Lightweight Software-Defined Error Correction for Memories

Irina Alam, Lara Dolecek, and Puneet Gupta

The key observation behind the techniques described in this chapter is that most if not all error correction techniques and codes assume that all words stored in the memory are equally likely and important. This obviously is not true due to architectural or application context. This chapter devises new coding and correction mechanisms which leverage software or architecture "side information" to dramatically reduce the cost of error correction (Fig. 1). The methodology proposed in Sect. 1 is for recovering from detected-but-uncorrectable (DUE) errors in main memories while Sects. 2 and 3 focus on lightweight correction in on-chip caches or embedded memories.

1 Software-Defined Error Correcting Codes (SDECC)

This section focuses on the concept of Software-Defined Error Correcting Codes (SDECC), a general class of techniques spanning hardware, software, and coding theory that improves the overall resilience of systems by enabling heuristic best-effort recovery from detected-but-uncorrectable errors (DUE). The key idea is to add software support to the hardware error correcting code (ECC) so that most memory DUEs can be heuristically recovered based on available *side information* (SI) from the corresponding un-corrupted cache line contents. SDECC does not degrade memory performance or energy in the common cases when either no errors or purely hardware-correctable errors occur. Yet it can significantly improve resilience in the critical case when DUEs actually do occur.

I. Alam · L. Dolecek · P. Gupta (✉)
ECE Department, UCLA, Los Angeles, CA, USA
e-mail: irina1@ucla.edu; dolecek@ee.ucla.edu; puneetg@ucla.edu

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_9

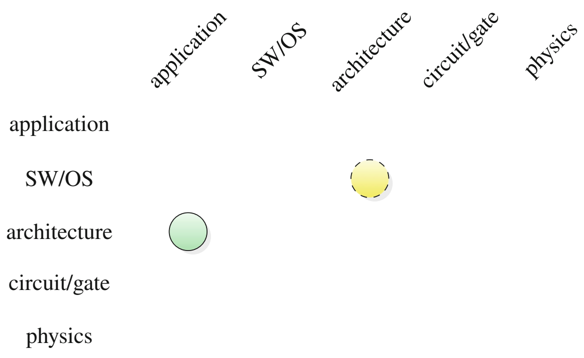


Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

Details of the concepts discussed in this section can be found in the works by Gottscho et al. [11, 12].

1.1 SDECC Theory

Important terms and notation introduced here are summarized in Table 1.

A (t)*SC*$(t+1)$*SD code corrects up to t symbol errors and/or detects up to (t+1) symbol errors. SDECC is based on the fundamental observation that when a* $(t+1)$*-symbol DUE occurs in a* (t)*SC*$(t+1)$*SD code, there remains significant information in the received string* $\mathbf{x}$. This information can be used to recover the original message $\mathbf{m}$ with reasonable certainty.

It is not the case that the original message was completely lost, i.e., one need not naïvely choose from all q^k possible messages. If there is a $(t+1)$ DUE, there are exactly

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

ways that the $(t+1)$ DUE could have corrupted the original codeword, which is less than q^k. Though a (t)SC$(t+1)$SD code can often detect more than $(t+1)$ errors, a $(t+1)$error is usually much more likely than higher bit errors. But guessing correctly out of N possibilities is still difficult. *In practice, there are just a handful of possibilities: they are referred to as* $(t+1)$DUE corrupted candidate codewords *(or candidate messages).*

Consider Fig. 2, which depicts the relationships between codewords, correctable errors (CEs), DUEs, and candidate codewords for individual DUEs for a Single-bit Error Correcting, Double-bit Error Detecting (SECDED) code. If the hardware ECC decoder registers a DUE, there can be several equidistant candidate codewords at the

Table 1 Important SDECC-specific notation

Term	Description
n	Codeword length in symbols
k	Message length in symbols
r	Parity length in symbols
b	Bits per symbol
q	Symbol alphabet size
t	Max. guaranteed correctable symbols in codeword
(t)SC$(t+1)$SD	(t)-symbol-correcting, $(t+1)$-symbol-detecting
N	Number of ways to have a DUE
μ	Mean no. of candidate codewords $\forall$ possible DUEs
P_G	Prob. of choosing correct codeword for a given DUE
$\overline{P_G}$	Avg. prob. of choosing correct codeword $\forall$ possible DUEs
d_{min}	Minimum symbol distance of code
`linesz`	Total cache line size in symbols (message content)
symbol	Logical group of bits
SECDED	Single-bit-error-correcting, double-bit-error-detecting
DECTED	Double-bit-error-correcting, triple-bit-error-detecting
SSCDSD	Single-symbol-error-correcting, double-symbol-error-detecting
ChipKill-correct	ECC construction and mem. organization that either corrects up to 1 DRAM chip failure or detects 2 chip failures

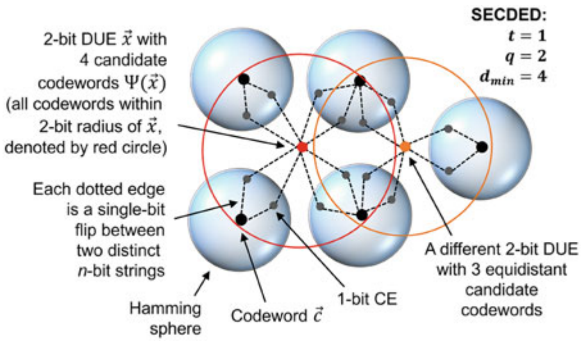

SECDED:
$t = 1$
$q = 2$
$d_{min} = 4$
2-bit DUE $\vec{x}$ with 4 candidate codewords $\Psi(\vec{x})$ (all codewords within 2-bit radius of $\vec{x}$, denoted by red circle)
Each dotted edge is a single-bit flip between two distinct n-bit strings
Hamming sphere
Codeword $\vec{c}$
1-bit CE
A different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

Fig. 2 Illustration of candidate codewords for 2-bit DUEs in the imaginary 2D-represented Hamming space of a binary SECDED code $(t = 1, q = 2, d_{min} = 4)$. The actual Hamming space has n dimensions

q-ary Hamming distance of exactly $(t + 1)$ from the received string $\mathbf{x}$. Without any *side information* (SI) about message probabilities, under conventional principles, each candidate codeword is assumed to be equally likely. However, in the specific case of DUEs, not all messages are equally likely to occur: this allows to leverage SI about memory contents to help choose the right candidate codeword in the event of a given DUE.

