Software-Defined Error-Correcting Codes

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

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

***Abstract*—Conventional error-correcting codes (ECCs) and system-level fault-tolerance mechanisms are currently treated as separate abstraction layers. This can reduce the overall efficacy of error detection and correction (EDAC) capabilities, impacting the reliability of memories by causing crashes or silent data corruption. To address this shortcoming, we propose Software-Defined ECC (SWD-ECC), a new class of heuristic techniques to recover from detected but uncorrectable errors (DUEs) in memory. It uses available side information to estimate the original message by first filtering and then ranking the possible candidate codewords for a DUE. SWD-ECC does not incur any hardware or software overheads in the cases where DUEs do not occur.**

As an exemplar for SWD-ECC, we show through offline analysis on SPEC CPU2006 benchmarks how to heuristically recover from 2-bit DUEs in MIPS instruction memory using a common (39,32) single-error-correcting, double-error-detecting (SECDED) code. We first apply coding theory to compute all of the candidate codewords for a given DUE. Second, we filter out the candidates that are not legal MIPS instructions, increasing the chance of successful recovery. Finally, we choose a valid candidate whose logical operation (e.g., add or load) occurs most frequently in the application binary image. Our results show that on average, 34% of all possible 2-bit DUEs in the evaluated set of instructions can be successfully recovered using this heuristic recovery strategy. If a DUE affects the bit fields used for instruction decoding, we are able to recover correctly up to 99% of the time. We believe these results to be a significant achievement compared to an otherwise-guaranteed crash which can be undesirable in many systems and applications. Moreover, there is room for future improvement of this result with more sophisticated uses of side information. We look forward to future work in this area.

I. Introduction

New approaches to improving memory resiliency are necessary. Memories are a primary cause of hardware failures in the field [1], [2], [3], and comprise a significant portion of datacenter cost [4], [5]. Error-correcting codes (ECCs) and system-level fault-tolerance techniques for memories have historically been treated as separate abstractions in the hardware/software stack. When detected but uncorrectable errors (DUEs) occur in memory, crashes or silent data corruptions often follow because the system and ECC algorithm fail to share their available *side information* about the error. A solution that crosses these abstraction layers could bring a significant improvement to system resiliency, which is critically needed in the nanoscale era [6].

We propose a novel class of techniques, which we call *Software-Defined Error-Correcting Codes (SWD-ECCs)*, that cross the abstraction gap between coding theory and hardware/software fault-tolerance techniques. These promise better-than-worst-case error detection and correction (EDAC) capabilities for memory. *The key idea in Software-Defined ECC is to leverage available side information about the underlying message data being stored in memory to heuristically recover from DUEs that exceed the guarantees provided by the ECC code by itself.* This is done by speculating on the correct value of the original message that has been corrupted.

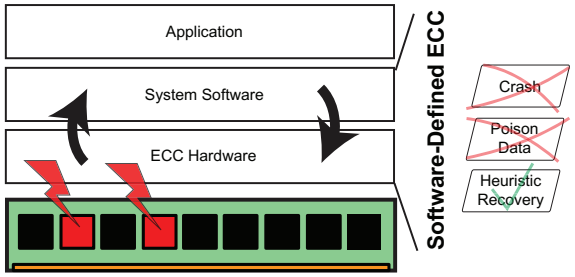


Fig. 1. High-level concept for Software-Defined Error-Correcting Codes (SWD-ECC), which, instead of crashing or poisoning data, heuristically recovers from DUEs that occur in memory words (red lightning bolts) via the collaboration of system software and ECC hardware.

SWD-ECC could be useful in a variety of systems. A large class of applications are naturally error-tolerant and approximation-friendly at the algorithmic level [7]. In these systems, DUEs tend to be more inconvenient than catastrophic, so probabilistic error recovery would likely be acceptable. Even in high-performance systems where correctness is paramount, SWD-ECC could enable faster recovery from a DUE with a reasonable chance of success. If the correctness of the recovery attempt can be eventually verified (e.g., through control flow checks [8] or symptoms of abnormal execution [9]), SWD-ECC could improve performance compared to performing a time-consuming rollback to a system checkpoint and then re-computing state. SWD-ECC could also be useful in real-time systems where missing a deadline is worse than the possibility of incorrect execution. Therefore, we believe that there is room for SWD-ECC as an option alongside crashing, silent data corruption, and state re-computation in response to memory DUEs. The high-level concept for SWD-ECC is shown in Fig. 1.

As an exemplar for SWD-ECC, we show how to heuristically recover from 2-bit DUEs that can occur in a 32-bit MIPS instruction memory that is protected using a common single-error-correcting, double-error-detecting (SECDED) code. This is done by leveraging properties of the ECC code, knowledge of the MIPS ISA, and statistics extracted from the compiled program binaries. We exhaustively study all possible 2-bit errors that can occur for each of the first 100 instructions in five SPEC CPU2006 benchmarks. On average, we are able to successfully recover from 34% of these errors: depending on the need for correctness, we believe this is significantly better than a guaranteed system crash that would occur in conventional systems.

