Improving Application Resilience by Extending Error Correction with Contextual Information

Alexandra Poulos*†, Dylan Wallace*†, Robert Robey‡, Laura Monroe*,
Vanessa Job*, Sean Blanchard*, William Jones† and Nathan DeBardeleben*
* Ultrascale Systems Research Center[1]
‡Eulerian Codes
Los Alamos National Laboratory
Los Alamos, New Mexico, 87545
Email: {lmonroe, vjob, ndebard, brobey, seanb}@lanl.gov
† Department of Computing Sciences
Coastal Carolina University
Conway, South Carolina, 29528
Email: {alpoulos, dewallace, wjones}@coastal.edu

***Abstract*—Extreme-scale systems are growing in scope and complexity as we approach exascale. Uncorrectable faults in such systems are also increasing, so resilience efforts addressing these are of great importance. In this paper, we extend a method that augments hardware error detection and correction (EDAC) contextually, and show an application-based approach that takes *detectable uncorrectable* (DUE) data errors and *corrects* them.**

We applied this application-based method successfully to data errors found using common EDAC, and discuss operating system changes that will make this possible on existing systems. We show that even when there are many acceptable correction choices (which may be seen in floating point), a large percentage of DUEs are corrected, and even the miscorrected data are very close to correct. We developed two different contextual criteria for this application: local averaging and global conservation of mass. Both did well in terms of closeness, but conservation of mass outperformed averaging in terms of actual correctness.

The contributions of this paper are: 1) the idea of *application-specific* EDAC-based contextual correction, 2) its demonstration with great success on a real application, 3) the development of two different contextual criteria, and 4) a discussion of attainable changes to the OS kernel that make this possible on a real system.

***Index Terms*—Fault tolerance, high performance computing, error correction codes, maximum likelihood decoding**

I. Introduction

High performance computers (HPC), supercomputers, and clusters face a number of challenges as they scale towards exascale [1]. These include power, programmability, reliability [2], and a variety of hardware and software changes which are required to achieve the massive scale necessary to solve the problems of tomorrow.

These machines require both software and hardware efforts to ensure that applications running on them get the right answer. Checkpoint/restart is commonly used to tolerate fail-stop events and globally recover from a failure. Other techniques include fault-tolerant middleware libraries and software-based algorithmic resilience to data corruption.

SRAM, DRAM, and co-processor memory often use error-detection and error-correction codes (EDAC) for fault correction. These codes offer a range of tradeoffs in terms of space, complexity, and power. Chip designers use these tradeoffs and system fault-tolerance requirements to determine what level of protection is required at which tier of the memory hierarchy. It is not uncommon for a system to have several different EDAC schemes present in the memory hierarchy alone.

Conventional coding techniques include Hamming codes [3], Hsiao codes [4], Bose-Chaudhuri-Hocquenghem (BCH) codes [5], Chipkill-correct [6], the simple parity, and other schemes. Semiconductor manufacturers also employ proprietary coding schemes in their products. These codes range in complexity, power-efficiency, data rates, and other important criteria, but a general characteristic of all of these codes is their *detection* and *correction* properties.

In general, the number of bit errors that a given code can detect is larger than the number of bit errors that it can successfully correct. This is a consequence of their mathematical structure. A system using these codes must then address in some way those errors that are detected but are not corrected.

The default behavior upon encountering a detectable but uncorrectable error (DUE) is to log an appropriate error message, and then immediately halt the entire system, including all running processes as well as the operating system itself. In an HPC system that uses checkpoint-restart as the primary defense against faults, the presence of a DUE would induce an application restart from the last known checkpoint file.

This treatment of detected but uncorrected faults leads to wasted time, wasted energy, decreased application efficiency, and longer overall time to solution. It would thus be advantageous if there were a method to address such post-EDAC faults, without impairing correctness of calculation, and without taking the extreme measures of shutting down the process.

[1]A portion of this work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy contract DE-FC02-06ER25750. The publication has been assigned the LANL identifier LA-UR-18-27672.

We note that mathematically, when an uncorrectable error manifests through an EDAC method, only a fraction of all faults can possibly have caused the particular error. We further reduce the set of possible faults by assuming that fewer concurrent faults are more likely to occur than a more numerous number of faults, so only a very small fraction of the possible faults are candidates [7]. Finally, only one or a few of these are even close to the expected value, based on application data for nearby cells. This situation is depicted in the notional Figure 1. We then accept as correct the closest tentative correction. If several choices are acceptably close, choosing the wrong one will cause silent data corruption (SDC), but the application may in fact tolerate this, because the incorrect value is close enough the correct value.

