Software-Defined ECC: Heuristic Recovery from Uncorrectable Memory Errors

Mark Gottscho, Clayton Schoeny, Lara Dolecek, and Puneet Gupta

Electrical and Computer Engineering Department
University of California, Los Angeles
{mgottscho, cschoeny}@ucla.edu, {dolecek, puneet}@ee.ucla.edu

ABSTRACT

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

We demonstrate the feasibility of SDECC for linear (t)-symbol-correcting, $(t+1)$-symbol-detecting codes and analyze existing SECDED, DECTED, and SSCDSD ChipKill-Correct constructions. SDECC requires minimal architectural support in the memory controller and adds no performance, energy, or parity storage overheads during normal memory access when DUEs do not occur.

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

1. INTRODUCTION

Hardware reliability is now a central issue in computing. Memory systems are a limiting factor in system reliability [58] because they are primarily designed to maximize bit storage density; this makes them particularly sensitive to manufacturing process variation, environmental operating conditions, and aging-induced wearout [43, 54]. DRAM errors are common in warehouse-scale computers: Google has observed 70000 FIT/Mb in commodity memory, with 8% of modules affected per year [58], while Facebook has found that 2.5% of their servers have experienced memory errors per month [39]. Advances in memory resiliency can also help improve system energy efficiency [7].

Error-correcting codes (ECCs) are a classic way to build resilient memories by adding redundant parity bits or symbols. A code maps each information message to a unique codeword that allows a limited number of errors to be detected and/or corrected. Errors in the context of ECC can be broadly categorized as *corrected errors* (CEs), *detected-but-uncorrectable errors* (DUEs), *mis-corrected errors* (MCEs), and *undetected errors* (UDEs). CEs are harmless but are typically reported to system software anyway [19] as they may indicate the possibility of future DUEs. UDEs may result in *silent data corruption* (SDC) of software state, while MCEs may cause *non-silent data corruption* (NSDC); neither are desirable.

When a DUE occurs, the entire system usually panics, or in the case of a supercomputer, rolls back to a checkpoint to avoid data corruption. Both outcomes harm system availability and can cause some state to be lost. In this paper, we consider the problem of memory DUEs, because they are more common than MCEs and UDEs and remain a key challenge to the reliability and availability of extreme-scale systems. For instance, even with state-of-the-art strong memory protection using a ChipKill-Correct ECC, the Blue Waters supercomputer suffers from a memory DUE rate of 15.98 FIT/GB [38]. This rate is high enough that whole-system checkpoints would likely be required every few hours, and would add a significant performance and energy overhead to HPC applications [64]. For industry-standard SECDED codes that perform better and use less energy than ChipKill, DUEs are at least an order of magnitude more frequent [38] and compound the reliability/availability problem further.

The theoretical development of ECCs have – thus far – implicitly assumed that every message/information bit pattern is equally likely to occur. In general-purpose memory systems, however, this assumption does not hold true. For instance, applications exhibit unique characteristics in control flow and data that arise naturally from the algorithm, inputs, OS, ISA, and micro-architecture. For the first time, we demonstrate how to exploit some of these characteristics to enhance the capabilities of existing ECCs in order to recover from a large fraction of otherwise-harmful DUEs.

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

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

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

- Sec 3: We derive the theoretical basis for software to compute a short list of candidate messages/codewords – one of which is guaranteed to be correct – for any (t)-symbol-correcting, $(t+1)$-symbol-detecting code upon receipt of a $(t+1)$-symbol memory DUE.
- Sec 4: We analyze new properties of commonly-used ECCs that demonstrate the feasibility of SDECC for real systems. We also propose optional support for a second-tier hash that can improve SDECC by pruning the list of candidate codewords before recovery.
- Sec 5: We describe architectural support for SDECC with main memory. In general, SDECC requires minimal changes to existing DRAM controllers and no changes to the ECC design. There is also no performance or energy penalty in the common case, i.e., in the vast majority of memory accesses when DUEs do not occur. Our optional hash costs a small amount of extra parity bit storage. SDECC incurs no storage overheads when used with purely-conventional ECC constructions.
- Sec 6: We propose and evaluate a cacheline entropy-based recovery policy that utilizes SI about patterns of application data in memory to guide successful recovery from DUEs; when SI is weak, it instead aborts recovery by panicking.
- Sec 7: We evaluate the efficacy of SDECC with a comprehensive DUE injection campaign that uses representative SPEC CPU2006 traces. We compare our policy with alternatives and consider the impact of SI quality on recovery rates.
- Sec 8: We evaluate the impact of SDECC on approximate applications by using online DUE injections on the AxBench suite, and by tracking the effect of resulting MCEs on output quality (benign, tolerable NSDC, and intolerable NSDC).

We begin by covering the necessary basics of ECC in order to understand our theoretical contributions that immediately follow.

Table 1: Important Background Notation

Term	Description
$\mathscr{C}$	linear block error-correcting code
$\mathbf{G}$	generator matrix
$\mathbf{H}$	parity-check matrix
n	codeword length in symbols
k	message length in symbols
r	parity length in symbols
b	bits per symbol
q	symbol alphabet size
t	max. guaranteed correctable symbols in codeword
$\Delta_q(\vec{u},\vec{v})$	q-ary Hamming distance between $\vec{u}$ and $\vec{v}$
d_{min}	minimum symbol distance of code
$wt_q(\cdot)$	q-ary Hamming weight
$\vec{m}$	original/intended message
$\vec{c}$	original/intended codeword
$\vec{e}$	error that corrupts original codeword
$\vec{x}$	received string with error (corrupted codeword)
$\vec{s}$	parity-check syndrome
$[n,k,d_{min}]_q$	shorthand for crucial ECC parameters
(t)SC$(t+1)$SD	(t)-symbol-correcting, $(t+1)$-symbol-detecting

2. ECC BACKGROUND

We introduce fundamental ECC concepts that are necessary to understand the theory and analysis of SDECC. Table 1 summarizes the terms introduced in this section. Throughout this paper, we will refer to the most important code parameters using the shorthand notation $[n,k,d_{min}]_q$ (not to be confused with citations).

2.1 Key Concepts

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

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

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

2.2 Minimum Distance and Error Correction Guarantees

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

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

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

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

2.3 ECC Decoder Assumptions

The typical decoding method for ECC hardware is to choose the maximum-likelihood codeword [36] under two implicitly statistical assumptions.

- Assumption 1: all symbols in a codeword are equally likely to be afflicted by faults (the symmetric channel model).
- Assumption 2: all messages are equally likely to occur.

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

3. SDECC THEORY

Important terms and notation introduced in this section are summarized in Table 2.

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

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

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

Table 2: Important SDECC-Specific Notation

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

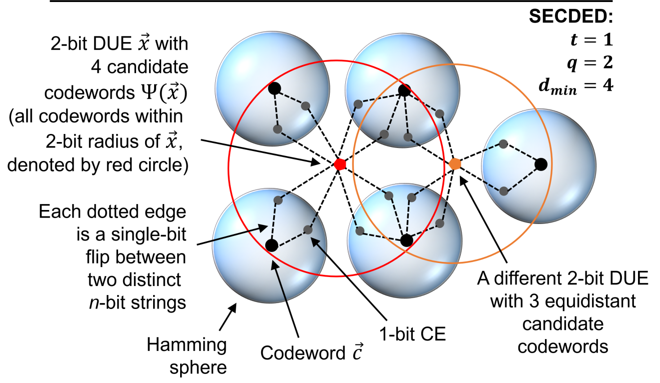

SECDED:
$t = 1$
$q = 2$
$d_{min} = 4$
2-bit DUE $\vec{x}$ with 4 candidate codewords $\Psi(\vec{x})$ (all codewords within 2-bit radius of $\vec{x}$, denoted by red circle)
Each dotted edge is a single-bit flip between two distinct n-bit strings
Hamming sphere
Codeword $\vec{c}$
1-bit CE
A different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

Figure 1: Illustration of candidate codewords for 2-bit DUEs in the imaginary 2D-represented Hamming space of a binary SECDED code. The actual Hamming space has n dimensions.

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

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

We retain Assumption 1 from Sec. 2.3, which means we assume all DUEs are equally likely to occur. However, in the specific case of DUEs, *we drop Assumption 2*: this allows us to leverage SI about memory contents to help choose the right candidate codeword in the event of a given DUE.