1.1.1 Computing the List of Candidates

The number of candidate codewords for any given $(t+1)$ DUE $\mathbf{e}$ has a linear upper bound that makes DUE recovery tractable to implement in practice [12]. The candidate codewords for any $(t+1)$-symbol DUE received string $\mathbf{x}$ is simply the set of equidistant codewords that are exactly $(t+1)$ symbols away from $\mathbf{x}$. This list depends on the error $\mathbf{e}$ and original codeword $\mathbf{c}$, but only the received string $\mathbf{x}$ is known. Fortunately, there is a simple and intuitive algorithm to find the list of candidate codewords with runtime complexity $O(nq/t)$. The detailed algorithm can be found in [12]. The essential idea is to try every possible single symbol *perturbation* $\mathbf{p}$ on the received string. Each *perturbed string* $\mathbf{y} = \mathbf{x} + \mathbf{p}$ is run through a simple software implementation of the ECC decoder, which only requires knowledge of the parity-check matrix $\mathbf{H}$ ($O(rn\log q)$ bits of storage). Any $\mathbf{y}$ characterized as a CE produces a candidate codeword from the decoder output and added to the list (if not already present in the list).

1.1.2 SDECC Analysis of Existing ECCs

Code constructions exhibit structural properties that affect the number of candidate codewords. In fact, distinct code constructions with the same $[n, k, d_{min}]_q$ parameters can have different values of μ and distributions of the number of candidate codewords. μ depends on the total number of minimum weight non-$\overrightarrow{0}$ codewords [12].

The SDECC theory is applied to seven code constructions of interest: SECDED, DECTED, and SSCDSD (ChipKill-Correct) constructions with typical message lengths of 64, and 128 bits. Table 2 lists properties that have been derived for each of them. Most importantly, the final column lists $\overline{P_G}$—the average (*random* baseline) probability of choosing correct codeword without SI for all possible DUEs. These probabilities are far higher than the naïve approaches of guessing randomly from q^k possible messages or from the N possible ways to have a DUE. Thus, SDECC can handle DUEs in a more optimistic way than conventional ECC approaches.

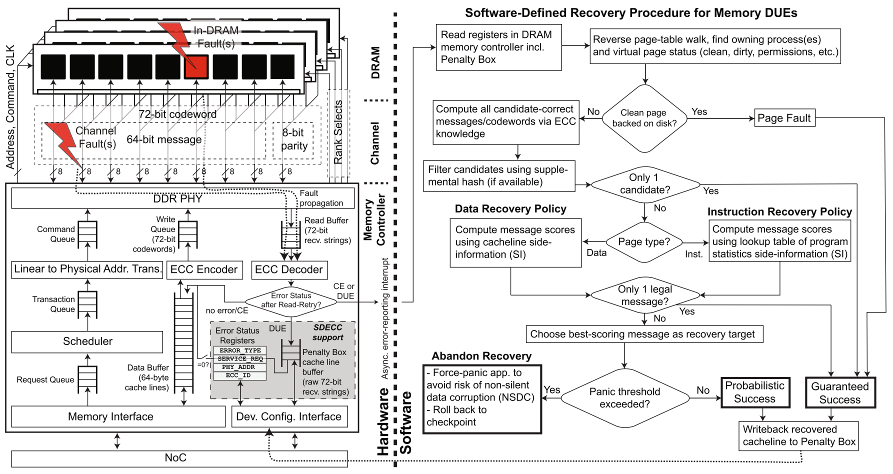
Table 2 Summary of code properties—$\overline{P_G}$ is most important for SDECC

Class of code	Code params. $[n, k, d_{min}]_q$	Type of code	Class of DUE $(t+1)$	Avg. # Cand. μ	Prob. Rcov. $\overline{P_G}$
32-bit SECDED	$[39, 32, 4]_2$	Hsiao [16]	2-bit	12.04	8.50%
32-bit SECDED	$[39, 32, 4]_2$	Davydov [7]	2-bit	9.67	11.70%
64-bit SECDED	$[72, 64, 4]_2$	Hsiao [16]	2-bit	20.73	4.97%
64-bit SECDED	$[72, 64, 4]_2$	Davydov [7]	2-bit	16.62	6.85%
32-bit DECTED	$[45, 32, 6]_2$	–	3-bit	4.12	28.20%
64-bit DECTED	$[79, 64, 6]_2$	–	3-bit	5.40	20.53%
128-bit SSCDSD	$[36, 32, 4]_{16}$	Kaneda [17]	2-sym.	3.38	39.88%

1.2 SDECC Architecture

SDECC consists of both hardware and software components to enable recovery from DUEs in main memory DRAM. A simple hardware/software architecture whose block diagram is depicted in Fig. 3 can be used. Although the software flow includes an instruction recovery policy, it is not presented in this chapter because DUEs on instruction fetches are likely to affect clean pages that can be remedied using a page fault (as shown in the figure).

The key addition to hardware is the *Penalty Box*: a small buffer in the memory controller that can store each codeword from a cache line (shown on the left-hand side of Fig. 3). When a memory DUE occurs, hardware stores information about the error in the Penalty Box and raises an error-reporting interrupt to system software. System software then reads the Penalty Box, derives additional context about the error—and using basic coding theory and knowledge of the ECC implementation—quickly computes a list of all possible *candidate messages*, one of which is guaranteed to match the original information that was corrupted by the DUE. A software-defined data recovery policy heuristically recovers the DUE in a best-effort manner by choosing the most likely remaining candidate based on available *side information* (SI) from the corresponding un-corrupted cache line contents; if confidence is low, the policy instead forces a panic to minimize the risk of accidentally induced mis-corrected errors (MCEs) that result in intolerable non-silent data corruption (NSDC). Finally, system software writes back the *recovery target* message to the Penalty Box, which allows hardware to complete the afflicted memory read operation.