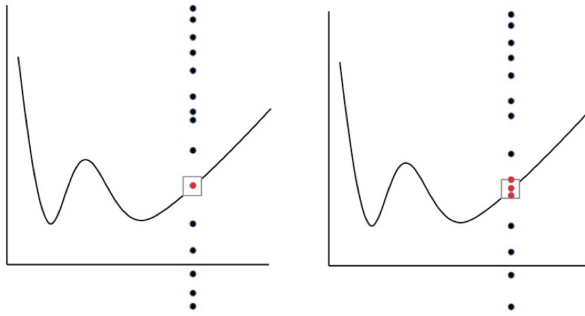


Fig. 1: These notional figures illustrate a 1-dimensional curve with a detected but uncorrected fault at the gray box. The dots represent all possible corrections. In the **left** graph, only one choice is acceptably close. In the **right** graph, three choices are acceptably close, and it is not clear which to choose.

The mathematical idea at the heart of this work is the interplay between different distance metrics. This exploits the idea that two Hamming-close words are often Euclidean-far.

In this paper, we apply contextual application knowledge to output provided by EDAC. We propose modifications to the operating system kernel and to applications, and demonstrate a software extension of existing EDAC mechanisms that would allow a system to continue operation through DUE errors in data by making use of contextual knowledge of the underlying behavior and structure of the application to turn what would have been uncorrectable errors to correctable ones.

The contributions of this paper are as follows: We introduce an application-based contextual method of error correction. We discuss errors in data in great detail. We provide recommendations for the necessary changes to the Linux kernel that would be required to redirect system halts to invoke application specific *handlers* to work in tandem with EDAC subsystems. We document the extensions to and resulting performance of two common EDAC codes [3] [4] that enable these application handlers to carry out corrections. Finally, we demonstrate the impact of these EDAC extensions to a well-known computational hydrodynamics proxy application [8] that is used at LANL and the US DOE complex at large.

II. Prior Work

In 2016, Gottscho et al [9] observed that only a small number of DUEs are actually possible corrections to a DUE caught through EDAC. They used information external to the EDAC method to select from this small number of possibilities and correct some of these DUEs. They then corrected for *instruction errors*, and noted that these methods might be applied to data errors. They filtered obvious incorrect instructions and used cache closeness to eliminate others, and achieved an impressive 34% correction of these instruction errors.

In 2017, Schoeny et al [10] followed up on this work by exploring unequal message protection for random-access memories, by selecting specific messages to have extra protection against errors. This type of research in context-aware resiliency is precisely the type of work that the extreme-scale computing community will need in order to address the growing prevalence and mitigation of faults as feature sizes continue to decrease and as system component counts increase.

In this paper, we follow up by concentrating on *data errors* in a system protected by two common SECDED codes. We use application-specific metrics to select the most likely possible correction, and test these on a relevant application. We also emphasize OS modifications supporting this scheme that can now be implemented in current-generation systems.

We look at data errors here because faults are more likely to occur in data space, due to its relative size. Also, an incorrect instruction value is likely to crash the program, and thus becomes obvious shortly, but an incorrect data value is likely to go unnoticed and cause overall incorrect results at the cost of scientific accuracy and impact to mission.

III. OS Modifications

While DUEs are not common, they are not rare enough that finding clever ways to address them would not be worthwhile. Supercomputers run tightly coupled numerical simulations that, generally speaking, are unable to keep a job running when one node (or the kernel on a node) in that job crashes from a DUE. Therefore, we are interested in understanding in what cases we can tolerate DUEs at an application level and keep applications running at an acceptable fidelity.

A memory location is checked for error not only when accessed by an application but also periodically checked by hardware-based scrubbers [11]. If a DUE occurs it is passed to the Linux kernel. There, the machine check exception (MCE) software has several options for how to handle it (explained in [12]). These options take on the form of the `tolerant` field where the default is for the kernel to log the event and panic (crash). The difference between the `tolerant` values of 0 and 1 is slight and has to do with the kernel's tolerance to dead locks but is not relevant to our discussion here. For a value of 3, the kernel simply logs the event and continues operation. This is, obviously, quite dangerous and is rarely done in practice. Our team uses this feature in neutron beam experiments, for instance. The final choice, 2, is the one of interest to us and it allows for sending a `SIGBUS` to an application.

