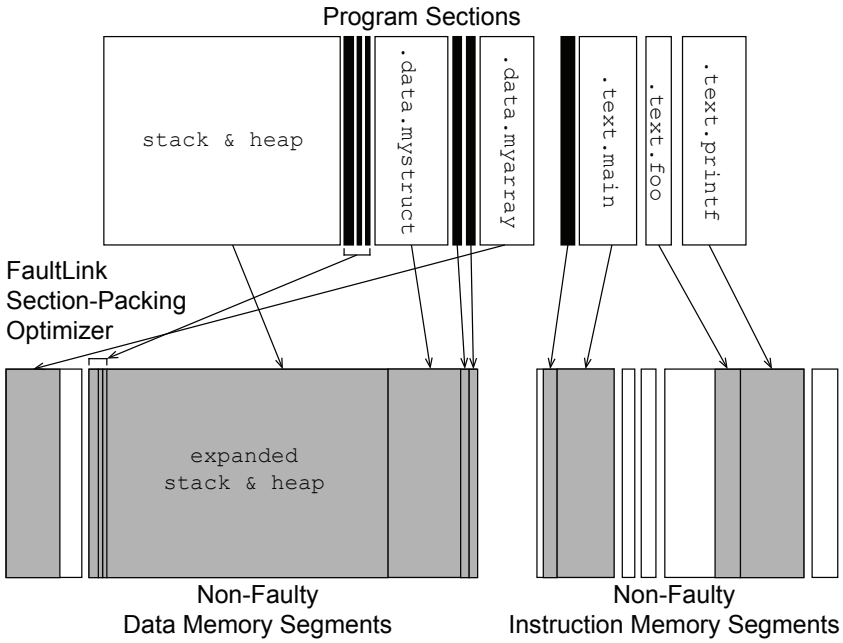


Fig. 5. FaultLink attempts to pack contiguous program sections into contiguous disjoint segments of non-faulty memory. Gray memory segments are occupied by mapped sections, while white segment areas are free space. The depicted gaps between some of the gray/white boxes indicate faulty memory regions that are not available for section-packing.

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 optimization will be feasible only if 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.

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 [50]. Let L_k be a random variable representing the length of the largest run of heads in k independent flips of a biased coin (with p as the probability of heads). The following equation is an approximation for the limiting behavior of L_k, i.e., the probability that longest run of heads is less than x and assuming $k(1-p) \gg 1$ [50]:

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

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. 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:

$$P\left(L_{\texttt{size}/4} < \frac{m_{\max}}{4}\right) \approx e^{-s^{\left(\frac{m_{\max}}{4} - \log_{s^{-1}}\left(\frac{\texttt{size}}{4}(1-s)\right)\right)}}. \tag{2}$$

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

5 SDELC

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

5.1 Architecture

The SDELC architecture is illustrated in Fig. 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