This paper is organized as follows. In Sec. II, we discuss background information and related work on ECC and system-level fault-tolerance techniques for memories. In Sec. III, we describe the fundamental concepts behind SWD-ECC and give several use case examples. We evaluate an exemplar implementation via offline static analysis on SPEC CPU2006

AUTHORS' COPY dated May 3, 2016
Minor changes from the Feb. 27, 2016 version that originally appeared at *SELSE-12: The 12th IEEE Workshop on Silicon Errors in Logic -- System Effects (SELSE)*, Austin, Texas, USA, March 29-30, 2016.

To appear in the Best of SELSE special session at the *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Toulouse, France, June 28-July 1, 2016.

executable code in Sec. IV. Sec. V concludes the paper.

II. Background and Related Work

Reliability mechanisms for memory systems can be broadly classified into fundamental EDAC capabilities built using channel coding theory, and system-level fault-tolerance methods that rely on them. We first review basic concepts and related work for ECC before discussing some relevant aspects of system-level fault-tolerance.

A. Error-Correction Codes

ECC [10] is the fundamental EDAC mechanism that guards against memory errors and is typically implemented in hardware. We consider the common class of systematic linear block codes for a binary symmetric channel (BSC). Any such code permits the notation (n,k) that specifies the length of a *codeword* and an input *message*, respectively, where $n > k$. Encoding is done by multiplying the k-bit message with the binary *generator matrix*. The resulting n-bit codeword is stored in the memory, and includes $r = n-k$ extra *parity-check* bits. A memory read obtains the *received string*, which is multiplied with the binary *parity-check matrix*. This decoding process yields a *syndrome*, which is an r-bit string containing the results for a set of parity-check equations. If the syndrome is 0, then the received string is a codeword and no errors were detected. The message is then extracted from the codeword by discarding the redundant parity bits. Otherwise, one or more bit-errors were found in the received string. To attempt correction, a *syndrome decoding* procedure is used to find the *maximum-likelihood* input codeword. For more information on the theory of ECCs, we refer the reader to [11].

The codes used for memories usually guarantee the correction of up to t-bit errors and the detection of $t+1$-bit errors in an n-bit codeword. The most common form of these codes is the *single-error-correcting double-error-detecting (SECDED)* family [12], which guarantees a minimum Hamming distance of 4 bits between codewords. There are many codes that are more powerful than SECDED and are sometimes used for memories, such as double-error-correcting, triple-error-detecting (DECTED), BCH [13], and ChipKill [14], but they come with much higher bit storage and/or performance overheads than SECDED. In this work, our implementation and evaluation is done using a (39,32) SECDED ECC code.

Many other advanced codes suitable for memory exist in the research literature. Several works have explored source coding and channel coding for fault models other than the BSC by focusing on emerging non-volatile random-access memories (NVMs) [15], [16], [17], [18] and storage-class flash memory [19], [20], [21], [22], [23]. ECCs that are suitable to approximate computing, e.g., Variable-Strength ECC [24] have been proposed. Others have advocated for using error avoidance techniques that could be used instead of ECC for coping with hard faults that can be characterized a priori [25], [26]. However, none of these works have explored how to heuristically recover from DUEs that can still occur in any type of memory, whether they be SRAM, DRAM, or a form of NVM. Their approaches are generally orthogonal to Software-Defined ECC, and we believe that many could be combined with the ideas in this work.

B. System-Level Fault-Tolerance

System-level fault-tolerance techniques are often used in addition to the fundamental EDAC mechanisms provided by the ECC hardware [27]. Checkpointing [28], mirroring, and sparing [29] are costly techniques commonly used in mainframes, supercomputers, and/or mission-critical systems. Checkpointing can periodically save the state of the entire system or application. In the case of a memory DUE, the system can be rolled back to the last checkpoint, hopefully avoiding catastrophic crashes or silent data corruption. There are also alternatives to memory ECC that can be used for error detection and typically rely on checkpointing as a correction/recovery mechanism [8], [9]. *Reliability management* techniques such as memory page retirement [30] or scrubbing [31] are opportunistic and incur little or no hardware cost. They allow the system performance to degrade gracefully from failures without high performance overheads, but often lack firm reliability guarantees because they can only speculate on the occurrence of future DUEs, not recover from existing ones. In contrast, SWD-ECC speculates on the correct outcome of a DUE given that it has already occurred.

All of the above system-level fault-tolerance techniques are complementary to the SWD-ECC concepts described in this paper and can be combined for improved system-level resilience against memory DUEs. For further information, we refer the reader to [32] for an excellent survey on recent work studying the reliability of computer systems.

III. Software-Defined ECC Concept

Software-Defined ECC (SWD-ECC) is a new approach in the field of memory resiliency that intersects both coding theory and system design to enable better-than-worst-case and opportunistic recovery from DUEs. We discuss how SWD-ECC can address the limitations of abstracted ECC and fault tolerance layers, a novel DUE heuristic recovery procedure, and usage considerations.