Figure 2 depicts the flow we discuss in this paper. As mentioned above, the boxes marked in red are the conventional pathways that would cause kernels to crash. In our work, we propose the pathway marked by the thick arrows. Note that this path includes application-specific ECC handlers with options to merely crash the *application*, rather than the entire system, if the application is not able to handle the DUE. We show in Section V that for a science application we studied, it may be very reasonable to in fact apply application context to handle the DUE and resume normal operation without crashing.

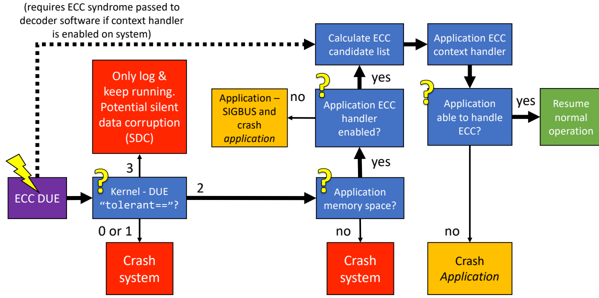


Fig. 2: Path through the Linux kernel handling ECC DUE. The thick arrow pathway depicts the route we propose in the optimal case for successfully handling a DUE with application context without crashing the system.

The default response of a user process receiving `SIGBUS` is to dump core and terminate; however, this signal can be caught by a signal handler allowing the process to run a user defined routine.[3] It is precisely in this routine that we will place our application-specific contextual correction mechanism.

The Linux kernel's `do_machine_check` routine does not currently do more than send a `SIGBUS` to the affected process. In order to intelligently decide what action to take, the application's signal handler will need to be told the linear memory address where the DUE occurred. It is possible to modify the kernel to perform the physical to virtual memory translation and publish that address into a user-readable file in the *sysfs* file system. The signal handler can then pass the address to an application mapping of the data structures in the application to its linear address space. As discussed in Section IV, the coset calculation (the process that deals with computing the possible candidate list of weight-2 members that correspond to the particular DUE encountered) depends on the computed EDAC syndrome as well as the value of the corrupted data. Vendors like AMD provide an interface for accessing the syndrome data via model-specific registers (MSR) for hardware that is protected by EDAC [13].

[3]Note that not all DUE events can be handled in this manner. If the processor context is corrupt, or a DUE occurs in kernel space memory, the kernel will correctly panic and reboot; however, for the purpose of this paper, we make the assumption that errors are most likely to occur in an application's data address space, due to the relatively large fraction of main memory that computational science suites typically encompass compared to the size of the kernel or process control block.

IV. ECC Extensions

As noted above, a variety of EDAC codes are used throughout a computer system. For the sake of simplicity, in this paper we only consider a single level of memory and assume that the memory level uses one of two common ECC codes, either a modified Hamming code or the Hsiao code. Both modified Hamming and Hsiao codes are characterized as single-error *correcting*, double-error *detecting* (SECDED) codes. Both codes have similar capabilities; however, Hsiao codes are minimal in that they have fewer ones in the parity-check matrix, and as such are more efficient when implemented in hardware [4]. In a SECDED code, a single-error can be successfully detected and corrected, and a two-bit error can be successfully detected *but not corrected*. As was discussed in Section III, when a two-bit error is detected, the operating system will typically log the event and then completely halt the system. In this work, we discuss the extension of the existing ECC hardware with a software-enhancement that can make use of contextual data to further improve the efficacy of current hardware-only strategies by correcting previously uncorrectable errors. We describe how the detection-correction calculations are performed under typical circumstances.

A. Basic Concepts in EDAC

A forward error-correcting code (ECC) is a method of adding redundancy to datavectors to permit detection and correction of errors occurring on that datavector. Forward error-correcting codes employ *maximum-likelihood decoding*, which means that those faults most likely to occur are corrected, but others are only detected, or may even be miscorrected. One simple example of an error-correcting code is the addition of a parity bit to the datavectors; this can only detect a single bit error, but not correct it.