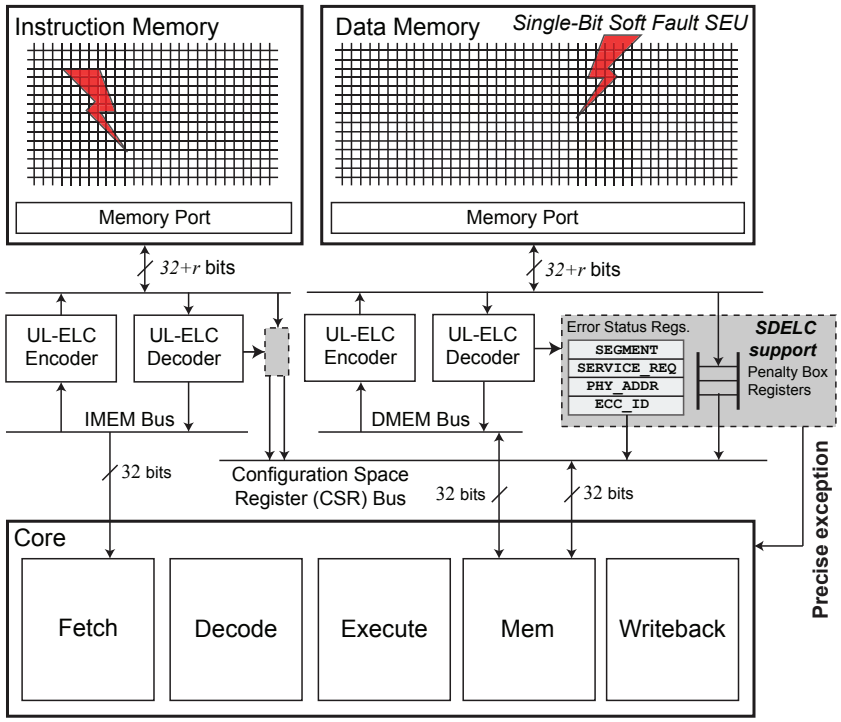


Fig. 6. Architectural support for SDELC on an microcontroller-class embedded system. Hard faults that would be managed by FaultLink are not shown.

scrub the error, update the register file appropriately, and manually advance pc before returning from the trap handler.

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 segments 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:

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

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 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.

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 [60], 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 paper.

Our UL-ELC construction for instruction memory has seven chunks that align to the finest-grain boundaries of the different fields in the RISC-V codecs. These codecs, the chunk assignments, and the complete parity-check matrix $\mathbf{H}$ are shown in Table 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

Table 1. Proposed 7-Chunk UL-ELC Construction with $r = 3$ for Instruction Memory (RV64G ISA v2.0 [60])

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-Check H	00000	00	11111	00000	111	11111	1111111	1	0	0
	00000	11	00000	11111	000	11111	1111111	0	1	0
	11111	00	00000	11111	111	00000	1111111	0	0	1

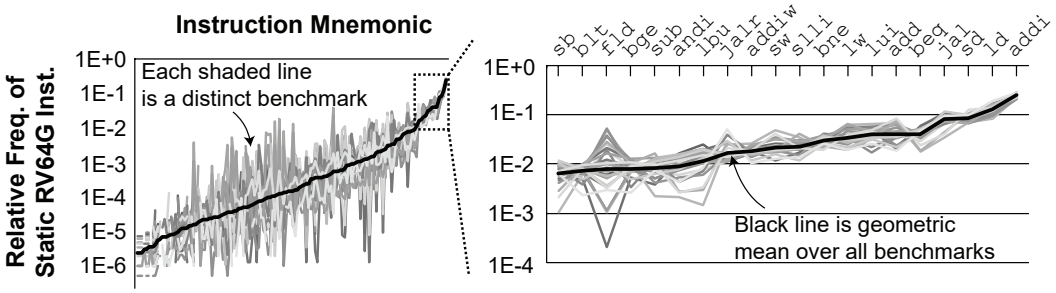


Fig. 7. The relative frequencies of static instructions roughly follow power law distributions. Results shown are for RISC-V with 20 SPEC CPU2006 benchmarks; we observed similar trends for MIPS and Alpha, as well as dynamic instructions.

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 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. 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 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.

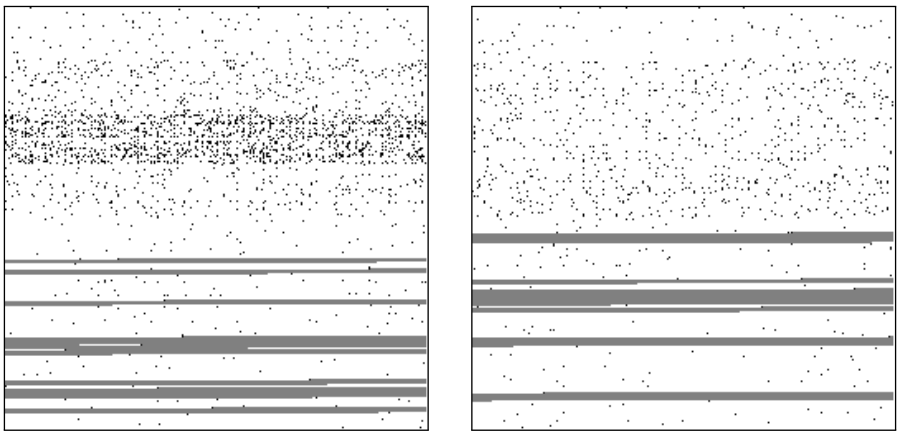
5.4 Recovering SEUs in Data Memory

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 [54].