A. Problems with the Existing Abstraction Stack

Conventional system-level fault-tolerance techniques ignore properties of the ECC code on which they rely. The ECC hardware is usually treated as a black box that simply reports whether a memory word had no error, a *corrected error (CE)*, or a DUE. Typically, little information about a DUE is given, perhaps other than the memory address of the corrupted word. Upon notification of a DUE, most systems trigger a kernel panic, while high-end systems might roll-back to a checkpointed state or poison the corrupted memory word to contain the effects of the error. This behavior may be undesirable in scenarios where forward progress must be made in a timely manner, or when the application is naturally error-tolerant.

Similarly, ECC codes that are used today are agnostic to patterns in the underlying message contents that arise from the behavior of the system and applications. The most common assumption is that all bit errors are equally likely – i.e., the memory is modeled as a BSC – and that all messages are equally likely to be encoded. This simplifies the maximum-likelihood decoding procedure, which essentially chooses the codeword that has the shortest Hamming distance to the received string. Unfortunately, the assumption often does not hold in reality, making this decoding procedure sub-optimal.

SWD-ECC addresses the above shortcomings of separate system-level fault-tolerance and ECC abstraction layers by using available *side information* about the source and/or the channel. Side information arises through the cooperation of hardware and software, and generally could be comprised

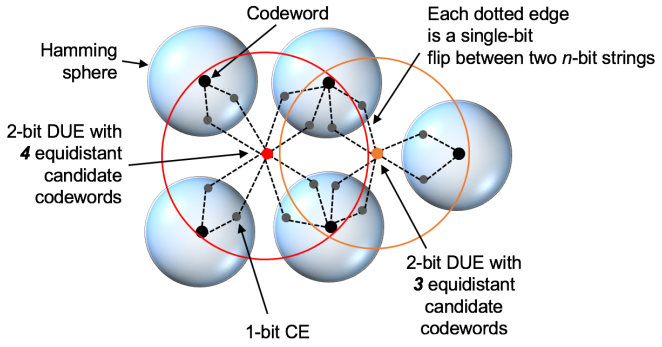


Fig. 2. Conceptual illustration of a two-dimensional partial slice of the n-dimensional binary space for a general (n,k) SECDED code. Each point represents an n-bit string, all of which are either codewords, CEs, or DUEs. Not all DUEs (red and orange points) have the same number of equidistant candidate codewords, as shown by their 2-bit radii (red and orange circles).

of *fault models* for the channel and/or *message contents* for the source. In this paper, we only consider the latter type of information.

B. Heuristic Recovery from DUEs

The key idea in SWD-ECC is that side information can be used to *heuristically recover* from DUEs by trying to correctly estimate the original uncorrupted message. We now outline the major requirements for SWD-ECC functionality along with examples. Throughout, we assume the use of SECDED codes, but the concepts can extend to others as well.

Candidate codewords. *The first requirement of SWD-ECC is the ability to find all possible* candidate codewords*, i.e., the specific codewords that, when corrupted with any double-bit error, result in the known-erroneous received string.* Consider Fig. 2, which shows a two-dimensional partial slice of a hypothetical (n,k) SECDED code. The larger black circles represent codewords (no errors), and the smaller gray circles represent strings with Hamming distances of 1 bit from codewords, making them inside the Hamming spheres that are centered on the codewords (CEs). The red and orange pentagons represent strings that have a Hamming distance of at least two bits from all codewords, making them lie outside the Hamming spheres (DUEs). For each DUE, there are several equidistant candidate codewords, one of which corresponds to the original message. In the figure, the equidistant codewords for each DUE fall inside the red and orange circles which have a radius of two bits. In a conventional SECDED decoder, because all messages are assumed to be equally likely, the decoder cannot differentiate between the candidate codewords. It gives up and notifies the system of a DUE. SWD-ECC, however, attempts to choose a candidate codeword that has the best chance of being the correct answer. It firsts computes an exact list of candidate codewords for a DUE. This procedure is similar to the information-theoretic concept of *list decoding* [33], [34]; the primary difference is that we only compute a list upon registering a DUE, instead of computing them on every memory access.

If we assume DUEs only occur as a result of a double-bit flip, one can compute the list of candidate codewords for a SECDED code by iteratively flipping each of n bits at a time in the received string. For each such *trial flip*, a *modified string* is obtained and then input to the SECDED decoder. Many of these modified strings will still be registered as DUEs by the SECDED hardware (making them 3-bit DUEs with respect to the correct answer), but some will land inside a nearby Hamming sphere. The latter group of modified strings will be interpreted as 1-bit CEs and are decoded to the set of candidate codewords.

At this stage, with a list of candidate codewords, the SWD-ECC problem is reduced to choosing the correct answer. Interestingly, the number of candidates (and the chance of recovery) depends on the exact positions of the two bits in error. This arises in SECDED codes that are based on *truncated* Hamming codes, such as the common (39,32) and (72,64) codes used in memories. In these codes, there exist some bit strings that have a Hamming distance of two bits from a DUE and are themselves DUEs instead of codewords. To visualize this, refer again to Fig. 2; the bit string indicated by the red pentagon has four equidistant candidate codewords, while another bit string (orange pentagon) has only three candidates.[1]

Randomly choosing a candidate codeword for recovery is not a good solution. Thus, *the second requirement of SWD-ECC, after the ability to find all candidate codewords, is the use of available side information to select the best candidate.* For simplicity, in this paper, we exclusively use message content as the side information, and ignore fault models other than the BSC. We now discuss some possibilities for side information that can be used in data and instruction memories protected by SECDED ECC.