Software-Defined Recovery Procedure for Memory DUEs
Address, Command, CLK
In-DRAM Fault(s)
72-bit codeword
Channel Fault(s)
64-bit message
8-bit parity
Rank Selects
DRAM
Channel
Memory Controller
DDR PHY
Fault propagation
Command Queue
Write Queue (72-bit codewords)
Read Buffer (72-bit recv. strings)
Linear to Physical Addr. Trans.
ECC Encoder
ECC Decoder
CE or DUE
Error Status after Read-Retry?
no error/CE
Transaction Queue
Scheduler
Error Status Registers
DUE
SDECC support
ERROR_TYPE
SERVICE_REQ
PHY_ADDR
ECC_ID
=0?
Penalty Box cache line buffer (raw 72-bit recv. strings)
Request Queue
Data Buffer (64-byte cache lines)
Memory Interface
Dev. Config. Interface
NoC
Async. error-reporting interrupt
Hardware
Software
Read registers in DRAM memory controller incl. Penalty Box
Reverse page-table walk, find owning process(es) and virtual page status (clean, dirty, permissions, etc.)
Compute all candidate-correct messages/codewords via ECC knowledge
No
Clean page backed on disk?
Yes
Page Fault
Filter candidates using supplemental hash (if available)
Only 1 candidate?
Yes
No
Data Recovery Policy
Instruction Recovery Policy
Compute message scores using cacheline side-information (SI)
Page type?
Data
Inst.
Compute message scores using lookup table of program statistics side-information (SI)
Only 1 legal message?
Yes
No
Choose best-scoring message as recovery target
Abandon Recovery
- Force-panic app. to avoid risk of non-silent data corruption (NSDC)
- Roll back to checkpoint
Yes
Panic threshold exceeded?
No
Probabilistic Success
Guaranteed Success
Writeback recovered cacheline to Penalty Box

Fig. 3 Block diagram of a general hardware and software implementation of SDECC. The figure depicts a typical DDRx-based main memory subsystem with 64-byte cache lines, x8 DRAM chips, and a $[72, 64, 4]_2$ SECDED ECC code. Hardware support necessary to enable SDECC is shaded in gray. The instruction recovery policy is outside the scope of this work [12]

Overheads The area and power overhead of the essential SDECC hardware support is negligible. The area required per Penalty Box is approximately 736 μm^2 when synthesized with 15 nm Nangate technology—this is approximately one millionth of the total die area for a 14 nm Intel Broadwell-EP server processor [9]. The SDECC design incurs no latency or bandwidth overheads for the vast majority of memory accesses where no DUEs occur. This is because the Penalty Box and error-reporting interrupt are not on the critical path of memory accesses. When a DUE occurs, the latency of the handler and recovery policy is negligible compared to the expected mean time between DUEs or typical checkpoint interval of several hours.

1.3 Data Recovery Policy

In this section, recovery of DUEs in data (i.e., memory reads due to processor loads) is discussed because they are more vulnerable than DUEs in instructions as mentioned before. Possible recovery policies for instruction memory have been discussed in [11]. There are potentially many sources of SI for recovering DUEs in data. Based on the notion of *data similarity*, a simple but effective data recovery policy called *Entropy-Z* is discussed here that chooses the candidate that minimizes overall cache line Shannon entropy.

1.3.1 Observations on Data Similarity

Entropy is one of the most powerful metrics to measure data similarity. Two general observations can be made about the prevalence of low data entropy in memory.

- **Observation 1.** There are only a few primitive data types supported by hardware (e.g., integers, floating-point, and addresses), which typically come in multiple widths (e.g., byte, halfword, word, or quadword) and are often laid out in regular fashion (e.g., arrays and structs).
- **Observation 2.** In addition to spatial and temporal locality in their memory access patterns, applications have inherent *value locality* in their data, regardless of their hardware representation. For example, an image-processing program is likely to work on regions of pixels that exhibit similar color and brightness, while a natural language processing application will see certain characters and words more often than others.

Similar observations have been made to compress memory [2, 18, 24, 26, 28, 35] and to predict [20] or approximate processor load values [22, 23, 36]. Low byte-granularity intra-cache line entropy is observed throughout the integer and floating-point benchmarks in the SPEC CPU2006 suite. Let $P(X)$ be the normalized relative frequency distribution of a `linesz`$\times b$-bit cache line that has been carved into equal-sized Z-bit symbols, where each symbol χ_i can take 2^Z possible values.[1] Then the Z-bit-granularity `entropy` is computed as follows:

$$\texttt{entropy} = -\sum_{i=1}^{\texttt{linesz}\times b/Z} P(\chi_i) log_2 P(\chi_i). \tag{2}$$

The average intra-cacheline byte-level entropy of the SPEC CPU2006 suite was found to be 2.98 bits (roughly half of maximum).

These observations can be leveraged using the data recovery policy *Entropy-Z Policy*. With this policy, SDECC first computes the list of candidate messages using the algorithm described in Sect. 1.1.1 and extracts the cache line side information. Each candidate message is then inserted into appropriate position in the affected cache line and the entropy is computed using Eq. 2. The policy then chooses the candidate message that minimizes overall cache line entropy. The chance that the policy chooses the wrong candidate message is significantly reduced by deliberately forcing a *panic* whenever there is a tie for minimum entropy or if the mean cache line entropy is above a specified threshold `PanicThreshold`. The downside to this approach is that some forced panics will be false positives, i.e., they would have otherwise recovered correctly.

In the rest of the chapter, unless otherwise specified, $Z = 8$ bits, $\texttt{linesz}\times b =$ 512 bits and `PanicThreshold` = 4.5 bits (75% of maximum entropy) are used, which were determine to work well across a range of applications. Additionally, the *Entropy-8* policy performs very well compared to several alternatives.

1.4 Reliability Evaluation

The impact of SDECC is evaluated on system-level reliability through a comprehensive error injection study on memory access traces. The objective is to estimate the fraction of DUEs in memory that can be recovered correctly using the SDECC architecture and policies while ensuring a minimal risk of MCEs.

1.4.1 Methodology

The SPEC CPU2006 benchmarks are compiled against GNU/Linux for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [34] using the official tools [25]. Each benchmark is executed on top of the RISC-V proxy kernel [32] using the Spike simulator [33] that was modified to produce representative memory access

[1]Entropy symbols are not to be confused with the codeword symbols, which can also be a different size.

traces. Each trace consists of randomly sampled 64-byte demand read cache lines, with an average interval between samples of one million accesses.

Each trace is analyzed offline using a MATLAB model of SDECC. For each benchmark and ECC code, 1000 q-ary messages from the trace were chosen randomly and encoded, and were injected with $min(1000, N)$ randomly sampled $(t + 1)$-symbol DUEs. N here is the number of ways to have a DUE. For each codeword/error pattern combination, the list of candidate codewords was computed and the data recovery policy was applied. A *successful recovery* occurs when the policy selects a candidate message that matches the original; otherwise, the policy either causes a *forced panic* or recovery fails by accidentally inducing an MCE. Variability in the reported results is negligible over many millions of individual experiments.

Note that the *absolute* error magnitudes for DUEs and SDECC's impact on *overall* reliability should not be compared directly between codes with distinct $[n, k, d_{min}]_q$ (e.g., a double-bit error for SECDED is very different from a double-chip DUE for ChipKill). Rather, what matters most is the *relative* fraction of DUEs that can be saved using SDECC for a given ECC code.