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* [30] that has been used in numerous works for cache and memory compression [5, 42, 66]. The Hamming distance is a good measure of data correlation, as shown later in Fig. 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 [11, 12, 19, 31, 47, 67].



(a) Chip 1 (b) Chip 2

Fig. 8. Result from applying FaultLink to the sha benchmark for two real test chips' 64 KB instruction memory at 650 mV.

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.

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.

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 [22] 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 [24] 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. 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. 8. This could be caused either by the design of the chip's power grid, which might have induced a voltage imbalance

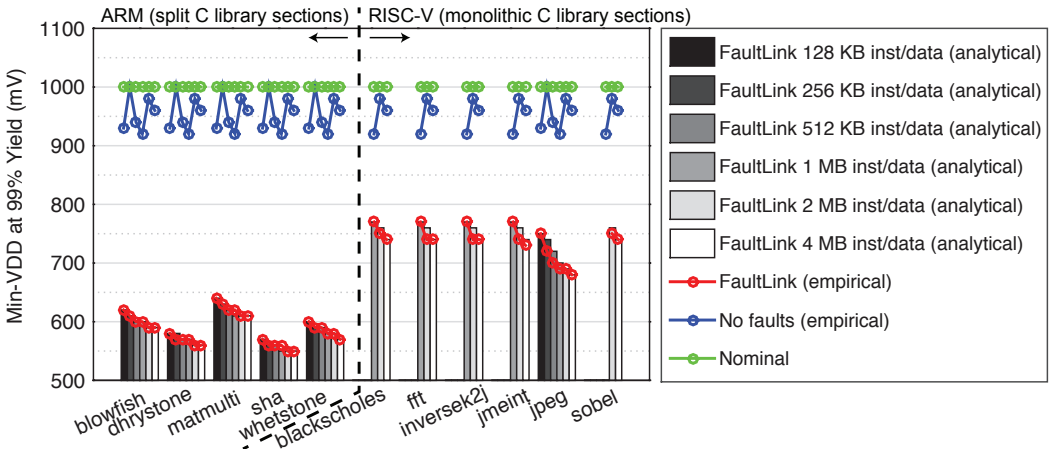


Fig. 9. Achievable min-VDD for FaultLink at 99% yield. Bars represent the analytical lower bound from Eqn. 2 and circles represent our actual results using Monte Carlo simulation for 100 synthetic fault maps.

between the two banks, or by within-die/within-wafer process variation. Chip 1 (Fig. 8a) also appears to have a cluster of weak rows in the first instruction bank. Because FaultLink chooses a solution with the sections packed into as few segments as possible, we find that the mapping for both chips prefers to use the second bank, which tends to have larger segments.

We achieved an average min-VDD of 700 mV for the real test chips. This is a reduction of 125 mV compared to the average non-faulty min-VDD of 825 mV, and 300 mV lower than the official technology specification of 1 V. FaultLink did not require more than 14 seconds on our machine to optimally section-pack any program for any chip at any voltage.

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 [67] 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 [60] and privileged specification v1.7 using the official tools. This is because unlike the standard C library for our ARM toolchain, the library included with the RISC-V toolchain has a monolithic `.text` section. This allows us to consider the impact of the library sections on min-VDD.