B. Terminology and a Simple Example

Many common EDAC systems are based on binary linear codes, like Hsiao codes [4] and extended Hamming codes [3]. We will be looking at (n, k, d) binary linear codes: they add redundancy to datavectors of length k to make codewords of length n. The parameter d is the minimum number of positions in which any two codewords differ. In what follows, all codes will be binary linear codes and we will refer to them as (n, k, d)-codes.

Each (n, k, d)-code has a *generator matrix* G and a *parity check matrix* H. Codewords are generated by multiplying a datavector of length k by an $k \times n$ generator matrix. The first k columns of the generator matrix are the identity matrix and the remaining $n-k$ columns generate the check bits that enable the detection and correction of errors. For instance, Hamming $(8, 4, 4)$ encodes datavectors of length 4 to make codewords of length 8. It has generator matrix G:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

To encode the datavector $[0,1,1,0]$ (6 in decimal, left most bit most significant), we compute $[0110]G = [01101100] = \bar{v}$. Note: here all arithmetic is done modulo 2. To compute the seventh position in $\bar{v}$ (second position from the right), we take $[0110][0111]^T = 2 \equiv 0 \pmod 2$. There are 2^k different length k vectors and because this is a linear code, they will all have different encodings. Thus a (n,k,d)-code contains 2^k codewords of length n [7].

The parity check matrix H for this Hamming $(8,4,4)$ code is

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

For any (n,k,d)-code, the parity check matrix H has the property that if you multiply H by the transpose of any codeword, the result is a vector of all zeros. For instance, if we take $H\bar{v}^T$ for our codeword $\bar{v} = [01101100]$, we get $[0000]$. The result of multiplying the parity check matrix by any vector of length n (codeword or not) is called the *syndrome* of that vector. For codewords, the syndrome will always be $\bar{0}$.

Suppose a bit gets flipped in our codeword $\bar{v}$. We can think of this as $\bar{v}$ being XOR-ed with a *fault vector* $\bar{f}$. For instance, to flip the fourth bit from the left, we can imagine $\bar{v}$ was XOR-ed with $[00010000]$ to create a corrupted vector $\bar{w}$:

$$\begin{aligned} \bar{w} &= \bar{v} \text{ XOR } \bar{w} = [01101100] \text{ XOR } [00010000] \\ &= [01111100] \end{aligned}$$

If we multiply the parity check matrix H by $\bar{w}$ transpose, we get the syndrome $[1110]$. Since the syndrome is not zero, we know that an error occurred.

In fact it is the case that if we took any codeword from the extended binary (8,4,4) code and flipped bit 4, the syndrome of the resulting vector would be exactly the same, i.e. $[1110]$. Also, since the length 8 zero vector is one of the codewords, namely $\bar{0}G$, the vector $[00010000]$ also has syndrome $[1110]$, or in decimal, 14. We will see that for any vector $\bar{w}$, if the syndrome of $\bar{w}$, namely $H\bar{w}^T$, is 14, it is most likely that $\bar{w}$ arose because bit 4 was flipped in a codeword. Thus we can recover the codeword by flipping bit 4 back.

The length n binary vectors can be partitioned into sets called *cosets*. Each coset contains vectors all of whom have the same syndrome. For any (n,k,d)-code, each coset will contain 2^k vectors of length n and there will be 2^{n-k} cosets. For instance, if we represent each eight-bit word as an integer, the coset that contains the vector $[00010000]$, (16 as an integer), is $\{16, 157, 91, 214, 56, 186, 124, 241, 14, 131, 69, 200, 41, 164, 98, 239\}$.

All elements of the same coset have the same syndrome, and no two different cosets have the same syndrome. The syndrome of any uncorrupted codeword is $\bar{0}$.

In Figures 3a and 3b, the first column contains the syndrome, and this syndrome indexes the associated coset. We define the *weight* of a binary vector to be the number of non-zero bits in the vector. In Figure 3b we see that syndrome 14 indexes a coset that only has one weight one member. In fact, all single-bit errors correspond to a syndrome that indexes a coset that only contains a single weight one member.

C. The Correction Process

When an error is detected the syndrome points to a coset. Each coset has a coset leader, which is a coset member with the smallest weight, i.e. a member with the smallest number of non-zero bits. All of the n possible weight one error vectors are leaders of their respective cosets (and also the only weight one members of their cosets). These codes use maximum-likelihood decoding, meaning that when a syndrome points to a coset with a weight one coset leader, the code is going to assume that a weight one error occurred (as opposed to more errors, which is less probable assuming independent error probabilities). It attempts to correct by XOR-ing the corrupted data with that weight one member.