Entropy-8 is exclusively used as the data recovery policy in all the evaluations. This is because when the raw successful recovery rates of six different policies for three ECCs without including any forced panics were compared, *Entropy-8* performed the best [12]. Few examples of alternate policies include *Entropy-Z* policy variants with $Z = 4$ and $Z = 16$ and *Hamming* which chooses the candidate that minimizes the average binary Hamming distance to the neighboring words in the cacheline. The 8-bit entropy symbol size performs best because its alphabet size ($2^8 = 256$ values) matches well with the number of entropy symbols per cacheline (64) and with the byte-addressable memory organization. For instance, both *Entropy-4* and *Entropy-16* do worse than *Entropy-8* because the entropy symbol size results in too many aliases at the cacheline level and because the larger symbol size is less efficient, respectively.

1.4.2 Recovery Breakdown

SDECC is evaluated next for each ECC using its conventional form, to understand the impact of the recovery policy's *(Entropy-8)* forced panics on the successful recovery rate and the MCE rate. The overall results with forced panics *taken* (main results, gray cell shading) and *not taken* are shown in Table 3.

There are two baseline DUE recovery policies: *conventional* (always panic for every DUE) and *random* (choose a candidate randomly, i.e., $\overline{P_G}$). It is observed that when panics are taken the MCE rate drops significantly by a factor of up to 7.3× without significantly reducing the success rate. This indicates that the `PanicThreshold` mechanism appropriately judges when SDECC is unlikely to correctly recover the original information.

These results also show the impact of code construction on successes, panics, and MCEs. When there are fewer average candidates μ then the chances of successfully

Table 3 Percent Breakdown of SDECC *Entropy-8* Policy (M = MCE, P = forced panic, S = success) [12]

	Panics taken			Panics not taken			*Random* baseline		
	M	P	S	M	P	S	M	P	S
Conv. baseline	–	100	–						
$[39, 32, 4]_2$ Hsiao	5.3	25.6	69.1	27.3	–	72.7	91.5	–	8.5
$[39, 32, 4]_2$ Davydov	4.5	25.2	70.3	24.0	–	76.0	88.3	–	11.7
$[72, 64, 4]_2$ Hsiao	4.7	23.7	71.6	24.7	–	75.3	95.0	–	5.0
$[72, 64, 4]_2$ Davydov	4.1	21.9	74.0	22.3	–	77.7	93.2	–	6.9
$[45, 32, 6]_2$ DECTED	2.2	20.3	77.5	14.5	–	85.5	71.8	–	28.2
$[79, 64, 6]_2$ DECTED	1.5	14.5	84.0	11.0	–	89.0	79.5	–	20.5
$[36, 32, 4]_{16}$ SSCDSD	1.5	12.8	85.7	8.5	–	91.5	60.1	–	39.9

recovering are much higher than that of inducing MCEs. The $[72, 64, 4]_2$ SECDED constructions perform similarly to their $[39, 32, 4]_2$ variants even though the former have lower baseline $\overline{P_G}$. This is a consequence of the *Entropy-8* policy: larger n combined with lower μ provides the greatest opportunity to differentiate candidates with respect to overall intra-cacheline entropy. For the same n, however, the effect of SECDED construction is more apparent. The Davydov codes recover about 3–4% more frequently than their Hsiao counterparts when panics are not taken (similar to the baseline improvement in $\overline{P_G}$). When panics are taken, however, the differences in construction are less apparent because the policy `PanicThreshold` does not take into account Davydov's typically lower number of candidates.

The breakdown between successes, panics, and MCEs is examined in more detail. Figure 4 depicts the DUE recovery breakdowns for each ECC construction and SPEC CPU2006 benchmark when forced panics are taken. Figure 4a shows the fraction of DUEs that result in success (black), panics (gray), and MCEs (white). Figure 4b further breaks down the forced panics (gray from Fig. 4a) into a fraction that are *false positive* (light purple, and would have otherwise been correct) and others that are *true positive* (dark blue, and managed to avoid an MCE). Each cluster of seven stacked bars corresponds to the seven ECC constructions.

It can be seen that much lower MCE rates are achieved than the *random* baseline yet also panic much less often than the *conventional* baseline for all benchmarks, as shown in Fig. 4a. This policy performs best on integer benchmarks due to their lower average intra-cacheline entropy. For certain floating-point benchmarks, however, there are many forced panics because they frequently have high data entropy above `PanicThreshold`. A `PanicThreshold` of 4.5 bits for these cases errs on the side of caution as indicated by the false positive panic rate, which can be up to 50%. Without more side information, for high-entropy benchmarks, it would be difficult for any alternative policy to frequently recover the original information with a low MCE rate and few false positive panics.

With almost no hardware overheads, SDECC used with SSCDSD ChipKill can recover correctly from up to 85.7% of double-chip DUEs while eliminating 87.2% of would-be panics; this could improve system availability considerably. However,

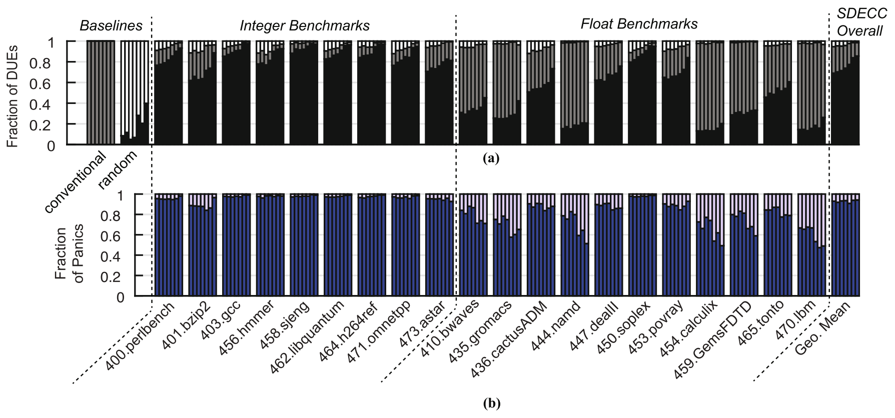


Fig. 4 Detailed breakdown of DUE recovery results when forced panics are taken. Results are shown for all seven ECC constructions, listed left to right within each cluster: $[39, 32, 4]_2$ Hsiao SECDED—$[39, 32, 4]_2$ Davydov SECDED—$[72, 64, 4]_2$ Hsiao SECDED—$[72, 64, 4]_2$ Davydov SECDED—$[45, 32, 6]_2$ DECTED—$[79, 64, 6]_2$ DECTED—$[36, 32, 4]_{16}$ SSCDSD ChipKill-Correct. (**a**) Recovery breakdown for the *Entropy-8* policy, where each DUE can result in an unsuccessful recovery causing an MCE (white), forced panic (gray), or successful recovery (black). (**b**) Breakdown of forced panics (gray bars in (**a**)). A true positive panic (dark blue) successfully mitigated a MCE, while a false positive panic (light purple) was too conservative and thwarted an otherwise-successful recovery [12]

SDECC with ChipKill introduces a 1% risk of converting a DUE to an MCE. Without further action taken to mitigate MCEs, this small risk may be unacceptable when application correctness is of paramount importance.