The expected min-VDD for 99% chip yield across 100 synthetic chip instances for seven memory capacities is shown in Fig. 9. The vertical bars represent our analytical estimates calculated using Eqn. 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

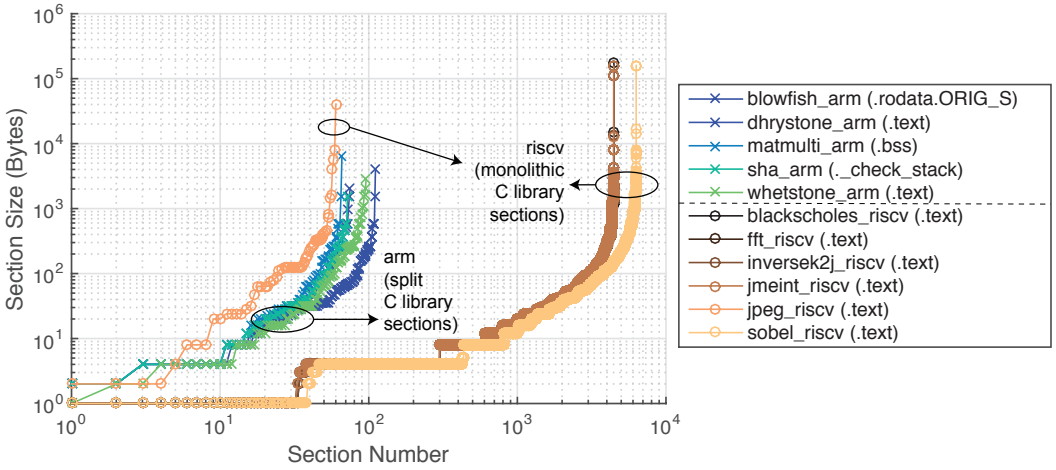


Fig. 10. Distribution of program section sizes. Packing the largest section into a non-faulty contiguous memory segment is the most difficult constraint for FaultLink to satisfy and limits min-VDD.

"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 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. 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. 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.

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

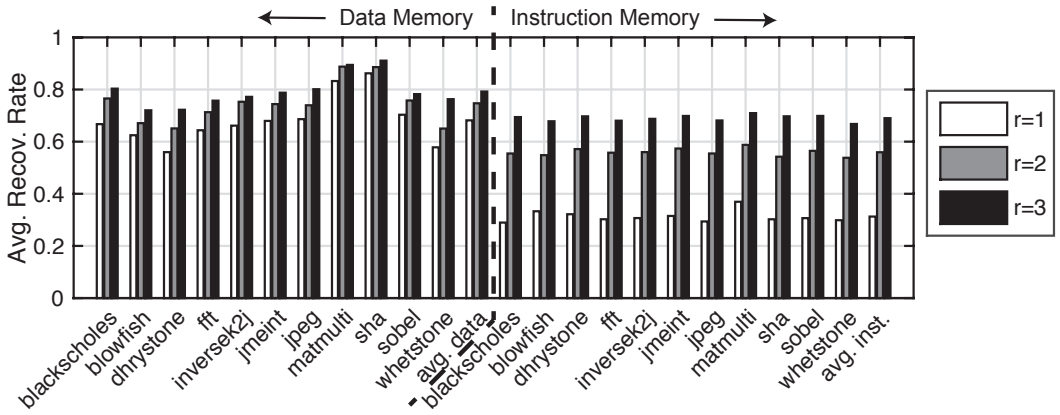


Fig. 11. Average rate of recovery using SDELC from single-bit soft faults in data and instruction memory. Benchmarks have already been protected against known hard fault locations using FaultLink. r is the number of parity bits in our UL-ELC construction.

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

6.2.1 Overall Results. The overall SDELC results are presented in Fig. 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.