The size of the candidate codeword list $|\Psi(\vec{x})|$ is independent of the original codeword; it depends only on the error vector $\vec{e}$ due to linearity of the code $\mathscr{C}$. That is,

$$|\Psi(\vec{x})| = |\Psi(\vec{c} + \vec{e})| = |\Psi(\vec{e})|. \tag{2}$$

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

One can better understand candidate codewords by visualizing the Hamming space of a code. Consider Fig. 1, which depicts the relationships between codewords, CEs, DUEs, and candidate codewords for individual DUEs for a SECDED code.

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

3.1 Number of Candidate Codewords

The number of candidate codewords $|\Psi(\vec{e})|$ for any given

$(t+1)$ DUE $\vec{e}$ has a linear upper bound that makes DUE recovery tractable to implement in practice.

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

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

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

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

$$\overline{P_G} = \frac{1}{N} \sum_{\vec{e}} \frac{1}{|\Psi(\vec{e})|}. \quad (3)$$

3.2 Average Number of Candidates

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

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

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

PROOF. In order to find the average number of candidate codewords, we must sum the number of candidate codewords for each unique $(t+1)$ q-ary error $\vec{e}_E$ where $E = i_1, i_2, \cdots, i_{(t+1)}$, and $i_1 \neq i_2 \neq \cdots \neq i_{(t+1)}$. We then divide that sum by the number of error-vectors (n choose $(t+1)$). By linearity and without loss of generality, assume $\vec{c} = \vec{0}$. We know that the only codewords $\vec{c'} \in \mathscr{C}$ that can satisfy $\Delta(\vec{c} - \vec{c'}, \vec{e}) = (t+1)$ have weight $W_q(d_{min})$. Each such $\vec{c'}$ that has $\vec{c}_{i_1} = \vec{e}_{i_1}$, $\vec{c}_{i_2} = \vec{e}_{i_2}$, etc., then has ($d_{min}$ choose $(t+1)$) distinct error-vectors $\vec{e}_E$. Thus summing over all error-vectors, each codeword $\vec{c'}$ with $wt(\vec{c'}) = d_{min}$ contributes to (d_{min} choose $(t+1)$) candidate codewords. To average, we divide $[(d_{min}$ choose $(t+1))] \times W_q(d_{min})$ by (n choose $(t+1)$).

Algorithm 1 **Compute list of candidate codewords $\Psi(\vec{x})$ for a $(t+1)$-symbol DUE $\vec{x}$ in a linear (t)SC$(t+1)$SD code with parameters $[n,k,d_{min}]_q$. For error vectors, subscripts indicate the symbol positions of errors, but not their q-ary values. For example, $\vec{e}_3$ corresponds to $[00100\ldots0]$.**

```
for $i = 1 : n$ do
    for $j = 1 : q-1$ do
        $\vec{p} \leftarrow j * \vec{e}_i$ //(symbol $i$ in $p$ gets $q$-ary value $j$, all others 0)
        $\vec{y} \leftarrow \vec{x} + \vec{p}$
        if Decoder($\vec{y}$) not DUE then
            $\vec{c'} \leftarrow$ Decoder($\vec{y}$) //Compute candidate codeword
            if $\vec{c'} \notin \Psi(\vec{x})$ then //If candidate not already in list
                $\Psi(\vec{x}) \leftarrow \Psi(\vec{x}) \cup \vec{c'}$ //Add candidate to list
            end if
        end if
    end for
end for
```

We also divide by $(q-1)^{(t+1)}$ because each non-zero element of the error vector $\vec{e}_E$ can take values from 1 to $q-1$. Finally, we add 1 to the expression since the original codeword $\vec{c}$ is a candidate codeword for every possible error-vector, and was not already counted in $W_q(d_{min})$. □

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

3.3 Computing the List of Candidates

So far we have bounded the number of candidate codewords for any $(t+1)$-symbol DUE; we now show how to find these candidates. The candidate codewords $\Psi(\vec{x})$ for any $(t+1)$-symbol DUE received string $\vec{x}$ is simply the set of equidistant codewords that are exactly $(t+1)$ symbols away from $\vec{x}$. More formally, let subscripts be used to index symbols in a vector, starting from the most significant position. Then

$$\Psi(\vec{x}) = \vec{c} \cup \{\vec{c'} \in \mathscr{C} :$$
$$\Delta_q(\vec{c'} - \vec{c}) = d_{min}, \vec{c'}_i = \vec{x}_i \; \forall i \text{ where } \vec{e}_i \neq 0\}. \quad (4)$$

Notice that this equation depends on the error $\vec{e}$ and original codeword $\vec{c}$, but we only know the received string $\vec{x}$.

Fortunately, there is a simple and intuitive algorithm (shown in Alg. 1) to find the list of candidate codewords $\Psi(\vec{x})$ with runtime complexity $O(nq/t)$. The essential idea is to try every possible single symbol *perturbation* $\vec{p}$ on the received string. Each *perturbed string* $\vec{y} = \vec{x} + \vec{p}$ is run through a simple software implementation of the ECC decoder, which only requires knowledge of the parity-check matrix **H** ($O(rn\log q)$ bits of storage). Any $\vec{y}$ characterized as a CE produces a candidate codeword from the decoder output.

3.4 Pruning Candidates using a Lightweight Hash

In systems that require high reliability and availability, we propose to optionally prune the list of candidate codewords using a lightweight second-tier hash of several codewords grouped together in a cacheline. This would increase the success rate of SDECC while dramatically reducing the risk of MCEs. Several previous works [23, 25, 26, 65] have also proposed the use of RAID-like multi-tier codes, but using traditional checksums instead than hashes.

We observe that second-tier hashes can also be used to prune a list of candidate codewords for a DUE. For instance, we can compute a h-bit *original hash* of the `linesz`$\times b$ total message bits of a cacheline when it is written to memory. Then when a DUE occurs, after the candidate messages are found using Alg. 1, we can compute in software the *candidate hashes* for each *candidate cacheline* and compare them with the original that is read from memory. On average for a universal hash function, the number of *incorrect* candidates $|\Psi(\vec{x})| - 1$ will be reduced by a factor of 2^h. In most cases, only one candidate will match the original hash and we can fully correct the DUE; there is a chance of hash collision, in which case the number of candidates is still reduced but not down to one.

Errors in the original hash can cause candidate pruning to fail. However, this is a concern only when there is simultaneously both a $(t+1)$-symbol DUE $\Psi(\vec{x})$ in one of the cacheline codewords and an error in the hash. Although we consider this situation to be very unlikely, there are two possible outcomes for a universal hash.

- Outcome 1. The hash cannot prune the list of candidates because no candidates' computed hashes match. This case is fairly benign: SDECC just falls back to the full list of candidates. The probability of this lower bounded by
$$\geq (2^h - |\Psi(\vec{x})|)/(2^h - 1). \tag{5}$$
- Outcome 2. The corrupted hash collides with the computed hash of an incorrect candidate. Unfortunately, this case is not benign: it causes an MCE because the original message is mistakenly pruned along with other incorrect candidates. However, for all but the smallest hashes, the probability is much less than Outcome 1 and is upper bounded by
$$\leq (|\Psi(\vec{x})| - 1)/(2^h - 1). \tag{6}$$

We assume there is no more than one DUE per cacheline. Accordingly, we also assume the hash is not corrupted in order to maintain a consistent fault model. Our future work will explore SDECC in the context of detailed fault models where these assumptions can be revisited.

4. SDECC ANALYSIS OF EXISTING ECCS

Code constructions exhibit structural properties that affect the number of candidate codewords $|\Psi(\vec{e})|$. Certain combinations of error positions produce fewer candidate codewords than others. This favors recovery of certain errors even if one simply guesses from the corresponding list of candidate codewords. In fact, distinct code constructions with the same $[n,k,d_{min}]_q$ parameters can have different values of μ and distributions of $|\Psi(\vec{e})|$.

We apply the SDECC theory to seven code constructions of interest in this paper: SECDED, DECTED, and SSCDSD (ChipKill-Correct) constructions with typical message lengths of 32, 64, and 128 bits. Table 3 lists properties that we have derived for each of them. Most importantly, the final column lists $\overline{P_G}$ — the *random* baseline probability of successful recovery without SI.

These probabilities are far higher than the naïve approaches of guessing randomly from q^k possible messages or from the N possible ways to have a DUE. Thus, our approach can handle DUEs in a more optimistic way than conventional ECC approaches.