2 Software-Defined Error-Localizing Codes (SDELC): Lightweight Recovery from Soft Faults at Runtime

For embedded memories, it is always challenging to address reliability concerns as additional area, power, and latency overheads of reliability techniques need to be minimized as much as possible. *Software-Defined Error-Localizing Codes* (SDELC) is a hybrid hardware/software technique that deals with single-bit soft faults at runtime 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 recovery policy. 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.

SDELC then relies on *side information* (SI) about application memory contents to heuristically recover from the single-bit fault. Unlike the general-purpose Software-Defined ECC (SDECC), SDELC focuses on heuristic error recovery that is suitable for microcontroller-class IoT devices.

Details of the concepts discussed in this section can be found in the work by Gottscho et al. [13].

2.1 Ultra-Lightweight Error-Localizing Codes (UL-ELC)

In today's systems, 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.

Localizing an error is more useful than simply detecting it. If the error is localized to 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., assign a dedicated parity bit to each chunk. However, this method is very inefficient because to create C chunks C parity bits are needed: essentially, split up the memory words into smaller pieces.

Instead *Ultra-Lightweight* ELCs (UL-ELCs) is simple and customizable—given r redundant parity bits—it 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 can be used to form the parity-check matrix $\mathbf{H}$ for the UL-ELC (for single-bit errors, the error syndrome is simply one of the columns of $\mathbf{H}$). To create a UL-ELC code, a distinct non-zero binary column vector of length r bits is assigned to each chunk. 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 and are called *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.

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:

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

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: (1) the length of the data message is independent of r and (2) each chunk can be an arbitrary length. The freedom to choose the length of the code and chunk sizes allows 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.

2.2 Recovering SEUs in Instruction Memory

This section focuses on 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. This SDELC implementation example targets the open-source and free 64-bit RISC-V (RV64G) ISA [34], 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 big-endian is used in this example.

The 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 4. The `opcode`, `rd`, `funct3`, and `rs1` fields are the most commonly used—and potentially the most critical—among the possible instruction encodings, so each of them is assigned 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

Table 4 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-`UJ`	`imm[20|10:1|11|19:12]`					`rd`	`opcode`	`parity`
Type-`U`	`imm[31:12]`					`rd`	`opcode`	`parity`
Type-`I`	`imm[11:0]`			`rs1`	`funct3`	`rd`	`opcode`	`parity`
Type-`R`	`funct7`		`rs2`	`rs1`	`funct3`	`rd`	`opcode`	`parity`
Type-`S`	`imm[11:5]`		`rs2`	`rs1`	`funct3`	`imm[4:0]`	`opcode`	`parity`
Type-`SB`	`imm[12|10:5]`		`rs2`	`rs1`	`funct3`	`imm[4:1|11]`	`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-	11111	00	00000	11111	000	11111	1111111	1	0	0
Check	00000	11	00000	00000	111	11111	1111111	0	1	0
H	00000	00	11111	11111	111	00000	1111111	0	0	1

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 4, the UL-ELC decoder can be certain that the `opcode` field has been corrupted.

The instruction recovery policy consists of three steps.

- **Step 1.** A software-implemented instruction decoder is applied to filter out any candidate messages that are illegal instructions. Most bit patterns decode to illegal instructions in three RISC ISAs that were 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, the probability of each valid message is estimated using a small pre-computed lookup table that contains the relative frequency that each instruction appears. The relative frequencies of legal instructions in most applications follow power-law distribution [13]. This is used to favor more common instructions.
- **Step 3.** The instruction that is most common according to the SI lookup table is chosen. In the event of a tie, the instruction with the longest leading-pad of 0s or 1s is chosen. This is because in many instructions, the MSBs represent immediate values (as shown in Table 4). These MSBs are usually low-magnitude signed integers or they represent 0-dominant function codes.

If the SI is strong, then there is normally a higher chance of correcting the error by choosing the right candidate.

2.3 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, evenly spaced UL-ELC constructions can be used and the software trap handler can be granted additional control about how to recover from errors, similar to the general idea from SuperGlue [31].

The SDELC recovery support can be built 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-cache line 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* [20].

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 [4, 5, 10, 21, 29, 37].

2.4 SDELC Architecture

The SDELC architecture is illustrated in Fig. 5 for a system with split on-chip instruction and data scratchpad memories (SPMs) (each with its own UL-ELC code) and a single-issue core that has an in-order pipeline.

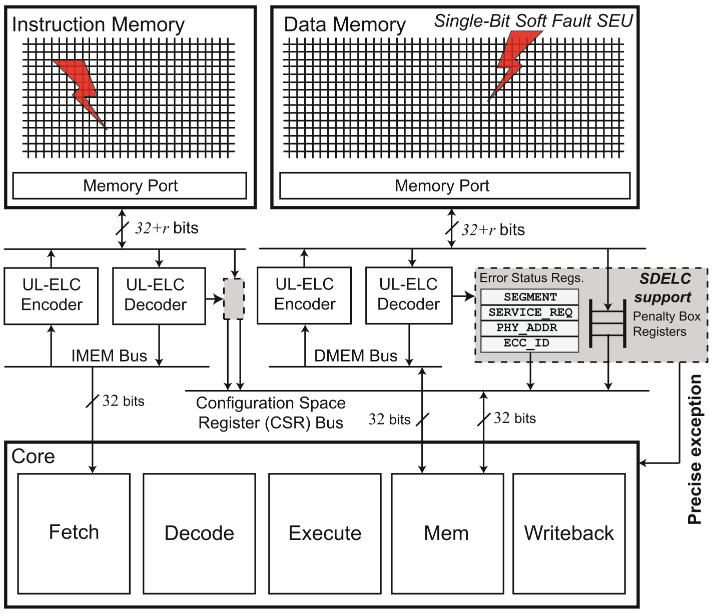


Fig. 5 Architectural support for SDELC on an microcontroller-class embedded system

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.

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, the candidate codeword is written back to instruction memory. If it is not accessible by the load/store unit, one could use hardware debug support such as JTAG. The previously trapped instruction is re-executed after returning from the trap handler, 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. The chosen candidate codeword is written back to data memory to scrub the error, the register file is updated appropriately, and `pc` is manually advanced before returning from the trap handler.