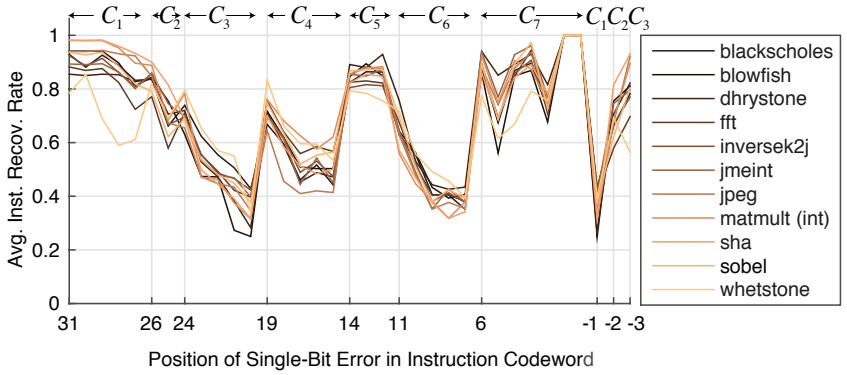


Fig. 12. Sensitivity of SDELC instruction recovery to the actual position of the single-bit fault with the $r = 3$ UL-ELC construction.

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. 12. Error positions -1, -2, and -3 correspond to the three parity bits in our UL-ELC construction from Table 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 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.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 [30] 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.

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.

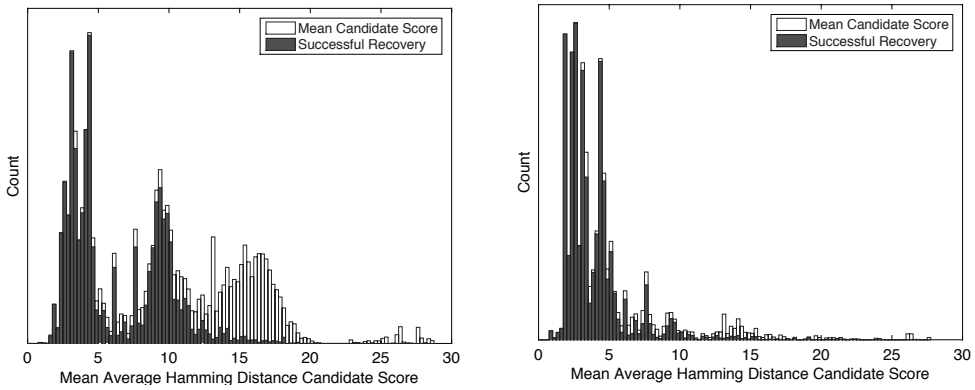


Fig. 13. Sensitivity of SDELC data recovery to the mean candidate Hamming distance score for two benchmarks and $r = 1$ parity code.

7.1 Fault-Tolerant Caches

There is an abundance of prior work on fault-tolerant and/or low-voltage caches. Examples include PADded Cache [52], Gated-VDD [43], Process-Tolerant Cache [2], Variation-Aware Caches [40], Bit Fix/Word Disable [61], ZerehCache [8], Archipelago [7], FFT-Cache [9], VS-ECC [6], Correctable Parity Protected Cache (CPPC) [35], FLAIR [44], Macho [34], DPCS [18], DARCA [36], and others (see related surveys by Mittal [37, 38]). 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 [18] achieves a similar min-VDD to FaultLink of around 600 mV, while FLAIR [44] 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 [35] 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).

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 [4] 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 [11] 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 [12, 19, 31, 55] 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 [46], re-compute corrupted data upon

detection [49], build radiation-tolerant SPMs using hybrid non-volatile memory [39] and duplicate data storage to guard against soft errors [28]; 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. [16] 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. [56] 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. [49] 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.

8 DISCUSSION

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

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.

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 [33, 45, 48, 53].

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.

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 [29], 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 [67]. 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.

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 [65]. For FaultLink with data memory, one could use smaller split stacks [64] 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.

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.

ACKNOWLEDGMENTS

The authors thank the anonymous CASES'17 program committee and reviewers for their detailed and constructive feedback. This work was supported by the 2016 USA Qualcomm Innovation Fellowship, the 2016 UCLA Dissertation Year Fellowship, and USA National Science Foundation grants CCF-1029030 (Variability Expedition) and CCF-1150212 (CAREER). The authors thank Dr. Fred Sala and Saptadeep Pal at UCLA for helpful discussions and Dr. Greg Wright from Qualcomm Research for his feedback and guidance of this work.

REFERENCES

[1] 1995. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (Version 1.2). (1995).