4.1 SECDED and DECTED

The class of SECDED codes ($t=1$, $q=2$, $d_{min}=4$) is simple and effective against random radiation-induced soft bit flips

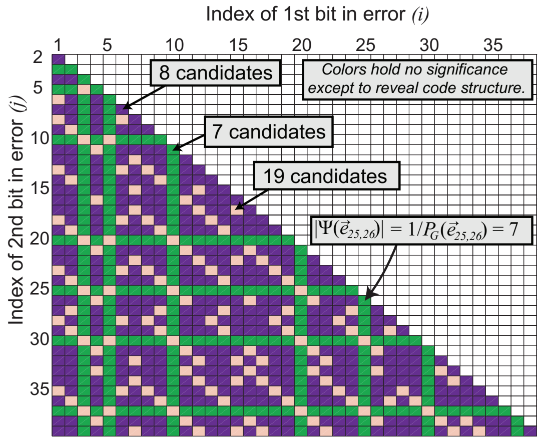


Figure 2: Number of candidate codewords $|\Psi(\vec{e}_{i,j})|$ for the Davydov $[39,32,4]_2$ SECDED code, where i and j represent the positions of the two bit-errors that cause a DUE.

in memory. Hsiao's $[39,32,4]_2$ and $[72,64,4]_2$ constructions are the most common implementations of SECDED because they minimize the number of decoder logic gates [21]. Davydov proposed alternative and more structured SECDED codes that instead minimize the probability of an MCE when there is a triple-bit error ($wt_q(\vec{e}) = 3$) by minimizing $W_2(4)$ [10]. We find that Davydov codes have an additional advantage in context of SDECC: Lemma 2 tells us that these Davydov SECDED constructions also minimize the average number of candidate codewords μ. This can lend them an advantage for heuristic recovery. Fig. 2 depicts how the structure of the $[39,32,4]_2$ Davydov construction determines the number of candidate codewords for all N possible DUE patterns. For the SECDED codes in this paper, the average number of candidate codewords μ ranges from 9.67 to 20.73, as shown in Table 3.

DECTED codes ($t=2$, $q=2$, $d_{min}=6$) can correct random 2-bit errors and detect 3-bit errors. While they are not typically used in commodity memory systems due to high overheads, they attract continued interest by industry and researchers. For this work, we simply add one extra parity bit to the $[127,113,5]_2$ and $[63,51,5]_2$ BCH codes [9] and then truncate them to obtain our own $[45,32,6]_2$ and $[79,64,6]_2$ DECTED constructions, respectively. For the DECTED codes in this paper, a baseline random-candidate recovery policy has around a 20-30% chance of success.

4.2 SSCDSD (ChipKill-Correct)

SSCDSD codes with 4-bit symbols ($t=1$, $q=16$, $d_{min}=4$) are a non-binary equivalent of SECDED codes. We use Kaneda's Reed-Solomon-based $[36,32,4]_{16}$ construction [27]. Messages are 128 bits long and codewords are 144 bits long, so they are convenient to deploy in industry-standard DDRx-based DRAM systems that are 72 bits wide. When two DRAM channels are run in lockstep with x4 DRAM chips, $[36,32,4]_{16}$ SSCDSD codes have the *ChipKill-Correct* property [12]. This is because they can completely correct any errors resulting from a single-chip failure, and detect any errors caused by a double-chip failure. We find that despite there being 141750 possible ways to have a double-chip DUE, on average, there are just 3.38 candidate codewords per DUE, with the random-candidate chance of success being 39.88%. Thus, we expect

Table 3: 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 $(t+1)$	# Min. Wt. Codew. $W_q(d_{min})$	# DUEs N	Avg. # Cand. μ	LBnd. Prob. Rcov. $1/\mu$	Prob. Rcov. $\overline{P_G}$
32-bit SECDED	$[39,32,4]_2$	Hsiao [21]	2-bit	1363	741	12.04	8.31%	8.50%
32-bit SECDED	$[39,32,4]_2$	Davydov [10]	2-bit	1071	741	9.67	10.34%	11.70%
64-bit SECDED	$[72,64,4]_2$	Hsiao [21]	2-bit	8404	2556	20.73	4.82%	4.97%
64-bit SECDED	$[72,64,4]_2$	Davydov [10]	2-bit	6654	2556	16.62	6.02%	6.85%
32-bit DECTED	$[45,32,6]_2$	–	3-bit	2215	14190	4.12	24.27%	28.20%
64-bit DECTED	$[79,64,6]_2$	–	3-bit	17404	79079	5.40	18.52%	20.53%
128-bit SSCDSD	$[36,32,4]_{16}$	Kaneda [27]	2-sym.	56310	141750	3.38	29.67%	39.88%

ChipKill to deliver the best results in our evaluation.

5. SDECC ARCHITECTURE

SDECC consists of both hardware and software components to enable recovery from DUEs in main memory DRAM. We propose a simple hardware/software architecture whose block diagram is depicted in Fig. 3 and will be referred throughout this section. Although the software flow includes an instruction recovery policy, we do not present it in this work because DUEs on instruction fetches are likely to affect clean pages that can be remedied using a page fault (as shown in the figure). In addition to basic hardware/software support, we also describe an implementation of the optional second-tier hash support for pruning candidates prior to recovery.

5.1 Penalty Box Hardware

The key addition to hardware is the *Penalty Box*: a small buffer in the memory controller that can store each codeword from a cacheline (shown on the left-hand side of Fig. 3). When a DUE occurs on a demand read (prefetch DUEs should have the request dropped), the ECC decoder writes the raw contents of the afflicted cacheline to the Penalty Box as-is (including the parity bits). It also asserts a new `SERVICE_REQ` bit in the error status register. The memory controller also blocks the forwarding of the afflicted cacheline to the requesting hardware resource. To avoid deadlock and performance degradation of the system, the controller still allows other memory requests to complete as usual in an out-of-order fashion. After the Penalty Box and `SERVICE_REQ` bit are set, the memory controller raises an asynchronous *non-maskable-interrupt* (NMI) to the OS to report the error. Similar registers and interrupts are supported in existing systems, e.g., Intel's Machine-Check Architecture [1].

Overheads. The area and power overhead of the essential SDECC hardware support is negligible. For example, with a typical $[72,64,4]_2$ SECDED or $[36,32,4]_{16}$ SSCDSD ChipKill-Correct ECC used in DRAM with 64-byte cache lines and a DDRx burst length of eight, the Penalty Box requires just 576 flip-flops arranged in a 72-wide and eight-deep shift register. The area required per Penalty Box is approximately $736\mu m^2$ when synthesized with 15nm Nangate technology — this is approximately one millionth of the total die area for a 14nm Intel Broadwell-EP server processor [18]. This shift register would add a negligible amount of leakage power. Our SDECC design incurs no latency or bandwidth overheads for the vast majority of memory accesses where no DUEs occur. This is because the Penalty Box and error-reporting interrupt are not on the critical path of memory accesses.

5.2 Software Stack

The OS responds to the interrupt (right-hand side of Fig. 3) and reads the Penalty Box and the Error Status Register through a device configuration interface (e.g., PCI). Software then reverse-walks the page tables to determine which process(es) own the physical page containing the DUE-afflicted cacheline. It also checks the status and permissions of the mapped virtual pages. If the page is backed on disk (clean), then a viable recovery solution is to unmap the DUE-afflicted physical page and re-allocate a new one filled with clean data from disk. If not, then we rely on a SDECC recovery policy. In this paper, we assume that DUE-afflicted pages are dirty.

The first task of the recovery policy is to compute the list of candidate codewords for the DUE. If a hash is available, it is used to prune the list of candidates. A heuristic recovery policy (presented in Sec. 6) scores each remaining candidate message using available SI. Successful recovery via choosing the best candidate message is probabilistic. Using appropriate statistical metrics, if the best-scoring candidate message is not sufficiently likely to be correct, then the policy forces the machine to panic or roll-back to a checkpoint. Recovery is abandoned whenever there are multiple DUEs per cacheline or if a second DUE arrives while the first is being handled; we do not consider these scenarios explicitly.

Overheads. When a DUE occurs, the latency of the handler and recovery policy is negligible compared to the expected mean time between DUEs or typical checkpoint interval of several hours. For instance, the total execution time of DUE recovery in our un-optimized offline MATLAB implementation is 9.6 ms (using a human-friendly string-based software ECC decoder for Alg. 1). A reasonable objective for an optimized implementation in C is ≤ 1 ms. This would fall within the measured range of 750 μs to 130 ms per error-reporting interrupt as reported by others [19].

5.3 Lightweight Hash Implementation

We compute a small universal hash in two steps that is easy to implement in hardware. First, we take the vertical parity of the cacheline to generate a kb-bit intermediary value. We then compact it to an h-bit hash using h randomly-generated balanced parity trees where each with $kb/2$ inputs. Experimentally, we find random inputs to distribute nearly uniformly across 2^h possible hash buckets (the ideal limit).