2.5 Soft Fault Recovery Using SDELC

To evaluate SDELC, Spike was modified to produce representative memory access traces of 11 benchmarks as they run to completion. Five benchmarks are `blowfish` and `sha` from the MiBench suite [14] as well as `dhrystone`, `matmulti`, and `whetstone`. The remaining six benchmarks were added from the AxBench approximate computing C/C++ suite [37]: `blackscholes`, `fft`, `inversek2j`, `jmeint`, `jpeg`, and `sobel`. Each trace was analyzed offline using a MATLAB model of SDELC. For each workload, 1000 instruction fetches and 1000 data reads were randomly selected from the trace and exhaustively all possible single-bit faults were applied to each of them.

SDELC recovery of the random soft faults was evaluated 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 4). A *successful recovery* for SDELC occurs when the policy corrects the error; otherwise, it fails by accidentally mis-correcting.

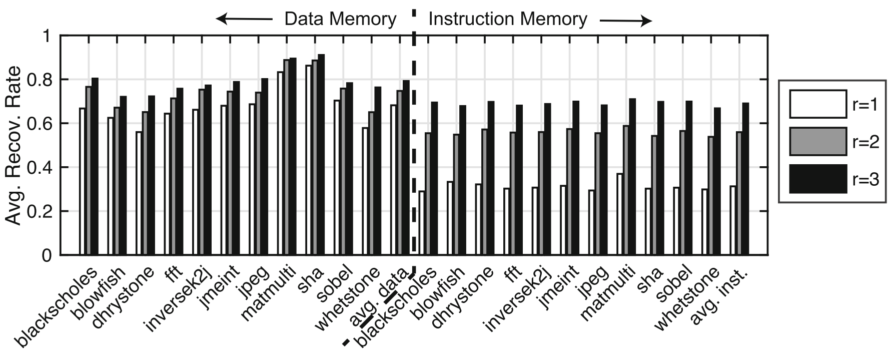


Fig. 6 Average rate of recovery using SDELC from single-bit soft faults in instruction and data memory. r is the number of parity bits in the UL-ELC construction

2.5.1 Overall Results

The overall SDELC results are presented in Fig. 6. The recovery rates are relatively consistent over each benchmark, especially for instruction memory faults, providing evidence of the general efficacy of SDELC. 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, these results are achieved 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 these results, using $r = 1$ parity for data seems reasonable, while $r = 3$ UL-ELC constructions can be used to achieve 70% recovery for both memories with minimal overhead.

3 Parity++ : Lightweight Error Correction for Last Level Caches and Embedded Memories

This section focuses on another novel lightweight error correcting code—Parity++: a novel lightweight unequal message protection scheme for last level caches or embedded memories 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-bit Error Detection (SED) code while providing SED for all messages and Single-bit Error Correction (SEC) for a subset of messages. Thus, it is stronger than just basic SED parity and has much lower parity storage overhead (3.5× and 4× lower for 32-bit and 64-bit memories, respectively) than a traditional Single-bit Error Correcting, Double-bit 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. This 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 or lightweight embedded memories. Parity++ is also evaluated with a memory speculation procedure [8] that can be generally applied to any ECC protected cache to hide the decoding latency while reading messages when there are no errors.

Details of the concepts discussed in this section can be found in the work by Alam et al. [1] and Schoeny et al. [30].

3.1 Application Characteristics

As mentioned in Sects. 1.3 and 2.2, data in applications is generally very structured and instructions mostly follow power-law distribution. This means most instructions in the memory would have the same opcode. Similarly, 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. The 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 [3, 35], base-delta-immediate compression [27], and bit-plane compression [19]. However, the main goal here is to provide stronger error protection to these special messages that are chosen based on the knowledge of data patterns in context.

3.2 Parity++ Theory

Parity++ is a type of *unequal message protection* code, in that specific messages are designated a priori to have extra protection against errors as shown in Fig. 7. As in [30], 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, it is referred to as messages; when discussing error detection/correction capabilities it is referred to as codewords.

Codewords in Parity++ have the following error protection guarantees: normal codewords have single-error detection; special codewords have single-error cor-

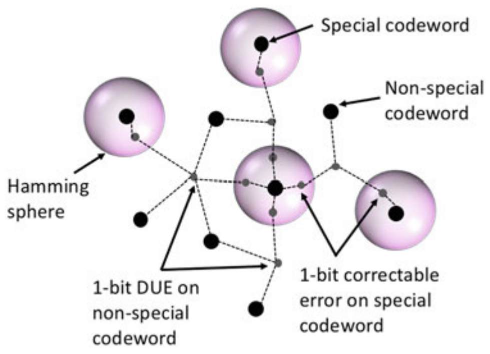

Special codeword
Non-special codeword
Hamming sphere
1-bit DUE on non-special codeword
1-bit correctable error on special codeword

Fig. 7 Conceptual illustration of Parity++ for 1-bit error (CE = Correctable Error, DUE = Detected but Uncorrectable Error)

rection. Let us partition the codewords in the code C into two sets, $\mathcal{N}$ and $\mathcal{S}$, representing the normal and special codewords, respectively. The minimum distance properties necessary for the aforementioned error protection guarantees of Parity++ are as follows:

$$\min_{\mathbf{u},\mathbf{v}\in\mathcal{N},\mathbf{u}\neq\mathbf{v}} d_H(\mathbf{u},\mathbf{v}) \geq 2, \tag{3}$$

$$\min_{\mathbf{u}\in\mathcal{N},\mathbf{v}\in\mathcal{S}} d_H(\mathbf{u},\mathbf{v}) \geq 3, \tag{4}$$

$$\min_{\mathbf{u},\mathbf{v}\in\mathcal{S},\mathbf{u}\neq\mathbf{v}} d_H(\mathbf{u},\mathbf{v}) \geq 3. \tag{5}$$

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., $n = k + 2$. Thus, Parity++ requires only two bits of redundancy.

For the context of this work, let us assume that Parity++ always has message length k as a power of 2. The overall approach to constructing the code is to create a Hamming subcode of a SED code [15]; when an error is detected, it is decoded 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, let us go through the steps of constructing the (34, 32) Parity++ code (as opposed to the generic $(k + 2, k)$ Parity++ code).