[2] Amit Agarwal, Bipul C. Paul, Hamid Mahmoodi, Animesh Datta, and Kaushik Roy. 2005. A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13, 1 (2005), 27–38.

[3] Yuvraj Agarwal, Alex Bishop, Tuck-Boon Chan, Matt Fotjik, Puneet Gupta, Andrew B. Kahng, Liangzhen Lai, Paul Martin, Mani Srivastava, Dennis Sylvester, Lucas Wanner, and Bing Zhang. 2014. *RedCooper: Hardware Sensor Enabled Variability Software Testbed for Lifetime Energy Constrained Application.* Technical Report. University of California, Los Angeles (UCLA).

[4] F. J. Aichelmann. 1984. Fault-Tolerant Design Techniques for Semiconductor Memory Applications. *IBM Journal of Research and Development* 28, 2 (1984), 177–183.

[5] Alaa Alameldeen and David Wood. 2004. *Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches.* Technical Report. University of Wisconsin, Madison.

[6] Alaa R. Alameldeen, Ilya Wagner, Zeshan Chishti, Wei Wu, Chris Wilkerson, and Shih-Lien Lu. 2011. Energy-Efficient Cache Design Using Variable-Strength Error-Correcting Codes. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA).*

[7] Amin Ansari, Shuguang Feng, Shantanu Gupta, and Scott Mahlke. 2011. Archipelago: A Polymorphic Cache Design for Enabling Robust Near-Threshold Operation. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[8] Amin Ansari, Shantanu Gupta, Shuguang Feng, and Scott Mahlke. 2009. ZerehCache: Armoring Cache Architectures in High Defect Density Technologies. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO).*

[9] Abbas BanaiyanMofrad, Houman Homayoun, and Nikil Dutt. 2011. 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).*

[10] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. 2002. 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).*

[11] Luis Angel D. Bathen and Nikil D. Dutt. 2011. E-RoC: Embedded RAIDs-on-Chip for Low Power Distributed Dynamically Managed Reliable Memories. In *Design, Automation, and Test in Europe (DATE).*

[12] Luis Angel D. Bathen, Nikil D. Dutt, Alex Nicolau, and Puneet Gupta. 2012. VaMV: Variability-Aware Memory Virtualization. In *Design, Automation, and Test in Europe (DATE).*

[13] Robert C. Baumann. 2005. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. *IEEE Transactions on Device and Materials Reliability* 5, 3 (2005), 305–316.

[14] Timothy J. Dell. 1997. *A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory.* Technical Report. IBM Microelectronics Division.

[15] Nikil Dutt, Puneet Gupta, Alex Nicolau, Abbas BanaiyanMofrad, Mark Gottscho, and Majid Shoushtari. 2014. Multi-Layer Memory Resiliency. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC).*

[16] Hamed Farbeh, Mahdi Fazeli, Faramarz Khosravi, and Seyed Ghassem Miremadi. 2012. Memory Mapped SPM: Protecting Instruction Scratchpad Memory in Embedded Systems against Soft Errors. In *Proceedings of the European Dependable Computing Conference (EDCC).*

[17] Eiji Fujiwara and Masato Kitakami. 1993. A class of Error Locating Codes for Byte-Organized Memory Systems. In *Proceedings of the International Symposium on Fault-Tolerant Computing.*

[18] Mark Gottscho, Abbas BanaiyanMofrad, Nikil Dutt, Alex Nicolau, and Puneet Gupta. 2015. DPCS: Dynamic Power/Capacity Scaling for SRAM Caches in the Nanoscale Era. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 3 (2015), 26.

[19] Mark Gottscho, Luis A. D. Bathen, Nikil Dutt, Alex Nicolau, and Puneet Gupta. 2015. ViPZonE: Hardware Power Variability-Aware Memory Management for Energy Savings. *IEEE Transactions on Computers (TC)* 64, 5 (2015), 1483–1496.