Side information for data memory. Consider a system where SWD-ECC is used to protect data memory. If it is known that a particular memory location is part of an array of unsigned integers of low magnitude – perhaps via program debug information with the help of the operating system – then SWD-ECC's heuristic recovery scheme can rule out any candidate codewords whose messages have 1s in the most-significant bit positions. Similarly, if the location is known to contain a memory address (i.e., it is a pointer), then SWD-ECC can reasonably eliminate all candidate codewords whose messages would point outside the virtual address space allocated to the application.

In the absence of high-level semantic program information such as the examples using data types just described, heuristic recovery can still be done using program statistics. Several works have shown that the data words in a cache line tend to be highly correlated [35], [36], [37]. Candidate codewords whose corresponding messages are distant (by any software-defined metric) from neighboring messages in the cache line could be eliminated for consideration by SWD-ECC. For instance, if the data types of words in the cache line are known, then the integral magnitude can be used as a distance metric. Even if the data types cannot be inferred, a simple majority-vote procedure on groups of bits could be used to aid recovery.

We leave further study of the above ideas to future work and now focus on instruction memory.

Side information for instruction memory. Now consider a scenario where we wish to heuristically recover from a DUE in instruction memory. The likelihood of recovery can be greatly improved by leveraging the ISA itself. In general, instruction sets are not fully populated: some bit strings indicate reserved or *illegal instructions*. For instance, in most RISC ISAs, instructions have a fixed length and a dedicated region for storing the *opcode*. This field describes the basic logical behavior of the instruction as well as the format of

[1]Note that the figure does not accurately illustrate the non-perfect nature of the code space; many dimensions are needed to draw this faithfully.

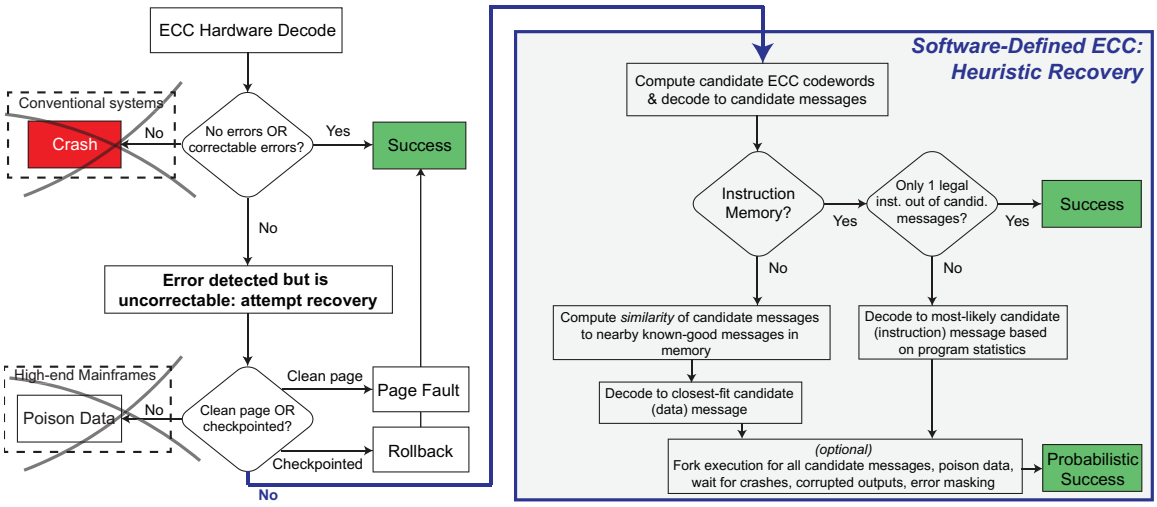


Fig. 3. Proposed system-level flow for Software-Defined ECC.

remaining bit fields. In our manual analysis of the MIPS1 instruction set [38], we found that only 41 out of 64 possible 6-bit opcode values are used. The remaining opcodes constitute illegal instructions. For the register-type (R-type) instructions, which have opcode 0x00, an additional 6-bit *funct* field is used to further specify the operation. Out of these, only 37 out of 64 values are legal. If the opcode is 0x11, indicating a floating-point operation, then a 5-bit *fmt* field is used, of which only three out of 32 values are legal. We use facts such as these to eliminate any candidate codewords/messages that constitute illegal instructions, improving chances of a successful recovery.

Recovery from DUEs in instruction memory can also be aided by program statistics. Programs tend to use a few instructions very frequently, while many specialized operations are rarely used, if at all. A list of valid candidate codewords that correspond to legal instructions can be ranked by the relative frequency of opcode appearance in the program image. The recovery target could simply be the candidate that appears most often overall.