Our hash could be computed in one clock cycle and is only on the critical path for memory writes; the single-cycle latency can be hidden by pipelining. The critical path is three 2-input XOR levels for vertical parity (up to 448 gates total) and up to six logic levels for the random trees (up to 1008 gates total). For 15nm Nangate, the total required XOR gate area is up to 644 μm^2. Our approach resembles the X-Compact tree from VLSI test compaction [42] and micro-architectural

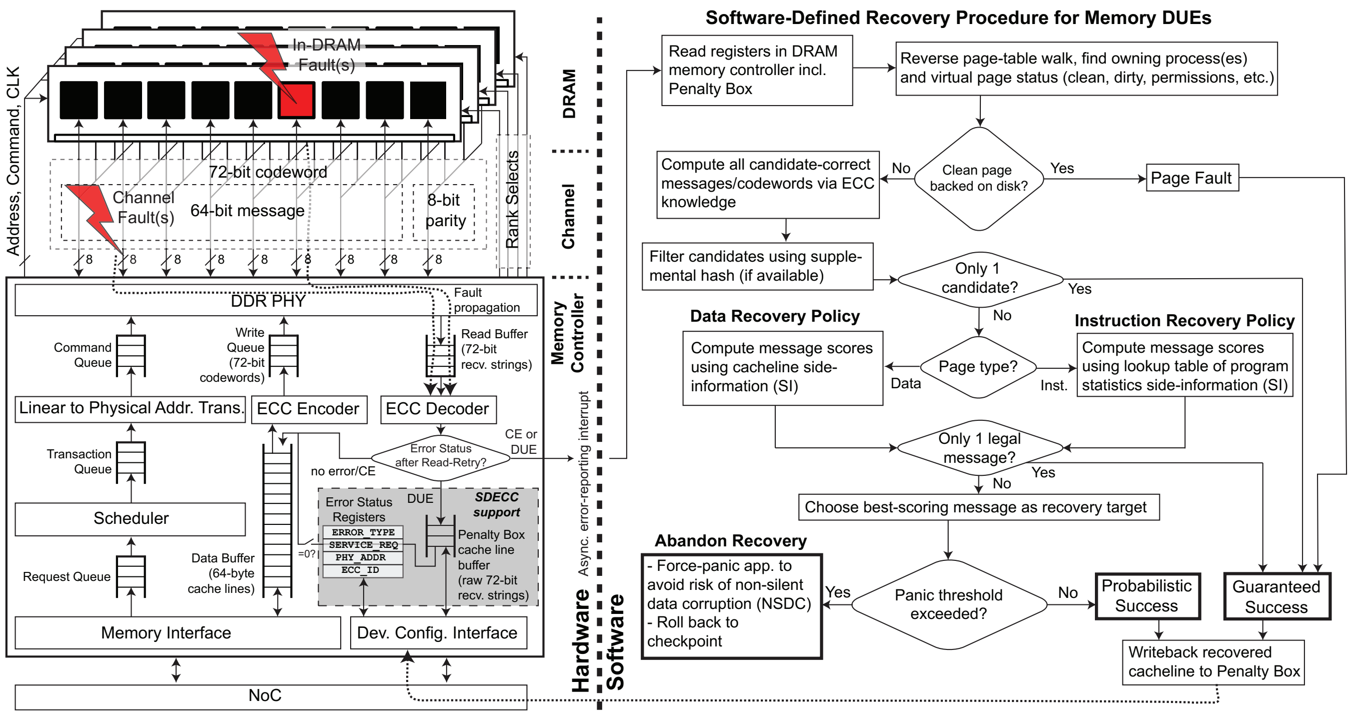
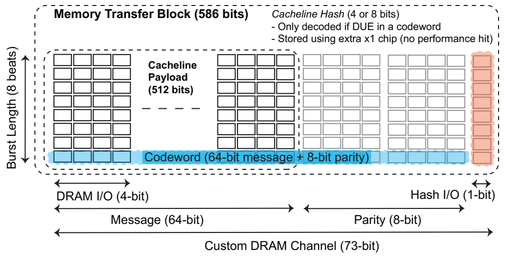
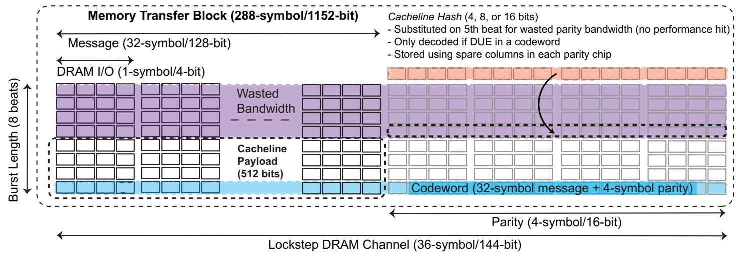

Software-Defined Recovery Procedure for Memory DUEs
Read registers in DRAM memory controller incl. Penalty Box
Reverse page-table walk, find owning process(es) and virtual page status (clean, dirty, permissions, etc.)
Clean page backed on disk?
Compute all candidate-correct messages/codewords via ECC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

Figure 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 (hash support not shown). The instruction recovery policy is outside the scope of this work.


Memory Transfer Block (586 bits)
Cacheline Hash (4 or 8 bits)
- Only decoded if DUE in a codeword
- Stored using extra x1 chip (no performance hit)
Cacheline Payload (512 bits)
Burst Length (8 beats)
Codeword (64-bit message + 8-bit parity)
DRAM I/O (4-bit)
Hash I/O (1-bit)
Message (64-bit)
Parity (8-bit)
Custom DRAM Channel (73-bit)

(a) $[72,64,4]_2$ **SECDED (modified DDRx channel)**


Memory Transfer Block (288-symbol/1152-bit)
Cacheline Hash (4, 8, or 16 bits)
- Substituted on 5th beat for wasted parity bandwidth (no performance hit)
- Only decoded if DUE in a codeword
- Stored using spare columns in each parity chip
Message (32-symbol/128-bit)
DRAM I/O (1-symbol/4-bit)
Wasted Bandwidth
Burst Length (8 beats)
Cacheline Payload (512 bits)
Codeword (32-symbol message + 4-symbol parity)
Parity (4-symbol/16-bit)
Lockstep DRAM Channel (36-symbol/144-bit)

(b) $[36,32,4]_{16}$ **SSCDSD (ChipKill-correct)**

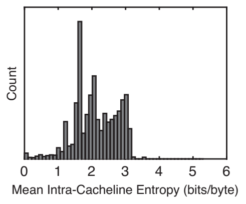
Figure 4: Proposed optional hardware support for short cacheline hashes with conventional SECDED and ChipKill-correct configurations. Lightweight hashes are useful to increase reliability of SDECC by pruning candidate messages in the event of a codeword DUE.

fingerprinting for processor error detection [59]. Commonly-used CRCs with the same number of check bits are a poor substitute for our hash function: they usually do not approach the universal hashing limit and are also infeasible to compute in a single clock cycle.

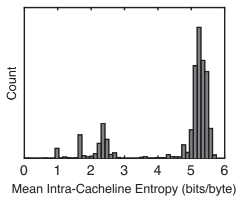
The second-tier hash could be accommodated in current DDRx memory systems with minor modifications. The hash is written to memory alongside the cacheline; during reads, if a DUE occurs, the original hash is stored in one additional h-bit register in the Penalty Box. Fig. 4 depicts two possible configurations for storing and accessing hash bits for $[72,64,4]_2$ SECDED and $[36,32,4]_{16}$ SSCDSD ChipKill-Correct memory organizations.

For SECDED (Fig. 4(a)), we propose that the standard 72-bit DRAM channel be widened by one bit to accommodate transfer of the hash bits in parallel with cacheline data as it is written. This would require an extra pin per memory module. Up to eight hash bits can be supported per cacheline; they would be stored using either an extra 1-bit-wide low-capacity DRAM device per rank, or by converting a single x4 DRAM per rank to a x5 DRAM. This design would have no impact on memory performance and a negligible impact on energy, but requires non-standard DRAM parts. For an $h = 8$-bit hash per cacheline, 1.56% additional storage is needed (128 MB per 16 GB rank).

For our ChipKill arrangement (Fig. 4(b)), we propose to transfer the hash during DDRx beats that would otherwise be wasted bandwidth (because the transfer size is larger than a cacheline). The hash bits could be stored using a few spare columns in each parity chip; up to 16 hash bits per cacheline could be supported for ChipKill with no externally-visible storage, performance, or energy overhead. If some spare columns contain local hard or soft faults, they are unlikely to have a system-level reliability impact because they are only used if there is a DUE on that particular cacheline. Corrupted hashes are not a major concern: on average, for the $[36,32,4]_{16}$ SSCDSD ChipKill-Correct code with an $h = 16$-bit hash, the probability of MCE caused by a corrupted hash (Eqn. 6) is



(a) `464.h264ref` **(integer)**



(b) `444.namd` **(float)**

Figure 5: Byte-granularity entropy distributions of 64-byte dynamic cacheline read data.

just 0.003%.