The first step is to create 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 this case, the (63, 57) Hamming code is used. Let α be a primitive element of GF(2^6) such that $1+x+x^6 = 0$, then the generator polynomial is simply $g_S(x) = 1 + x + x^6$ (and the generator matrix is constructed using the usual polynomial coding methods). The next step is to shorten this code to (32, 26) by expurgating and puncturing (i.e., deleting) the right and bottom 31 columns and rows. Then add a column of 1s to the end, resulting in a generator matrix, which is denoted 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 the (34, 32) Parity++ code, $\mathbf{G_N}$ is added 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, a generator matrix is built 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 the special messages. Note that Conditions 3 and 5 are satisfied; however, Condition 4 is not satisfied. To meet the requirement, a single non-linear parity bit is added that is a NOR of the bits corresponding to $\mathbf{G_N}$, in this 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 the (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 that are deemed to be special do not have to follow this pattern as one can simply apply a pre-mapping before the encoding step, and a post-mapping after the decoding step.

In the (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.

Similar unequal message protection scheme can be used for providing DECTED protection to special messages, while non-special messages get SECDED protection. The code construction has been explained in detail in [30].

3.3 Error Detection and Correction

The received—possibly erroneous—vector $\mathbf{y}$ is divided 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 non-linear 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, $\mathbf{H}_S$, its corresponding parity-check matrix can be easily retrieved. 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 us take a look at two concrete examples for the $(10, 8)$ Parity++ code. Without any pre-mapping, a special message begins with $\log_2(3) + 1 = 4$ zeros. Let the 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.

3.4 Architecture

In an ECC protected cache, every time a cache access is initiated, the target block is sent through the ECC decoder/error detection engine. If no error is detected, the cache access is completed and the cache block is sent to the requester. If an error is detected, the block is sent through the ECC correction engine and the corrected block is eventually sent to the requester. 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 does not fall in the critical path. However, a block goes through the detection engine every time a cache access is initiated.

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 the analyses.

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++, all messages go through the chain of XOR gates for error detection and only the non-systematic 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.

3.5 Experimental Methodology

Parity++ was evaluated over applications from the SPEC 2006 benchmark suite. Two sets of core micro-architectural parameters (provided in Table 5) 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 [6], fast forwarding for one billion instructions and executing for two billion instructions.

The first processor was 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 were 4-way associative. The LLC was a unified 1MB 8-way associative L2 cache. The second processor was a dual core out-of-order architecture. The L1 instruction and data caches had the same configuration as the previous processor. The LLC comprises of both L2 and L3 caches. The L2 was a shared 512KB cache while the L3 was a shared 2MB 16-way associative cache. For both the baseline processors it was assumed that the LLCs (L2 for the InO processor and L2 and L3 for the OoO processor) have SECDED ECC protection.

Table 5 Core micro-architectural parameters

	Processor-1	Processor-2
Cores	1 (@ 2 GHz)	2 (@ 2 GHz)
Core type	InO (@ 2 GHz)	OoO (@ 2 GHz)
Cache line size	64B	64B
L1 Cache per core	32 KB I$, 64 KB D$	32 KB I$, 64 kB D$
L2 Cache	1 MB (unified) 8-way	512 KB (shared, unified) 8-way
L3 Cache	–	2 MB 16-way (shared)
Memory configuration	4 GB of 2133 MHz DDR3	8 GB of 2133 MHz DDR3
Nominal voltage	1 V	1 V

The performance evaluation was done only for cases where there are no errors. Thus, latency due to error detection was taken into consideration but not error correction as correction is rare when compared to the processor cycle time and does not 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 was increased by ∼9% due to the lower storage overhead of Parity++ compared to SECDED. This is the iso-area case since the additional area coming from reduction in redundancy is used to increase the total capacity of the last level caches.

3.6 Results and Discussion

In this section the performance results obtained from the Gem5 simulations (as mentioned in Sect. 3.5) are discussed. Figures 8 and 9 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.

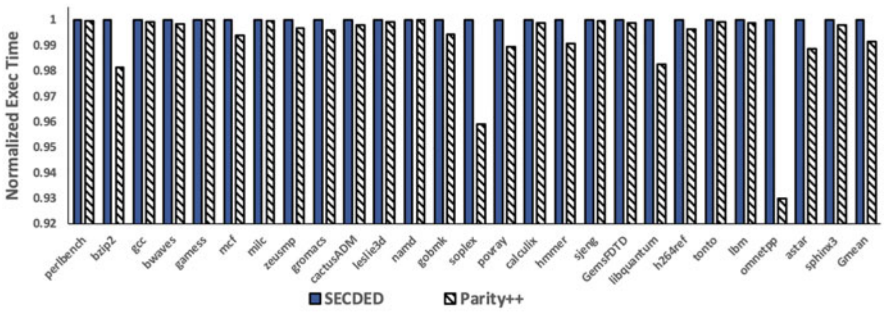


Fig. 8 Comparing normalized execution time of Processor-I with SECDED and Parity++

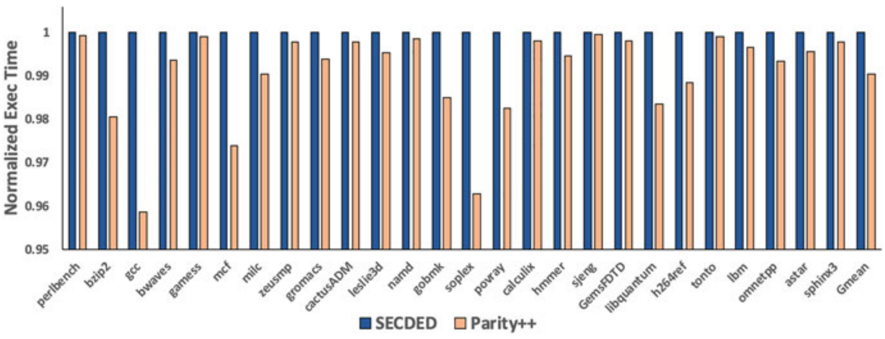


Fig. 9 Comparing normalized execution time of Processor-II with SECDED and Parity++

For both the core configurations, the observations are almost similar. It was considered that both Parity++ and SECDED protected caches have additional cache hit latency of one cycle (due to ECC decoding) for all read operations. The results show that with the exact same hit latency, Parity++ has up to 7% lower execution time than SECDED due to the additional memory capacity. The applications showing higher performance benefits are mostly memory intensive. Hence, additional cache capacity with Parity++ reduces overall cache miss rate. 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++ does not lead to a significant improvement in performance.

On the other hand, if the cache capacity is kept constant (iso-capacity), Parity++ helps to save $\sim$5–9% of last level cache area (cache tag area taken into consideration) as compared to SECDED. 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 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 [1], it provides a better protection mechanism than SECDED in such devices.