When a weight two error occurs, there is no unique minimum weight member in its coset, meaning that more than one two-bit error could have resulted in the same syndrome. This is illustrated in Figure 3b. This is precisely why these codes are able to *detect* but not *correct* 2-bit errors, and why the operating systems are generally configured to halt the system, since the hardware system alone cannot correct the error.

For instance, suppose we use the $(8,4,4)$ extended Hamming code as before and the value $[0,1,1,0]$ is encoded to create the codeword $\bar{v} = [01101100]$. Say we experience a two-bit fault at positions 4 and 5, i.e., the fault vector is $\bar{f} = [00011000]$. This results in the corrupted vector

$$\bar{w} = \bar{v} \text{ XOR } \bar{f} = [01110100].$$

which is 116 decimal. Compute the syndrome by taking $H\bar{w}^T = [0110]^T$, which is 6 decimal. We see in Figure 3a, that syndrome 6 corresponds to the coset $\{6, 139, 77, 192, 33, 172, 106, 231, 24, 149, 83, 222, 63, 178, 116, 249\}$. From Figure 3b, also indexed by syndrome 6, we see that this coset contains four weight 2 members, the fault vectors $6, 192, 33$, and 124. When we XOR these fault vectors with the corrupted vector $\bar{w}$ and find that $\bar{w}$ could have come a 2-bit error in original codewords $\{114, 180, 85, 108\}$. Looking at the first 4 bits of the 8 bit representations of these codewords, we see that they correspond to the datavectors $\{7, 11, 5, 6\}$, which is our candidate list.

Note that the originally encoded value 6 is among these possibilities. In Section IV-D of the paper, we show how contextual information can be used to select the best candidate for correction for DUEs.

D. The Scheme and Why It Works

Under the assumption that *at most* a single bit error per word has occurred, we are guaranteed to find the original *correct* codeword in the coset associated with the syndrome produced by the hardware SECDED ECC subsystem. Weight-two errors are detected but not corrected, which means that there is more than one weight-two element in its coset.

However, we note that there are only a relatively small number of weight two members within a weight-two coset.

Syn	Fault patterns															
0	0	141	75	198	39	170	108	225	30	147	85	216	57	180	114	255
1	1	140	74	199	38	171	109	224	31	146	84	217	56	181	115	254
2	2	143	73	196	37	168	110	227	28	145	87	218	59	182	112	253
4	4	137	79	194	35	174	104	229	26	151	81	220	61	176	118	251
8	8	133	67	206	47	162	100	233	22	155	93	208	49	188	122	247
14	16	157	91	214	55	186	124	241	14	131	69	200	41	164	98	239
17	32	173	107	230	7	138	76	193	62	179	117	248	25	148	82	223
11	64	205	11	134	103	234	44	161	94	211	21	152	121	244	50	191
13	128	13	203	70	167	42	236	97	158	19	213	88	185	52	242	127
3	3	142	72	197	36	169	111	226	29	144	86	219	58	183	113	252
5	5	136	78	195	34	175	105	228	27	150	80	221	60	177	119	250
6	6	139	77	192	33	172	106	231	24	149	83	222	63	178	116	249
9	9	132	66	207	46	163	101	232	23	154	92	209	48	189	123	246
10	10	135	65	204	45	160	102	235	20	153	95	210	51	190	120	245
12	12	129	71	202	43	166	96	237	18	159	89	212	53	184	126	243
15	17	156	90	215	54	187	125	240	15	130	68	201	40	165	99	238

(a) Partition of the set of all 8-bit faults into the code and its cosets. Each 8-bit word is represented as an integer, for conciseness. The top row is the code itself, and the rest of the rows are cosets obtained by XORing the first row element with the codeword immediately above.