6. DATA RECOVERY POLICY

In this work, we focus on recovery of DUEs in data (i.e., memory reads due to processor loads) because they are more vulnerable than DUEs in instructions (i.e., memory reads due to instruction fetches) as explained earlier.

There are potentially many sources of SI for recovering DUEs. Based on the notion of *data similarity*, we propose a simple but effective data recovery policy called *Entropy-Z* that chooses the candidate that minimizes overall cacheline Shannon entropy.

6.1 Observations on Data Similarity

Entropy is one of the most powerful metrics to measure data similarity. We make two general observations about the prevalence of low data entropy in memory.

- Observation 1. There are only a few primitive data types supported by hardware (e.g., integers, floating-point, and addresses), which typically come in multiple widths (e.g., byte, halfword, word, or quadword) and are often laid out in regular fashion (e.g., arrays and structs).
- 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, 30, 44, 50, 51, 73] and to predict [37] or approximate processor load values [40, 41, 75].

We observe low byte-granularity intra-cacheline entropy throughout the integer and floating-point benchmarks in the SPEC CPU2006 suite. Let $P(X)$ be the normalized relative frequency distribution of a $\texttt{linesz}\times b$-bit cacheline that has been carved into equal-sized Z-bit symbols, where each symbol χ_i can take 2^Z possible values.[1] Then we compute the Z-bit-granularity `entropy` as follows:

$$\texttt{entropy} = -\sum_{i=1}^{\texttt{linesz}\times b/Z} P(\chi_i) log_2 P(\chi_i). \qquad (7)$$

Consider two representative examples for $Z = 8$ and $\texttt{linesz}\times b = 512$ bits in Fig. 5. The maximum possible intra-cacheline entropy here is six bits/byte because there can be only $2^6 = 64$ distinct byte values in a cacheline; anything less can be exploited as SI by SDECC recovery. We find that although floating-point values tend to have higher entropy *within* a word compared to integer values, entropy *between* neighboring words is often comparable. The average intra-cacheline byte-level entropy of the SPEC CPU2006 suite to be 2.98 bits/byte (roughly half of maximum).

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

Algorithm 2 ***Entropy-Z*** **data recovery policy. Given a q-ary list of n-symbol candidate codewords $\Psi(\vec{x})$, a q-ary list of n-symbol error-free neighboring cacheline codewords L_n (the SI), and a `PanicThreshold` value, produce a q-ary k-symbol recovery target message $\vec{m_{\text{target}}}$ and a flag `SuggestToPanic`.**

```
$M \leftarrow \Psi(\vec{x})$ with the $r$ parity symbols stripped //Extract candidate messages
$L_k \leftarrow L_n$ with the $r$ parity symbols stripped //Extract cacheline SI
Declare candidate entropy list entropy with $|M|$ elements
for $i = 1 : |M|$ do
    entropy[$i$] $\leftarrow$ $Z$-bit calculation for candidate cacheline //Eqn. 7
    //($M[i]$ inserted into appropriate position in $L_k$)
end for
entropy$_{\text{min}}$ $\leftarrow$ min(entropy[$i$]$\forall i$)
$i_{\text{min}} \leftarrow$ argmin(entropy[$i$]$\forall i$)
$\vec{m_{\text{target}}} \leftarrow M[i_{\text{min}}]$
if tie for entropy$_{\text{min}}$ or mean(entropy[$i$]$\forall i$) $>$ PanicThreshold then
    SuggestToPanic $\leftarrow$ True
else
    SuggestToPanic $\leftarrow$ False
end if
```

6.2 Entropy-Z Policy

We leverage these observations using our proposed data recovery policy, described in Alg. 2. Essentially, with this policy, SDECC chooses the candidate message that minimizes overall cacheline entropy. We mitigate the chance that our policy chooses the wrong candidate message by deliberately forcing a *panic* whenever there is a tie for minimum entropy or if the mean cacheline entropy is above a specified threshold `PanicThreshold`. The downside to this approach is that some forced panics will be false positives, i.e., they would have otherwise recovered correctly.

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

7. RELIABILITY EVALUATION

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

7.1 Methodology

The SPEC CPU2006 benchmarks are compiled against GNU/Linux for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [69] using the official tools [48]. Each benchmark is executed on top of the RISC-V proxy kernel [67] using the Spike simulator [68] that we modified to produce representative memory access traces. We only include the 20 benchmarks which successfully ran to completion. Each trace consists of randomly-sampled 64-byte demand read cache-

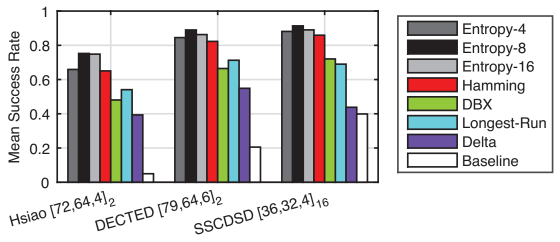


Figure 6: Comparison of raw success rate (no forced panics) for different SDECC data recovery policies averaged over all benchmarks. No hashes are used.

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, we randomly choose 1000 q-ary messages from the trace, encode them, and inject $min(1000, N)$ randomly-sampled $(t+1)$-symbol DUEs. For each codeword/error pattern combination, we compute the list of candidate codewords using Alg. 1 and apply the data recovery policy using Alg. 2. A *successful recovery* occurs when the policy selects a candidate message that matches the original; otherwise, we either cause a *forced panic* or recovery fails by accidentally inducing an MCE. Variability in the reported results is negligible over many millions of individual experiments.

Note that the *absolute* error magnitudes for DUEs and SDECC's impact on *overall* reliability should not be compared directly between codes with distinct $[n, k, d_{min}]_q$ (e.g., a double-bit error for SECDED is very different from a double-chip DUE for ChipKill). Rather, we are concerned with the *relative* fraction of DUEs that can be saved using SDECC for a given ECC code. For evaluations that include second-tier hashes we assume there is no error in the hash itself, as explained earlier.

7.2 Comparison of Data Recovery Policies

We first compare the *raw* successful recovery rates of six different policies for three ECCs *without including any forced panics nor any second-tier hash*. Thus any un-successful recovery here is an MCE. The raw success rate averaged over the SPEC CPU2006 suite for each policy is shown for three ECC constructions in Fig. 6. The depicted baseline represents the average probability $\overline{P_G}$ that we randomly select the original codeword out of a list of candidates for all possible DUEs.

The alternative policies under consideration are the following. *Hamming* chooses the candidate that minimizes the average binary Hamming distance to the neighboring words in the cacheline. *DBX* chooses the candidate that maximizes 0/1-run lengths in the output of the DBX transform [30] of the cacheline. *Longest-Run*, is inspired by frequent pattern compression (FPC) [2] and chooses the candidate message with the longest run of 0/1 in the cacheline. *Delta* is inspired by frequent value compression (FVC) [73] and chooses the candidate message that minimizes the sum of squared integer deltas to the other words in the cacheline.

Our *Entropy-Z* policy variants recovered the most DUEs overall. Of these three, *Entropy-8* ($Z = 8$) performed better than $Z = 4$ and $Z = 16$. The 8-bit entropy symbol size performs best because its alphabet size ($2^8 = 256$ values) matches well with the number of entropy symbols per 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.

Table 4: Percent Breakdown of SDECC *Entropy-8* Policy without Hashes (S = success, P = forced panic, M = MCE)

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

The other four policies all significantly under-performed our *Entropy-Z* variants, with the exception of *Hamming*. It performs nearly as well as the *Entropy-4* policy for integer workloads but fails on many low-entropy cases that have low Hamming distances.

Because *Entropy-8* performed the best for all benchmarks and for all ECC constructions, we exclusively use it in all remaining evaluations.