[20] Mark Gottscho, Clayton Schoeny, Lara Dolecek, and Puneet Gupta. 2016. Software-Defined Error-Correcting Codes. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W).*

[21] Puneet Gupta, Yuvraj Agarwal, Lara Dolecek, Nikil Dutt, Rajesh K. Gupta, Rakesh Kumar, Subhasish Mitra, Alexandru Nicolau, Tajana Simunic Rosing, Mani B. Srivastava, Steven Swanson, and Dennis Sylvester. 2013. Underdesigned and Opportunistic Computing in Presence of Hardware Variability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 32, 1 (2013), 8–23.

[22] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the IEEE International Workshop on Workload Characterization (IWWC).*

[23] Said Hamdioui, Georgi Gaydadjiev, and Ad J. van de Goor. 2004. The State-of-art and Future Trends in Testing Embedded Memories. In *International Workshop on Memory Technology, Design and Testing (MTDT)*.

[24] Said Hamdioui, Ad J. van de Goor, and Mike Rodgers. 2002. March SS: A Test for All Static Simple RAM Faults. In *International Workshop on Memory Technology, Design, and Testing (MTDT)*.

[25] Nam Sung Kim, Krisztian Flautner, David Blaauw, and Trevor Mudge. 2004. Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 12, 2 (2004), 167–184.

[26] Liangzhen Lai. 2015. *Cross-Layer Approaches for Monitoring, Margining and Mitigation of Circuit Variability*. Ph.D. Dissertation. University of California, Los Angeles (UCLA).

[27] Serge Lamikhov-Center. 2016. ELFIO: C++ Library for Reading and Generating ELF Files. (2016). http://elfio.sourceforge.net/

[28] F. Li, G. Chen, M. Kandemir, and I. Kolcu. 2005. Improving Scratch-Pad Memory Reliability Through Compiler-Guided Data Block Duplication. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[29] Man-Lap Li, Pradeep Ramachandran, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. 2008. 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)*.

[30] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[31] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G Zorn. 2011. 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)*.

[32] Shih-Lien Lu, Qiong Cai, and Patrick Stolt. 2013. Memory Resiliency. *Intel Technology Journal* 17, 1 (2013).

[33] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. 2014. 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)*.

[34] Tayyeb Mahmood, Seokin Hong, and Soontae Kim. 2015. Ensuring Cache Reliability and Energy Scaling at Near-Threshold Voltage with Macho. *IEEE Transactions on Computers (TC)* 64, 6 (2015), 1694–1706.

[35] Mehrtash Manoochehri, Murali Annavaram, and Michel Dubois. 2011. CPPC: Correctable Parity Protected Cache. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[36] Michail Mavropoulos, Georgios Keramidas, and Dimitris Nikolos. 2015. A Defect-Aware Reconfigurable Cache Architecture for Low-Vccmin DVFS-Enabled Systems. In *Design, Automation, and Test in Europe (DATE)*.

[37] Sparsh Mittal. 2014. A Survey of Architectural Techniques for Improving Cache Power Efficiency. *Sustainable Computing: Informatics and Systems* 4, 1 (2014), 33–43.

[38] Sparsh Mittal. 2016. A Survey of Architectural Techniques for Managing Process Variation. *Comput. Surveys* 48, 4 (2016).

[39] Amir Mahdi Hosseini Monazzah, Hamed Farbeh, Seyed Ghassem Miremadi, Mahdi Fazeli, and Hossein Asadi. 2013. FTSPM: A Fault-Tolerant ScratchPad Memory. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

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

[41] Preeti Ranjan Panda, Nikil Dutt, and Alexandru Nicolau. 1999. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*.

[42] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. 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)*.

[43] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. 2000. 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)*.

[44] Moinuddin K. Qureshi and Zeshan Chishti. 2013. Operating SECDED-Based Caches at Ultra-Low Voltage with FLAIR. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[45] Ashish Ranjan, Swagath Venkataramani, Xuanyao Fong, Kaushik Roy, and Anand Raghunathan. 2015. Approximate Storage for Energy Efficient Spintronic Memories. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*.

[46] Mohamed M. Sabry, David Atienza, and Francky Catthoor. 2014. OCEAN: An Optimized HW/SW Reliability Mitigation Approach for Scratchpad Memories in Real-Time SoCs. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014).