C. Use Models

SWD-ECC can heuristically recover from DUEs without changing the ECC code rate or implementation, but this can come at the expense of performance. In the common cases when no errors or only CEs occur, decoding complexity and memory system performance remain unaffected. When a (rare) DUE actually does occur, however, software is used to assist the ECC hardware in recovery, hurting performance. However, a chance at correct recovery from rare errors may be worth a temporary loss in performance, particularly if the performance overheads of alternative techniques such as rolling back to a checkpoint are large.

We propose a high-level SWD-ECC methodology that is shown in Fig. 3. Upon a read from memory, the ECC hardware checks for errors: if there are no errors or only a CE, then ECC is successful, whereas if there is a DUE, the system attempts to recover (instead of crashing, like many conventional systems). If the memory page in question is clean, a page fault can be used to recover from the DUE; if a recent checkpoint exists and the performance penalty is modest, a rollback can be triggered. If none of these are viable options, SWD-ECC takes over with heuristic recovery instead of poisoning the data as might be done in high-end systems [31]. We use the heuristic recovery procedures described above depending on whether the DUE occurred in data or instruction memory.

Chance of incorrect recovery. There is always a possibility that the selected recovery target is incorrect. Depending on the system scenario, it may be desirable to speculatively fork execution of the process impacted by the DUE. Each fork could receive a unique and poisoned candidate codeword to use in its version of execution. Parallel execution of each fork would continue until one of the following conditions occur: *(i)* crashes, assertion failures, or other symptoms of abnormal execution [9] occur on all but one fork; *(ii)* only one fork contains non-poisoned state, i.e., the others logically masked the error; *(iii)* multiple forks reach a milestone with identical states, allowing them to be joined and assumed correct; *(iv)* all forks' outputs are measurably incorrect except one; or *(v)* multiple forks survive to a point where state must be made permanent, in which case it may be best to forfeit progress of all forks, and roll back to the last good checkpoint or restart execution of the workload from the beginning.

Compression as an alternative to SWD-ECC. An alternative approach to SWD-ECC might instead use lossless compression on the message contents (source coding) [35], [36], [37], so that they have higher entropy before being channel coded with ECC. The tradeoffs between compression and SWD-ECC are not yet clear; we leave this to future work.

IV. Evaluation

As an exemplar for the class of SWD-ECC approaches, we evaluate our proposed method for heuristically recovering from DUEs that affect a 32-bit MIPS instruction memory through offline analysis on SPEC CPU2006 benchmarks.

A. Experimental Setup

We used a common (39,32) SECDED code whose exact generator and parity-check matrices can be found in [39]. The only side information we considered is the MIPS1 ISA itself [38] and the relative frequencies that instruction *operations* appear in a compiled program image. We assumed a BSC fault model, i.e., all possible 2-bit flips in a codeword are equally likely to occur.

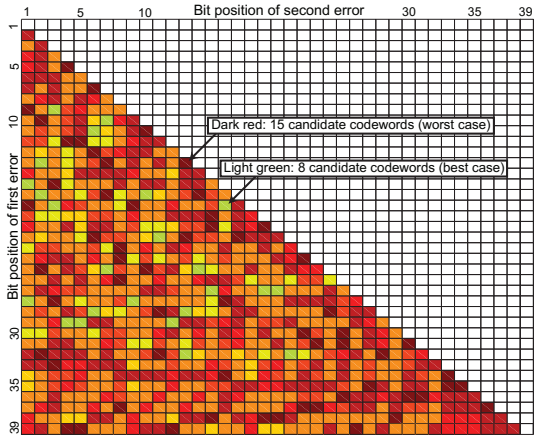


Fig. 4. This heatmap depicts the number of candidate codewords as a function of the two error bit locations in a DUE using our (39,32) SECDED ECC code. Hotter colors indicate more candidates, which range from 8 to 15. There are exactly 741 possible 2-bit error patterns.

We used *bzip2*, *h264ref*, *mcf*, *perlbench*, and *povray* benchmarks from the SPEC CPU2006 suite. Each benchmark was cross-compiled for 32-bit MIPS1 using *gcc*. We fully disassembled each binary image using the *readelf* tool, and used a MATLAB script to compute statistics on the relative frequencies that each operation (instruction mneumonic, e.g., *add*, *lw*, *beq*, etc.) appears in each program image.

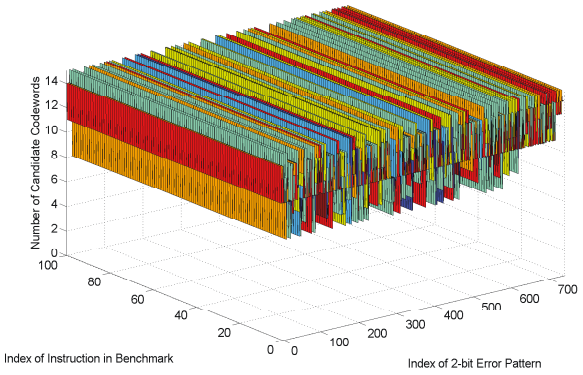
We examined all possible $\binom{39}{2} = 741$ 2-bit error vectors on the first 100 instructions from each program's *.text* section. For instance, the error vector 0 is $1100...0000_2$, the second is $1010...0000_2$, and so on, until error vector 740, which is $0000...0011_2$. Each error vector was *xor*-ed with the 39-bit SECDED-encoded instruction that was under consideration, which provided the received string. These were the inputs to the SWD-ECC heuristic recovery procedure.