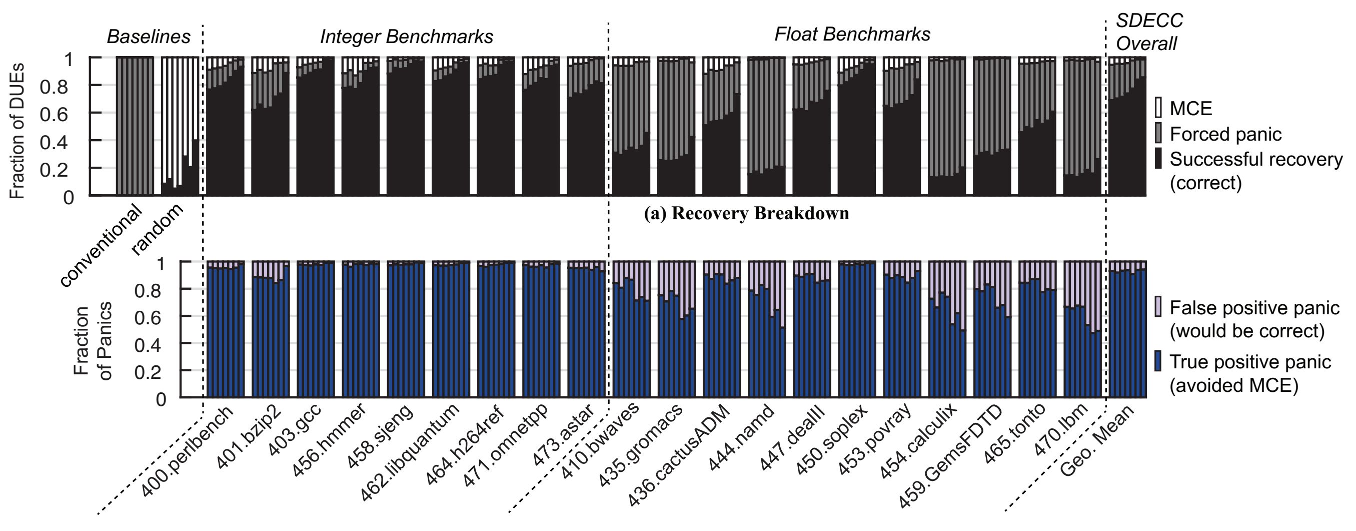
7.3 Recovery Breakdown Without Hashes

Having established *Entropy-8* as the best recovery policy, we now consider the impact of its forced panics on the the successful recovery rate and the MCE rate. Again, we evaluate SDECC for each ECC using its conventional form, without any second-tier hashes to help prune the lists of candidates.

The overall results with forced panics *taken* (main results, gray cell shading) and *not taken* are shown in Table 4. The results for *Entropy-8* on three codes shown earlier in Fig. 6 are repeated in this table for comparison (i.e., the corresponding success rates when panics are not taken). There are two baseline DUE recovery policies: *conventional* (always panic for every DUE) and *random* (choose a candidate randomly, i.e., $\overline{P_G}$).

We observe that when panics are taken the MCE rate drops significantly by a factor of up to 7.3× without significantly reducing the success rate. This indicates that our `PanicThreshold` mechanism appropriately judges when we are unlikely to correctly recover the original information.

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



(a) Recovery Breakdown

(b) Forced Panic Breakdown

Figure 7: Detailed breakdown of DUE recovery results when forced panics are taken and no hashes are used. Results are shown for all seven ECC constructions, listed left to right within each cluster: $[39,32,4]_2$ **Hsiao SECDED –** $[39,32,4]_2$ **Davydov SECDED –** $[72,64,4]_2$ **Hsiao SECDED –** $[72,64,4]_2$ **Davydov SECDED –** $[45,32,6]_2$ **DECTED –** $[79,64,6]_2$ **DECTED –** $[36,32,4]_{16}$ **SSCDSD ChipKill-Correct.**

We examine the breakdown between successes, panics, and MCEs in more detail. Fig. 7 depicts the DUE recovery breakdowns for each ECC construction and SPEC CPU2006 benchmark when forced panics are taken. Fig. 7(a) shows the fraction of DUEs that result in success (black), panics (gray), and MCEs (white). For clarity, the two baselines from Table 4 are repeated on the left and the same panic taken results from the table are repeated on the right (SDECC Overall). Fig. 7(b) further breaks down the forced panics (gray from Fig. 7(a)) into a fraction that are *false positive* (light purple, and would have otherwise been correct) and others that are *true positive* (dark blue, and managed to avoid an MCE). Each cluster of seven stacked bars corresponds to the seven ECC constructions.

We achieve much lower MCE rates than the *random* baseline yet also panic much less often than the *conventional* baseline for all benchmarks, as shown by Fig. 7(a). Our policy performs best on integer benchmarks due to their lower average intra-cacheline entropy. For certain floating-point benchmarks, however, there are many forced panics because they frequently have high data entropy above `PanicThreshold` (e.g., as seen earlier with `444.namd` in Fig. 5b). A `PanicThreshold` of 4.5 bits for these cases errs on the side of caution as indicated by the false positive panic rate, which can be up to 50%. Without more side information, for high-entropy benchmarks, we believe it would be difficult for any alternative policy to frequently recover the original information with a low MCE rate and few false positive panics.

With almost no hardware overheads, SDECC used with SSCDSD ChipKill can recover correctly from up to 85.7% of double-chip DUEs while eliminating 87.2% of would-be panics; this could improve system availability considerably. However, SDECC with ChipKill introduces a 1% risk of converting a DUE to an MCE. Without further action taken to mitigate MCEs, this small risk may be unacceptable when application correctness is of paramount importance.

7.4 Recovery with Hashes

Table 5: Prct. Breakdown of SDECC *Entropy-8* Policy with Hashes (S = success, P = panic, M = MCE)

checksum size	panics taken			panics not taken			*random* baseline		
	S	P	M	S	P	M	S	P	M
***conv.* baseline**	-	100	-						
			$[72,64,4]_2$ **Hsiao – 2-bit DUEs**						
none	71.6	23.7	4.7	75.3	-	24.7	5.0	-	95.0
4-bit	87.8	11.4	0.8	93.7	-	6.3	28.8	-	71.2
8-bit	98.56	1.36	0.08	99.4	-	0.6	86.6	-	13.3
			$[36,32,4]_{16}$ **SSCDSD ChipKill-Correct – 2-chip DUEs**						
none	85.7	12.8	1.5	91.5	-	8.5	39.9	-	60.1
4-bit	98.05	1.86	0.09	99.2	-	0.8	77.0	-	23.0
8-bit	99.940	0.058	0.002	99.98	-	0.02	98.1	-	1.9
16-bit	99.9999	9e-5	0*	100*	-	0*	99.992	-	0.008

**out of 20 million DUE trials*

The second-tier hash can dramatically reduce the SDECC panic and MCE rates by pruning the list of candidate messages before applying the recovery policy.

The recovery breakdowns for second-tier hashes per cacheline on overall MCE rates using $[72,64,4]_2$ Hsiao SECDED and $[36,32,4]_{16}$ SSCDSD ChipKill-Correct ECCs are shown in Table 5. The results for the non-hash cases are repeated from Table 4 for comparison. We do not include 16-bit hashes for the SECDED code because they are unsupported in our architecture.

The results show that even a small 4-bit hash added to every cacheline can reduce the induced MCE rate by up to substantial 16.6×. When high reliability is required, we suggest to use SDECC with a hash of at least 8 bits. Using SDECC with ChipKill and an 8-bit hash, we successfully recovered from 99.940% of double-chip DUEs; with a 16-bit hash, no MCE occurred at all in 20 million trials.

SDECC with hashes can recover from nearly 100% of double-chip DUEs with 4× lower storage overhead than a pure DSC ECC solution and with no common-case performance or notable energy overheads.

8. APPROXIMATION EVALUATION

We have found that SDECC *without* second-tier hashes can still recover a large fractions of DUEs and requires almost no hardware changes. There are many approximate computing applications where some degree of output error is tolerable, but errors should be controlled as much as possible without adding too much overhead. We briefly study the effect of SDECC on application output quality using SECDED (without hash) due to its low latency, area, performance, and energy overheads which are well suited for approximate applications and cost-sensitive systems.

8.1 Methodology

We built AxBench [74] for RV64G in a similar fashion to Sec. 7.1 although we could not run `kmeans` with the proxy kernel successfully. We use our modified version of Spike to run each benchmark to completion 1000 times. For each run, using the $[72,64,4]_2$ Hsiao SECDED code, we inject one DUE on a random demand memory read, emulate the candidate message computation and *Entropy-8* recovery policy, and observe the effects on program behavior. No hashes are used.

8.2 Impact on Output Quality

The percent breakdown of attempted DUE recoveries (success, forced panic, or total MCEs) for each benchmark is shown in the top part of Table 6. The bottom part of the table breaks down the MCE total further with respect to all DUEs injected. For normal program termination, output quality is judged using application-specific metrics defined by AxBench. Consistent with AxBench, we define a tolerable output to be within 10% of the "golden" result [74]. Successes and *benign* MCEs both cause 0% output error, while *tolerable NSDCs* are MCEs that result in $0\% <$ output error $< 10\%$. *Intolerable NSDCs* result in output error $\geq 10\%$. For abnormal program terminations, we characterize the cause: intentional forced panic caused by our policy, or unintentional *crashes* and *hangs* caused by induced MCEs.