Acknowledgments The authors thank Dr. Mark Gottscho and Dr. Clayton Schoeny who were collaborators on the work presented in this chapter.

References

1. Alam, I., Schoeny, C., Dolecek, L., Gupta, P.: Parity++: lightweight error correction for last level caches. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), pp. 114–120 (2018). https://doi.org/10.1109/DSN-W.2018.00048
2. Alameldeen, A., Wood, D.: Frequent pattern compression: a significance-based compression scheme for L2 caches. Tech. rep., University of Wisconsin, Madison (2004)
3. Alameldeen, A.R., Wood, D.A.: Frequent pattern compression: a significance-based compression scheme for L2 caches (2004)
4. Bathen, L.A.D., Dutt, N.D.: E-RoC: embedded RAIDs-on-chip for low power distributed dynamically managed reliable memories. In: Design, Automation, and Test in Europe (DATE) (2011)
5. Bathen, L.A.D., Dutt, N.D., Nicolau, A., Gupta, P.: VaMV: variability-aware memory virtualization. In: Design, Automation, and Test in Europe (DATE) (2012)

6. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (2011). http://doi.acm.org/10.1145/2024716.2024718
7. Davydov, A., Tombak, L.: An alternative to the Hamming code in the class of SEC-DED codes in semiconductor memory. IEEE Trans. Inf. Theory **37**(3), 897–902 (1991)
8. Duwe, H., Jian, X., Kumar, R.: 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 (2015). https://doi.org/10.1109/HPCA.2015.7056055
9. Gelas, J.D.: The Intel Xeon E5 v4 review: testing broadwell-EP with demanding server workloads (2016). http://www.anandtech.com/show/10158/the-intel-xeon-e5-v4-review
10. Gottscho, M., Bathen, L.A.D., Dutt, N., Nicolau, A., Gupta, P.: ViPZonE: hardware power variability-aware memory management for energy savings. IEEE Trans. Comput. **64**(5), 1483–1496 (2015)
11. Gottscho, M., Schoeny, C., Dolecek, L., Gupta, P.: Software-defined error-correcting codes. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), pp. 276–282 (2016)
12. Gottscho, M., Schoeny, C., Dolecek, L., Gupta, P.: Software-defined ECC: heuristic recovery from uncorrectable memory errors. Tech. rep., University of California, Los Angeles, Oct 2017
13. Gottscho, M., Alam, I., Schoeny, C., Dolecek, L., Gupta, P.: Low-cost memory fault tolerance for IoT devices. ACM Trans. Embed. Comput. Syst. **16**(5s), 128:1–128:25 (2017)
14. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: a free, commercially representative embedded benchmark suite. In: Proceedings of the IEEE International Workshop on Workload Characterization (IWWC) (2001)
15. Hamming, R.W.: Error detecting and error correcting codes. Bell Labs Tech. J. **29**(2), 147–160 (1950)
16. Hsiao, M.Y.: A class of optimal minimum odd-weight-column SEC-DED codes. IBM J. Res. Dev. **14**(4), 395–401 (1970)
17. Kaneda, S., Fujiwara, E.: Single byte error correcting – double byte error detecting codes for memory systems. IEEE Trans. Comput. **C-31**(7), 596–602 (1982)
18. Kim, J., Sullivan, M., Choukse, E., Erez, M.: Bit-plane compression: transforming data for better compression in many-core architectures. In: Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA) (2016)
19. Kim, J., Sullivan, M., Choukse, E., Erez, M.: Bit-plane compression: transforming data for better compression in many-core architectures. In: Proceedings of the 43rd International Symposium on Computer Architecture, ISCA '16, pp. 329–340. IEEE, Piscataway (2016). https://doi.org/10.1109/ISCA.2016.37
20. Lipasti, M.H., Wilkerson, C.B., Shen, J.P.: Value locality and load value prediction. In: Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (1996)
21. Liu, S., Pattabiraman, K., Moscibroda, T., Zorn, B.G.: 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)
22. Miguel, J.S., Badr, M., Jerger, N.E.: Load value approximation. In: Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO) (2014)
23. Miguel, J.S., Albericio, J., Jerger, N.E., Jaleel, A.: The Bunker Cache for spatio-value approximation. In: Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO) (2016)
24. Mittal, S., Vetter, J.: A survey of architectural approaches for data compression in cache and main memory systems. IEEE Trans. Parallel Distrib. Syst. **27**(5), 1524–1536 (2015)
25. Nguyen, Q.: RISC-V tools (GNU toolchain, ISA simulator, tests) – git commit 816a252. https://github.com/riscv/riscv-tools

26. Pekhimenko, G., Seshadri, V., Mutlu, O., Gibbons, P.B., Kozuch, M.A., Mowry, T.C.: 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)
27. Pekhimenko, G., Seshadri, V., Mutlu, O., Gibbons, P.B., Kozuch, M.A., Mowry, T.C.: 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, pp. 377–388. ACM, New York (2012). http://doi.acm.org/10.1145/2370816.2370870
28. Pekhimenko, G., Seshadri, V., Kim, Y., Xin, H., Mutlu, O., Gibbons, P.B., Kozuch, M.A., Mowry, T.C.: Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In: Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO) (2013)
29. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: 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)
30. Schoeny, C., Sala, F., Gottscho, M., Alam, I., Gupta, P., Dolecek, L.: Context-aware resiliency: unequal message protection for random-access memories. In: Proc. IEEE Information Theory Workshop, Kaohsiung, pp. 166–170, Nov 2017
31. Song, J., Bloom, G., Palmer, G.: 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)
32. Waterman, A.: RISC-V proxy kernel – git commit 85ae17a. https://github.com/riscv/riscv-pk/commit/85ae17a
33. Waterman, A., Lee, Y.: Spike, a RISC-V ISA Simulator – git commit 3bfc00e. https://github.com/riscv/riscv-isa-sim
34. Waterman, A., Lee, Y., Patterson, D., Asanovic, K.: The RISC-V instruction set manual volume I: user-level ISA version 2.0 (2014). https://riscv.org
35. Yang, J., Zhang, Y., Gupta, R.: Frequent value compression in data caches. In: Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO) (2000)
36. Yazdanbakhsh, A., Pekhimenko, G., Thwaites, B., Esmaeilzadeh, H., Mutlu, O., Mowry, T.C.: Mitigating the memory bottleneck with approximate load value prediction. IEEE Des. Test **33**(1), 32–42 (2016)
37. Yazdanbakhsh, A., Mahajan, D., Esmaeilzadeh, H., Lotfi-Kamran, P.: AxBench: a multiplatform benchmark suite for approximate computing. IEEE Des. Test **34**(2), 60–68 (2017)