[47] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[48] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2013. Approximate Storage in Solid-State Memories. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[49] Hossein Sayadi, Hamed Farbeh, Amir Mahdi Hosseini Monazzah, and Seyed Ghassem Miremadi. 2014. 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)*.

[50] Mark F. Schilling. 2012. The Surprising Predictability of Long Runs. *Mathematics Magazine* 85, 2 (2012), 141–149.

[51] Stanley E. Schuster. 1978. Multiple Word/Bit Line Redundancy for Semiconductor Memories. *IEEE Journal of Solid-State Circuits (JSSC)* 13, 5 (1978), 698–703.

[52] Philip P. Shirvani and Edward J. McCluskey. 1999. PADded Cache: A New Fault-Tolerance Technique for Cache Memories. In *Proceedings of the VLSI Test Symposium*.

[53] Majid Shoushtari, Abbas BanaiyanMofrad, and Nikil Dutt. 2015. Exploiting Partially-Forgetful Memories for Approximate Computing. *IEEE Embedded Systems Letters (ESL)* 7, 1 (2015), 19–22.

[54] Jiguo Song, Gedare Bloom, and Gabriel Palmer. 2016. SuperGlue: IDL-Based, System-Level Fault Tolerance for Embedded Systems. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[55] Rick van Rein. 2016. BadRAM: Linux Kernel Support for Broken RAM Modules. (2016).

[56] Daniel P. Volpato, Alexandre K.I. Mendonca, Luiz C.V. dos Santos, and José Luís Güntzel. 2010. 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)*. 127–132.

[57] Jiajing Wang and Benton H. Calhoun. 2011. Minimum Supply Voltage and Yield Estimation for Large SRAMs Under Parametric Variations. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 19, 11 (2011), 2120–2125.

[58] Lucas Wanner, Charwak Apte, Rahul Balani, Puneet Gupta, and Mani Srivastava. 2013. Hardware Variability-Aware Duty Cycling for Embedded Sensors. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 21, 6 (2013), 1000–1012.

[59] Lucas Wanner, Liangzhen Lai, Abbas Rahimi, Mark Gottscho, Pietro Mercati, Chu-Hsiang Huang, Frederic Sala, Yuvraj Agarwal, Lara Dolecek, Nikil Dutt, Puneet Gupta, Rajesh Gupta, Ranjit Jhala, Rakesh Kumar, Sorin Lerner, Subhasish Mitra, Alexandru Nicolau, Tajana Simunic Rosing, Mani B. Srivastava, Steve Swanson, Dennis Sylvester, and Yuanyuan Zhou. 2015. NSF Expedition on Variability-Aware Software: Recent Results and Contributions. *De Gruyter Information Technology (it)* 57, 3 (2015).

[60] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. 2014. The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.0. (2014).

[61] Chris Wilkerson, Hongliang Gao, Alaa R. Alameldeen, Zeshan Chishti, Muhammad Khellah, and Shih-Lien Lu. 2008. Trading off Cache Capacity for Reliability to Enable Low Voltage Operation. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[62] Jack K. Wolf. 1965. On an Extended Class of Error-Locating Codes. *Information and Control* 8, 2 (1965), 163–169.

[63] J. K. Wolf and B. Elspas. 1963. Error-Locating Codes – A New Concept in Error Control. *IEEE Transactions on Information Theory* 9, 2 (1963), 113–117.

[64] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K Iyer. 2002. Architecture Support for Defending Against Buffer Overflow Attacks. In *Workshop on Evaluating and Architecting Systems for Dependability*.

[65] Chao Yan and Russ Joseph. 2016. Enabling Deep Voltage Scaling in Delay Sensitive L1 Caches. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[66] Jun Yang, Youtao Zhang, and Rajiv Gupta. 2000. Frequent Value Compression in Data Caches. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 258–265.

[67] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design and Test* 34, 2 (2017), 60–68.

Received April 7, 2017; revised June 9, 2017; accepted June 30, 2017