For most AxBench benchmarks, our *Entropy-8* recovery policy results in similar success, forced panic, and total MCE rates to our findings with the traces from the SPEC CPU2006 suite. The most challenging case here is `fft`, which has roughly $2\times$ the MCE rate and $3\times$ to $4\times$ the forced panic rate of the other five benchmarks; this is because the program's inputs and computations tend to produce high-entropy data.

The MCE breakdown demonstrates that intolerable NSDCs induced by SDECC are uncommon. Most induced MCEs are actually benign; this agrees with prior work on SDCs [17, 33, 55]. We find that a significant fraction of MCEs cause unintended crashes where the final outcome is no worse than the conventional baseline that always panics for every DUE. A small fraction of the MCEs result in measurable output error, but in the majority of cases, even these produce *tolerable NSDCs* within our 10% output quality window. In the worst case, 1.0% of attempted DUE recoveries result in intolerable NSDCs, while in the best case, it is just 0.1%. Thus, SDECC can be a useful low-cost aid to improve availability and reliability of approximate computing systems with minimal overheads.

9. DISCUSSION

We briefly estimate the system-level availability benefits of SDECC and discuss the alternative use of stronger codes.

Table 6: Prct. Breakdown of Output Quality using *Entropy-8* Policy and Hsiao $[72,64,4]_2$ SECDED Code without Hash

	blackscholes	fft	inversek2j	jmeint	jpeg	sobel
Success	83.8	49.5	82.9	90.4	92.4	90.8
Forced Panic	9.6	38.6	11.4	4.9	4.6	6.0
MCE Total	6.4	11.8	5.5	4.5	2.8	3.1
Breakdown of MCE Total						
Benign	4.8	6.5	4.2	3.2	1.5	2.5
Crash	0.5	0.9	0.2	0.8	0.6	0.5
Hang	0.4	0.0	0.0	0.0	0.0	0.0
Tol. NSDC	0.5	3.3	0.6	0.1	0.4	0.0
Intol. NSDC	0.1	1.0	0.4	0.4	0.3	0.1

9.1 System-Level Benefits

We project the impact of SDECC on a typical supercomputer workload. Consider a hypothetical warehouse-scale computer that is similar to Blue Waters with 22640 compute nodes, where each has 64 GB of DRAM protected using the $[36,32,4]_{16}$ SSCDSD ChipKill-Correct code, and the memory double-chip DUE rate is 15.98 FIT/GB [38]. We use Tiwari's checkpoint model [64]. Suppose an application nominally runs for 500 hours on the system, and that checkpoints take 30 minutes to save or restore and are taken according to the estimated optimal checkpoint interval. We assume that panics caused by memory DUEs are the only cause of failure that warrants checkpointing, and again that hashes in error.

The projected results for the baseline system and SDECC both with and without hashes is shown in Table 7. We find that SDECC alone can deliver a substantial 12% speedup of the application even if no hashes are used. However, an induced MCE is expected to occur once every 2900 hours, or around once in every 5.5 runs of the application. If we use an 8-bit hash can substantially reduce the optimal checkpoint interval to deliver a 18.1% speedup, with an induced MCE occurring only once every 2.2 million hours. A 16-bit hash could obviate the need to checkpoint the application entirely (the expected SDECC forced panic rate is just 0.9 ppm). Thus, we believe our approach could substantially improve the reliability and availability of a supercomputer when memory errors are a significant source of failures.

9.2 SDECC vs. Stronger Codes

One cannot achieve 100% DUE recovery rates without using considerably stronger ECCs or larger second-tier hashes, both of which are impractical. For instance, a SECDED code could be either be upgraded to a DEC code (estimated $2\times$ parity storage, $2\times$ latency, and $14\times$ area overhead vs. SECDED [47]). Alternatively, a $[77,64,5]_2$ 4-error-detect checksum construction could be used, but it is not supported by our architecture because it needs 13 second-tier checksum bits per cacheline (only 8 checksum/hash bits can be supported). For ChipKill, we would need either a Double ChipKill-Correct construction (estimated $2\times$ parity storage, $2\times$ bandwidth overheads vs. SSCDSD [34, 77]) or a significantly more complex $[38,32,5]_{16}$ 4-symbol-detect checksum construction, which needs 24 extra checksum bits per cacheline ($1.5\times$ to $3\times$ more than proposed for our hashes).

10. RELATED WORK

Resilient memory architecture. Recently, the community has become concerned about worsening memory reliability, which can have profound implications for large-scale systems [5, 11,

Table 7: Projected Benefits of SDECC for a Supercomputing Application using $[36,32,4]_{16}$ SSCDSD ChipKill-Correct ECC

scheme/hash size	opt. chkpt. intvl. [64]	speedup	util.	MTT ind. MCE
Baseline	6.6 hours	-	84.4%	N/A
SDECC/none	18.4 hours	12.0%	94.5%	2.9 Khours
SDECC/4-bit	48.2 hours	16.1%	98.0%	48.0 Khours
SDECC/8-bit	272.9 hours	18.1%	99.7%	2.2 Mhours
SDECC/16-bit	N/A	18.5%	100%	N/A

19, 22, 35, 39, 49, 56, 58, 60, 61, 62, 66]. Problems with memory resiliency can largely be attributed to manufacturing process variations [6, 8, 13]. Accordingly, researchers have generally focused on lowering the overhead of strong ECC implementations [3, 4, 14, 23, 24, 28, 29, 31, 32, 34, 46, 53, 65, 70, 71, 76, 77] and dealing with hard faults [3, 4, 15, 45, 46, 52, 57, 71, 78]. Three state-of-the-art works relate closely to our contributions. Bamboo ECC [31], Error Pattern Transformation [15], and XED [46] each propose ways to reduce the occurrence of DUEs without necessarily increasing code strength, but unlike this work, they do not consider how to handle DUEs when they do occur.

List decoding. SDECC is related to the theory of list decoding [16, 20, 63, 72]. Unfortunately, the list decoding theory has not produced a low-cost and computationally-efficient decoder suitable for use with memory. The distinction of our approach is that it retains all the advantages of conventional ECCs yet it can also produce a list of candidate codewords whenever a DUE occurs. We also describe a novel methodology to choose the best candidate codeword given SI about memory contents.

11. CONCLUSION

SDECC is a new approach to improve the resiliency of systems by recovering from a large fraction of memory DUEs. SDECC is based on the fundamental observation that when a DUE occurs, there are a small number of candidate codewords that can be computed, wherein one is practically guaranteed to be correct. Policies that leverage SI about memory contents can be used to recover successfully from many DUEs when they occur, while incurring negligible overheads in the common cases. Directions for future work include adaptive software and ECC support for memory fault models and development of software mechanisms that can eventually verify the correctness of SDECC recovery.

Acknowledgment

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.

12. REFERENCES

[1] "Intel 64 and IA-32 Architectures Software Developer Manuals." [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

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

[3] A. R. Alameldeen, Z. Chishti, C. Wilkerson, W. Wu, and S.-L. Lu, "Adaptive Cache Design to Enable Reliable Low-Voltage Operation," *IEEE Transactions on Computers (TC)*, vol. 60, no. 1, pp. 50–63, 2011.

[4] A. R. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu, "Energy-Efficient Cache Design Using Variable-Strength Error-Correcting Codes," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011.

[5] E. Baseman, N. DeBardeleben, K. Ferreira, S. Levy, S. Raasch, V. Sridharan, T. Siddiqua, and Q. Guan, "Improving DRAM Fault Characterization Through Machine Learning," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2016.

[6] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[7] S. Borkar and A. A. Chien, "The Future of Microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[8] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter Variations and Impact on Circuits and Microarchitecture," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2003.

[9] R. Bose and D. Ray-Chaudhuri, "On a Class of Error Correcting Binary Group Codes," *Information and Control*, vol. 3, no. 1, pp. 68–79, 1960.

[10] A. Davydov and L. Tombak, "An Alternative to the Hamming Code in the Class of SEC-DED Codes in Semiconductor Memory," *IEEE Transactions on Information Theory*, vol. 37, no. 3, pp. 897–902, 1991.

[11] N. DeBardeleben, S. Blanchard, V. Sridharan, S. Gurumurthi, J. Stearley, K. B. Ferreira, and J. Shalf, "Extra Bits on SRAM and DRAM Errors - More Data from the Field," in *Workshop on Silicon Errors in Logic – System Effects (SELSE)*, 2014.