For each pair of instruction and error vector, we computed all possible candidate codewords. Next, the *candidate messages* were filtered for legality when interpreted as MIPS instructions. To achieve this, we isolated and extracted the C++ code that implements the MIPS instruction decoder from the gem5 simulator [40]. Our version of this ISA decoder simply indicates whether a 32-bit binary value is a legal or illegal MIPS instruction, and if it is legal, the type of operation is reported. Finally, the remaining *valid messages* were ranked by the relative frequency that their mneumonics (e.g., *add*, *lw*, *beq*, etc.) appear in the entire program image.

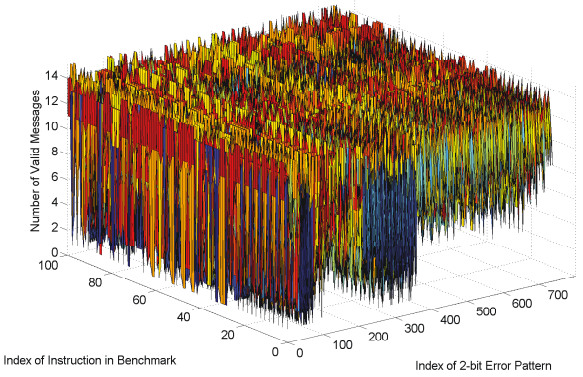
B. Results

We analyze the properties of our selected (39,32) SECDED ECC code before evaluating the efficacy of our heuristic recovery scheme for MIPS instruction memory using a basic *filtering-only* approach and our final and improved *filtering-and-ranking* approach.

Properties of SECDED code. The number of candidate codewords for a DUE as a function of the exact locations of the two bits in error are depicted in Fig. 4. The results are independent of the input message because the code is linear. At worst, there are 15 candidate codewords for a double-bit error, and at best, there are eight candidates; on average, there are approximately 12 possibilities. The heatmap indicates that some 2-bit DUEs have almost twice the baseline likelihood of successful recovery compared to others.



(a) Candidate Codewords/Messages



(b) Valid Messages (Filtered Subset of Candidate Messages)

Fig. 5. Number of possible recovery targets as a function of the two bitwise error locations and the instruction index in the *mcf* benchmark. A lower number is better, as it increases the probability of successful heuristic recovery from a DUE. The number of candidate codewords in Fig. 5(a) is independent of the message (instruction) due to the linearity of the ECC code, and is therefore the same for all applications. The filtered subset, shown in Fig. 5(b), is used in both of the *filtering-only* and *filtering-and-ranking* recovery strategies. Here, the number of valid messages indeed depends on the original message (instruction).

***Filtering-only* recovery strategy.** We first consider whether the input message has any effect on the number of candidate codewords for all 741 error patterns by filtering those that correspond to illegal instruction messages. Fig. 5(a) shows the number of candidate codewords as a function of the unique 2-bit error pattern and the original message, which is one of the first 100 instructions from the *mcf* benchmark. We can see that the particular encoded instruction message has no effect because the ECC code is linear. However, once the candidate messages are filtered for instruction legality, as shown in Fig. 5(b), the number of valid messages becomes dependent on the original instruction. On average, the number of valid messages decreases by approximately two compared to the number of candidate messages. In the best cases, the number is reduced to just one possibility: without any additional information, the probability of successful recovery from these 2-bit DUEs is already 100%! The best-filtered candidate messages have errors in the *opcode*, *funct*, and/or *fmt* instruction fields. Errors that occur in the *register address*, *memory address*, or *immediate* fields do not filter the candidate codewords as effectively. This is because in the MIPS ISA, these fields can legally be any value.

We evaluated the efficacy of choosing a random decode target out of the full set of candidate messages as well as the filtered set of valid messages. Fig. 6 depicts the fraction

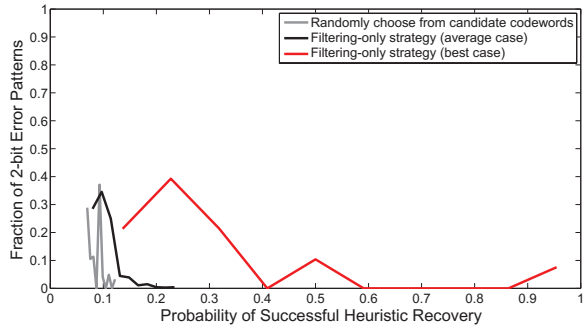


Fig. 6. Baseline histogram of successful rates for heuristic recovery over all 741 possible 2-bit DUE error locations for MIPS instructions in *bzip2* using a (39,32) SECDED code using the *filtering-only* strategy.