Syn	Weight of fault patterns															
0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	8
1	1	3	3	5	3	5	5	3	5	3	3	5	3	5	5	7
2	1	5	3	3	3	3	5	5	3	3	5	5	5	5	3	7
4	1	3	5	3	3	5	3	5	3	5	3	5	5	3	5	7
8	1	3	3	5	5	3	3	5	3	5	5	3	3	5	5	7
14	1	5	5	5	5	5	5	5	3	3	3	3	3	3	3	7
17	1	5	5	5	3	3	3	3	5	5	5	5	3	3	3	7
11	1	5	3	3	5	5	3	3	5	5	3	3	5	5	3	7
13	1	3	5	3	5	3	5	3	5	3	5	3	5	3	5	7
3	2	4	2	4	2	4	6	4	4	2	4	6	4	6	4	6
5	2	2	4	4	2	6	4	4	4	4	2	6	4	4	6	6
6	2	4	4	2	2	4	4	6	2	4	4	6	6	4	4	6
9	2	2	2	6	4	4	4	4	4	4	4	4	2	6	6	6
10	2	4	2	4	4	2	4	6	2	4	6	4	4	6	4	6
12	2	2	4	4	4	4	2	6	2	6	4	4	4	4	6	6
15	2	4	4	6	4	6	6	4	4	2	2	4	2	4	4	6

(b) Weights of the 8-bit faults from Figure 3a. These are colored by correctability of the faults: blue → correct, green → correctable, yellow → detectable but uncorrectable, peach → detectable but mis-corrected, and red → undetectable. No weight-2 fault is correctable, because any coset having any weight-2 vector has more than one.

Fig. 3: Here we show the 16 members (left) and weights (right) of each of the 16 cosets that correspond to the 16 unique 4-bit syndromes (abbreviated "Syn" above) for the Hamming(8,4,4) code. The rows in each subfigure indexed by syndrome 14 correspond to the single bit-flip example below, while the rows indexed by 6 correspond to the double bit-flip example.

This means that for this type of DUE error, we need only consider a small subset of the elements in the coset. For example, in Figure 3b, the coset indexed by syndrome 6 only has four weight-two members. Furthermore, this list of weight-two vectors must only be computed once for a given EDAC scheme, and as such, can be stored in a lookup table (LUT). This bodes well from a computational point of view, as this additional work represents overhead to the existing ECC schemes employed today.

We now have a reduced set of possible candidate values, and one of these **must** have been the original correct codeword. To reduce the choice further, we make use of application-specific contextual information to *further eliminate* unlikely candidates from consideration. For example, in a 3-dimensional computational fluid dynamics calculation, we might look at the distance of the neighboring cells to the small number of candidates and choose the nearest candidate as our corrected word.

In other words, by making use of context-specific information about the application, we may be able to make a more educated guess about which candidate corrections are more likely to correspond to the initially stored value.

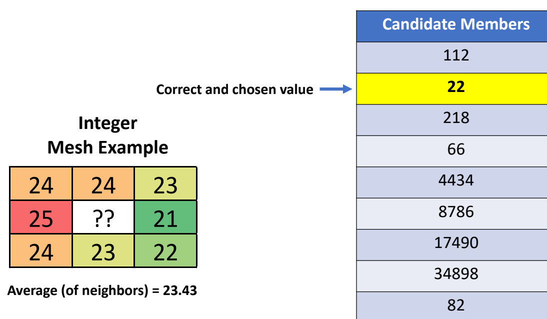


Fig. 4: Here we use a simple 16-bit Hamming code to encode integer data, representing temperatures in a 2D heat-transfer model, in order to illustrate the concept of context specific knowledge that is leveraged to identify the correct original data value of the center cell from the list of possible candidates in the coset list, thereby moving a DUE to a correctable error. The average of the surrounding neighbors is 23.43, and as such, 22, the correct value, is the closest candidate. In this case the strategy picks the correct answer.

E. The Scheme Illustrated by the Example

In order to demonstrate this concept with a simple "hand" example, let us assume that we are dealing with a simple two-dimensional heat-transfer application, as depicted in Figure 4.

In this example, for the sake of simplicity, we use a 16-bit Hamming code to encode integer data representing temperatures in a 2D heat-transfer model. In this example, we simulate a 2-bit fault on the data element located in the center of the mesh, in this case, the integer 22. We then take the corrupted data value and calculate the syndrome to find that a 2-bit error in fact occurred, which for this SECDED code results in a detectable but uncorrectable error. We then calculate the list of possible values that have the same syndrome to construct the coset. As we can see, the correct value, 22, is in fact one of the possible candidates. At this point, this list of candidates, along with the information from the operating system that allows logical address of the data value can be determined by the application's helper function, as discussed in Section III, we are now ready to pinpoint the correction. In this case, since we are dealing with 2D heat transfer data, one potential conclusion is to assume [4] that the correct original value is numerically close to the average of the surrounding neighbors. This average is calculated to be 23.43. By searching through the members

[4]We acknowledge that this assumption may not necessarily hold, and that in general each "protected" data-structure would potentially require its own intelligent helper to make this determination. The purpose of this paper is as a proof of concept and to begin to study under what conditions this strategy would and would not work well.

of the coset list, the handler quickly identifies that the value 22 is the closest, and corrects the corrupted value to this number.