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

[13] N. Dutt, P. Gupta, A. Nicolau, A. BanaiyanMofrad, M. Gottscho, and M. Shoushtari, "Multi-Layer Memory Resiliency," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2014.

[14] H. Duwe, X. Jian, and R. Kumar, "Correction Prediction: Reducing Error Correction Latency for On-Chip Memories," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[15] H. Duwe, X. Jian, D. Petrisko, and R. Kumar, "Rescuing Uncorrectable Fault Patterns in On-Chip Memories through Error Pattern Transformation," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[16] P. Elias, "List Decoding for Noisy Channels," Massachusetts Institute of Technology (MIT), Tech. Rep., 1957.

[17] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-layer Resilience Analysis," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[18] J. D. Gelas, "The Intel Xeon E5 v4 Review: Testing Broadwell-EP With Demanding Server Workloads," 2016. [Online]. Available: http://www.anandtech.com/show/10158/the-intel-xeon-e5-v4-review

[19] M. Gottscho, M. Shoaib, S. Govindan, B. Sharma, D. Wang, and P. Gupta, "Measuring the Impact of Memory Errors on Application Performance," *IEEE Computer Architecture Letters (CAL)*, 2016.

[20] V. Guruswami, "List Decoding of Error-Correcting Codes," Ph.D. Dissertation, Massachusetts Institute of Technology (MIT), 2001.

[21] M. Y. Hsiao, "A Class of Optimal Minimum Odd-Weight-Column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[22] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[23] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar, "Low-Power, Low-Storage-Overhead Chipkill Correct via Multi-line Error Correction," in *Proceedings of the IEEE International Conference for*

High Performance Computing, Networking, Storage and Analysis (SC), 2013.

[24] X. Jian and R. Kumar, "ECC Parity: A Technique for Efficient Memory Error Resilience for Multi-Channel Memory Systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[25] X. Jian, J. Sartori, H. Duwe, and R. Kumar, "High Performance, Energy Efficient Chipkill Correct Memory with Multidimensional Parity," *IEEE Computer Architecture Letters (CAL)*, vol. 12, no. 2, pp. 39–42, 2013.

[26] X. Jian, V. Sridharan, and R. Kumar, "Parity Helix: Efficient Protection for Single-Dimensional Faults in Multi-Dimensional Memory Systems," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[27] S. Kaneda and E. Fujiwara, "Single Byte Error Correcting – Double Byte Error Detecting Codes for Memory Systems," *IEEE Transactions on Computers (TC)*, vol. C-31, no. 7, pp. 596–602, 1982.

[28] D. W. Kim and M. Erez, "RelaxFault Memory Repair," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[29] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe, "Multi-Bit Error Tolerant Caches Using Two-Dimensional Error Coding," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2007.

[30] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-Plane Compression: Transforming Data for Better Compression in Many-core Architectures," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[31] J. Kim, M. Sullivan, and M. Erez, "Bamboo ECC: Strong, Safe, and Flexible Codes for Reliable Computer Memory," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[32] J. Kim, M. Sullivan, S. Lym, and M. Erez, "All-Inclusive ECC: Thorough End-to-End Protection for Reliable Computer Memory," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[33] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[34] S. Li, D. H. Yoon, K. Chen, and J. Zhao, "MAGE : Adaptive Granularity and ECC for Resilient and Power Efficient Memory Systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[35] X. Li, M. C. Huang, K. Shen, and L. Chu, "A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility," in *USENIX Annual Technical Conference (ATC)*, 2012.

[36] S. Lin and D. J. Costello, *Error Control Coding*. Prentice Hall, 2004.

[37] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[38] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014, pp. 610–621.

[39] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015.

[40] J. S. Miguel, J. Albericio, N. E. Jerger, and A. Jaleel, "The Bunker Cache for Spatio-Value Approximation," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2016.

[41] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2014.

[42] S. Mitra and K. Kim, "X-Compact: An Efficient Response Compaction Technique," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 23, no. 3, pp. 421–432, 2004.

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

[44] S. Mittal and J. Vetter, "A Survey of Architectural Approaches for Data Compression in Cache and Main Memory Systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 5, pp. 1524–1536, 2015.

[45] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "Citadel: Efficiently Protecting Stacked Memory from TSV and Large Granularity Failures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, 2016.

[46] P. J. Nair, V. Sridharan, and M. K. Qureshi, "XED: Exposing On-Die Error Detection Information for Strong Memory Reliability," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[47] R. Naseer and J. Draper, "Parallel Double Error Correcting Code Design to Mitigate Multi-Bit Upsets in SRAMs," *Proceedings of the IEEE European Solid-State Circuits Conference (ESSCIRC)*, 2008.

[48] Q. Nguyen, "RISC-V Tools (GNU Toolchain, ISA Simulator, Tests) – git commit 816a252." [Online]. Available: https://github.com/riscv/riscv-tools

[49] P. Nikolaou, Y. Sazeides, L. Ndreu, and M. Kleanthous, "Modeling the Implications of DRAM Failures and Protection Techniques on Datacenter TCO," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2015.

[50] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2013.

[51] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[52] M. K. Qureshi, "Pay-As-You-Go: Low-Overhead Hard-Error Correction for Phase Change Memories," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2011.

[53] M. K. Qureshi and Z. Chishti, "Operating SECDED-Based Caches at Ultra-Low Voltage with FLAIR," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6575314

[54] A. Rahimi, L. Benini, and R. K. Gupta, "Variability Mitigation in Nanometer CMOS Integrated Systems: A Survey of Techniques From Circuits to Software," *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1410–1448, 2016.

[55] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Characterizing the Impact of Intermittent Hardware Faults on Programs," *IEEE Transactions on Reliability (TR)*, vol. 64, no. 1, pp. 297–310, 2015.

[56] F. Sala, H. Duwe, L. Dolecek, and R. Kumar, "A Unified Framework for Error Correction in On-chip Memories," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2016.

[57] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, not ECC, for Hard Failures in Resistive Memories," *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2010.

[58] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," *Communications of the ACM*, vol. 54, no. 2, pp. 100–107, 2011.

[59] J. C. Smolens, "Fingerprinting: Hash-Based Error Detection in Microprocessors," Ph.D. Dissertation, Carnegie Mellon University, 2008.

[60] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng Shui of Supercomputer Memory: Positional

Effects in DRAM and SRAM Faults," in *Proceedings of the IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[61] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[62] V. Sridharan and D. Liberty, "A Study of DRAM Failures in the Field," in *Proceedings of the IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[63] M. Sudan, "List Decoding: Algorithms and Applications," in *Springer Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2001, pp. 25–41.

[64] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[65] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, "LOT-ECC: LOcalized and Tiered Reliability Mechanisms for Commodity Memory Systems," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2012.

[66] S. Wang, H. C. Hu, H. Zheng, and P. Gupta, "MEMRES: A Fast Memory System Reliability Simulator," *IEEE Transactions on Reliability (TR)*, vol. 65, no. 4, pp. 1783–1797, 2016.

[67] A. Waterman, "RISC-V Proxy Kernel – git commit 85ae17a." [Online]. Available: https://github.com/riscv/riscv-pk/commit/85ae17a

[68] A. Waterman and Y. Lee, "Spike, a RISC-V ISA Simulator – git commit 3bfc00e." [Online]. Available: https://github.com/riscv/riscv-isa-sim

[69] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.0," 2014. [Online]. Available: https://riscv.org

[70] W. Wen, M. Mao, X. Zhu, S. H. Kang, D. Wang, and Y. Chen, "CD-ECC: Content-Dependent Error Correction Codes for Combating Asymmetric Nonvolatile Memory Operation Errors," in *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, 2013.

[71] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, "Reducing Cache Power with Low-Cost, Multi-Bit Error-Correcting Codes," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2010.

[72] J. Wozencraft, "List Decoding," *Quarterly Progress Report*, 1958.

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

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

[75] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "Mitigating the Memory Bottleneck with Approximate Load Value Prediction," *IEEE Design & Test*, vol. 33, no. 1, pp. 32–42, 2016.

[76] D. H. Yoon and M. Erez, "Memory Mapped ECC: Low-Cost Error Protection for Last Level Caches," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2009.

[77] D. H. Yoon and M. Erez, "Virtualized and Flexible ECC for Main Memory," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[78] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, "FREE-p: Protecting Non-Volatile Memory Against both Hard and Soft Errors," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.