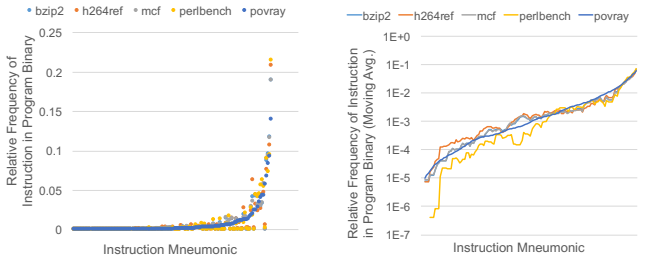
of 2-bit error patterns that achieved each rate of successful heuristic recovery for the *bzip2* benchmark. On average, over the first 100 instructions, the filtering-only method mildly improves the probability of successful recovery (black vs. gray lines). However, for the best cases out of the 100 *bzip2* instructions, the improvement is stark (red line) with a range of ≈15% to ≈95% chance of recovery. We improve these results further by ranking the filtered valid codewords/messages.

***Filtering-and-ranking* recovery strategy.** After filtering, we evaluate the benefit of ranking the valid instruction messages by the relative frequency that their operations appear in the whole program image. The distribution of unique MIPS operations (e.g., *add*, *mul*, *beq*, *sw*, etc.) in each of the five benchmarks are shown in Fig. 7. It is clear that the distributions resemble a power law: some instructions occur very frequently, with *lw* comprising approximately 20% of all operations in each benchmark, while other instructions (e.g., *div*) occur orders of magnitude less often. This information is very useful to SWD-ECC. Hypothetically, if two valid messages under consideration are, for example, *lw* and the rare *sqrt.s*, then SWD-ECC would choose the much more common *lw* instruction as the recovery target.

The fraction of original instructions that could be successfully recovered as a function of the 2-bit error pattern over all five benchmarks is shown in Fig. 8. When the errors are located in the *opcode*, *funct*, and *fmt* fields, the original message can be recovered up to 99% of the time. When both errors are located in the least significant bits of a codeword (roughly, indices 350 through 740), the chances of recovery drop to ≈15%. As noted earlier, this is because the low-order bit fields in MIPS instructions can usually be any value without making the instruction illegal. Moreover, we found that the candidate codewords for an error pattern in the low-order bits tend to have the same operation. For instance, two particular bits that are flipped in the *target address* field of a *j* (jump) instruction causes most of the candidate messages to also be *j* instructions. In these cases, our heuristic recovery scheme fails to distinguish the possibilities, and chooses one of the *j* options randomly. This causes a lower success rate. Nevertheless, our approach achieved an average 34% success rate over all error patterns and the 500 instructions tested from our benchmarks. We consider this a significant achievement compared with a guaranteed system failure that would otherwise occur in most conventional platforms.

V. Conclusion

Software-Defined ECC (SWD-ECC) is a novel approach to improving the resilience of memory to faults with no



(a) Linear-scale (b) Log-scale, 10-sample moving average

Fig. 7. The MIPS instructions that appear in applications (e.g., *add*, *lw*, *bne*, etc.) roughly follow a power law distribution. We use this in the *filtering-and-ranking* strategy as additional side information for heuristic correction of DUEs that occur in instruction memory. To improve figure clarity, instruction mneumonics are not labeled.

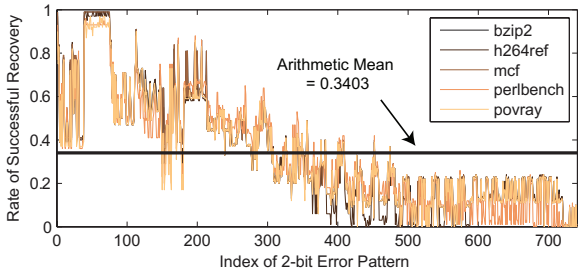


Fig. 8. Rate of successful heuristic recovery for MIPS instruction memory for five benchmarks over all 741 possible 2-bit DUE error locations with a (39,32) SECDED ECC code. Here, we apply the *filtering-and-ranking* strategy, i.e., candidate codewords are first filtered for valid instruction messages before choosing the most-commonly occurring operation as the recovery target.

required change to the hardware architecture. It works through hardware/software collaboration, where the system-level fault-tolerance schemes exploit theoretical fundamentals of the underlying ECC code, and the ECC code exploits available side information about messages stored in memory. Our results showed that on average, 34% of DUEs in an instruction memory can be recovered successfully. We consider this a significant achievement considering that most systems crash upon receiving a DUE (which can be considered as a 0% success rate). Moreover, there is still room for improvement with a more sophisticated use of side information. We acknowledge, however, that in many scenarios it may be preferable to crash deterministically upon encountering a DUE, rather than continuing workload execution without a guarantee of correctness. We outlined a general approach to coping with this issue for applications that are not approximation-friendly or algorithmically error-tolerant. Promising SWD-ECC research topics include approaches for data memories, instruction memories with other ISAs, and even use cases beyond memory systems altogether. The ideas in this paper might be applied to the storage, communications, and information theory fields, and could find use in various domains of computing from embedded and mobile to cloud and supercomputing. Our future work on SWD-ECC seeks to exploit other types of side information for heuristic recovery, derive theoretical properties, adapt the approach to 64-bit ISAs, and study the impact on system resiliency.