One question that the above example helps make clear is that the ability to correctly identify the original data depends heavily on the distribution of the members of the coset and how "far" these number are from each other and from the correct solution. For example, if the coset had contained the number 23, then our simple strategy of picking the element closest to the average would not have worked and would have incorrectly identified 23 instead of 22. As a result, in Section V we present the results of our study of the distribution of coset members in isolation in a large stand-alone statistical-based approach for both integer and floating-point data types, and conclude with a presentation of the performance results of this approach when implemented in CLAMR, a cell-based adaptive mesh refinement hydrodynamics proxy application of interest to the United States Department of Energy.

V. Results

We look here at the distribution of candidates in cosets, and then determine the distance between candidates and an estimated correct solution. An application-specific handler with this contextual information can then eliminate more distant values as candidates, and choose the closest as a correction. We implement this strategy for a hydrodynamics proxy application and present overall performance results.

A. Distribution of Candidates in Cosets

To calculate the distribution of candidates, we sampled across the codeword space and across the possible 2-bit error space. We sampled the space instead of performing an exhaustive search because the 32- and 64-bit lengths tested, plus the additional parity bits, made iterating across all possible combinations intractable on the systems that we used to conduct our experiments.

We conducted 100K trials of randomly-chosen initial values with randomly-chosen 2-bit faults. We started with the 32- and 64-bit integer space for Hamming and Hsiao encodings, and found that the members of the cosets were distributed relatively close to the correct value. We found that the distribution was not closely tied to the selected EDAC strategy, as seen in Figure 5. Additionally, the general distribution of coset members did not exhibit as much variation as in the single 16-bit "hand" example in Figure 4. This reinforces the concept that it is important to conduct these statistical studies to determine the general behavior.

Given that floating-point values are of much more interest to the scientific computing community, we must also explore this space in addition to integers. To address this, we conducted the same set of experiments as described above, but this time for 32- and 64-bit IEEE floating-point numbers. These results are presented in Figure 6.

Here we see that the distribution of values of coset members is clustered around and to the left of the normalized logdistance of 0, in much the same way as for integers. The values concentrated around the normalized logdistance of 0 indicate that the distance is proportional the magnitude of the original value. We can see that there are some instances of distances much further to the right which correspond to cases where the bit-flips impact the *exponent* rather than the *mantissa*. Furthermore, since the IEEE floating-point representation uses many fewer bits for the exponent than for the mantissa (8 versus 23 for single-precision and 11 versus 52 for double-precision), it makes sense that corruptions will more often manifest in the fraction, thus resulting in even smaller absolute distances. This consequence is also seen for coset members, as we see in Figure 6.

This would seem to bode poorly for context-aware handlers in floating-point applications, as this clustering of values will likely make it harder to identify the correct value from all the coset members. However, we will show later that this *may not* be as important for some applications.

As discussed in Section IV-D, the number of candidates in each coset that could possibly be the original value is relatively small. This list is computed using a lookup table that can be pre-computed for the particular EDAC scheme, and therefore further reduces the run-time overhead of this approach.

The example in Figure 7 shows that the number of candidate values to be considered is small compared to the overall size of the code, and that the sizes do not depend on the choice of data representation. This is because the EDAC strategies in use are agnostic to the meaning of the data encoded, and simply see the values as a collection of meaningless bits.

This shows how the handler will need to consider only a small number of candidate values and so will correct quickly, because of the short candidate list and the easy calculations.