Acknowledgment

The authors thank the anonymous reviewers for their constructive feedback. This work was supported by the NSF Grant Nos. CCF-1029030 and CCF-1150212.

REFERENCES

[1] 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, 2011.

[2] V. Sridharan and D. Liberty, "A Field Study of DRAM Errors," tech. rep., AMD, 2012.

[3] 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.

[4] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, vol. 4. Morgan and Claypool Publishers, 2009.

[5] 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.

[6] S. Borkar and A. A. Chien, "The Future of Microprocessors," *Communications of the ACM*, vol. 54, no. 5, 2011.

[7] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate Computing and the Quest for Computing Efficiency," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2015.

[8] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error Detection Using Control Flow Assertions," in *Proceedings of the IEEE Symposium on Computer Arithmetic*, 2003.

[9] N. J. Wang and S. J. Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, 2006.

[10] R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 29, no. 2, 1950.

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

[12] M. Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, no. 4, 1970.

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

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

[15] 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 International Conference on Computer-Aided Design (ICCAD)*, 2013.

[16] P. Chi, C. Xu, X. Zhu, and Y. Xie, "Building Energy-Efficient Multi-Level Cell STT-MRAM-Based Cache Through Dynamic Data-Resistance Encoding," in *Proceedings of the International Conference on Quality Electronic Design (ISQED)*, 2014.

[17] W. Wen, Y. Zhang, M. Mao, and Y. Chen, "State-Restrict MLC STT-RAM Designs for High-Reliable High-Performance Memory System," in *In Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2014.

[18] S. Hong, J. Lee, and S. Kim, "Ternary cache: Three-valued MLC STT-RAM caches," in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, 2014.

[19] Y. Cassuto, M. Schwartz, V. Bohossian, and J. Bruck, "Codes for Asymmetric Limited-Magnitude Errors With Application to Multilevel Flash Memories," *IEEE Transactions on Information Theory*, vol. 56, no. 4, 2010.

[20] J. Wang, K. Vakilinia, T.-Y. Chen, T. Courtade, G. Dong, T. Zhang, H. Shankar, and R. Wesel, "Enhanced Precision Through Multiple Reads for LDPC Decoding in Flash Memories," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 5, 2014.

[21] C. Schoeny, F. Sala, and L. Dolecek, "Analysis and Coding Schemes for the Flash Normal-Laplace Mixture Channel," in *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*, 2015.

[22] F. Sala, K. A. S. Immink, and L. Dolecek, "Error Control Schemes for Modern Flash Memories," *IEEE Consumer Electronics*, vol. 4, 2015.

[23] A. Hareedy, B. Amiri, S. Zhao, R. Galbraith, and L. Dolecek, "Enhanced Precision Through Multiple Reads for LDPC Decoding in Flash Memories," in *In Proceedings of the IEEE Global Communications Conference, Exhibition, and Industry Forum (GLOBECOM)*, 2015.

[24] 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.

[25] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, not ECC, for Hard Failures in Resistive Memories," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010.

[26] M. Gottscho, A. BanaiyanMofrad, N. Dutt, A. Nicolau, and P. Gupta, "DPCS: Dynamic Power/Capacity Scaling for SRAM Caches in the Nanoscale Era," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, 2015.

[27] S. Wang, H. C. Hu, H. Zheng, and P. Gupta, "MEMRES: A Fast Memory System Reliability Simulator," in *The 11th IEEE Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2015.

[28] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, 1987.

[29] F. J. Aichelmann, "Fault-Tolerant Design Techniques for Semiconductor Memory Applications," *IBM Journal of Research and Development*, vol. 28, no. 2, 1984.

[30] R. van Rein, "BadRAM: Linux Kernel Support for Broken RAM Modules." Available at http://rick.vanrein.org/linux/badram/ and accessed 2015-12-04.

[31] "Intel 64 and IA-32 Architectures Software Developer Manuals." Available at http://www.intel.com and accessed on 2015-05-01.

[32] S. Mittal and J. Vetter, "A Survey of Techniques for Modeling and Improving Reliability of Computing Systems," *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[33] P. Elias, "List Decoding for Noisy Channels," tech. rep., Massachusetts Institute of Technology (MIT), Cambridge, MA, 1957.

[34] V. Guruswami, "List Decoding with Side Information," in *Proceedings of the IEEE Annual Conference on Computational Complexity*, 2003.

[35] 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.

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

[37] 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.

[38] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 4 ed., 2012.

[39] "Lattice Semiconductor ECC Module Reference Design RD1025," tech. rep., 2012. Available at http://www.latticesemi.com and accessed 2015-05-01.

[40] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.