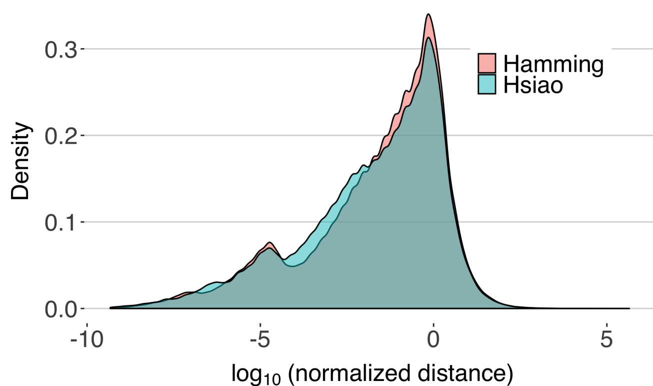
B. Performance in CLAMR

The distribution of candidate corrections for reasonably small values can be very heavily concentrated near the actual correct solution. A relevant question is how well miscorrections will be tolerated in a real application.

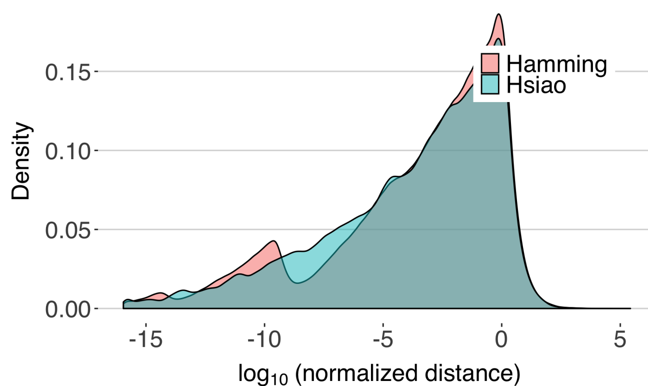
We tested this correction method using CLAMR [8], a cell-based adaptive mesh refinement hydrodynamics code that serves as a performance proxy for many other applications of interest. It has been used on a number of occasions as a vehicle to study fault tolerance and resilience [14], [15]. CLAMR simulates the displacement of a volume of water when perturbed and the resulting waves using the shallow water equations [16].

In our experiment, CLAMR was initialized, and then allowed to run to a certain point, where the state of the application is then checkpointed. The associated data was then used as a starting point for studying our strategy's behavior under fault injection. We iterated across every cell in this grid, and randomly injected 2-bit faults into the floating-point data representing the wave height at that point.

For each injection, the EDAC subsystem detected the DUE, and called our helper routine that calculates all weight-2 members of the coset containing the error. These were masked onto the corrupted values to determine the candidate corrections, as described in Section IV. The application-specific handler

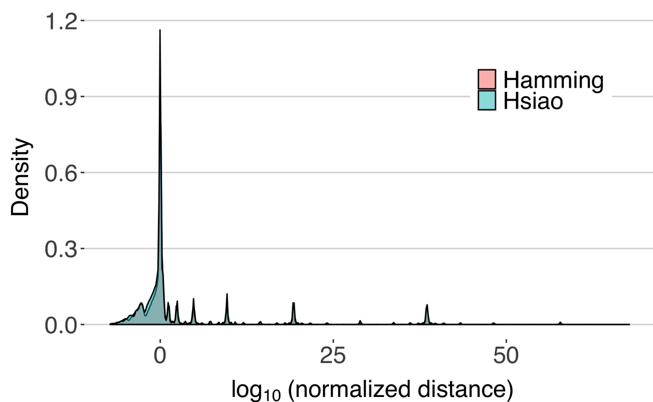


(a) Distribution of distances from the correct value to all members of the coset for **32-bit integers**.

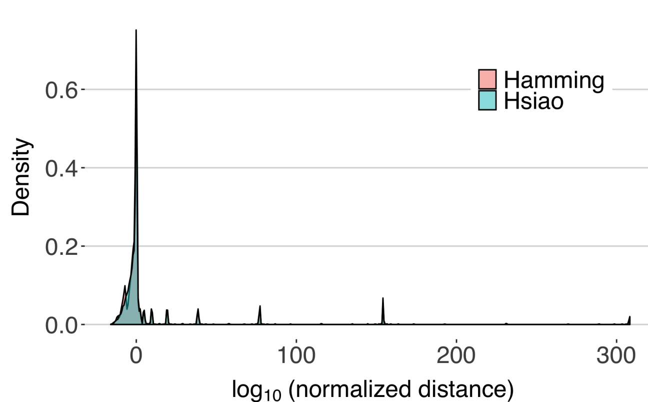


(b) Distribution of distance from the correct value to all members of the coset for **64-bit integers**.

Fig. 5: The distances between the correct data value and all candidate members of the coset *not including the correct value itself* are calculated and plotted for 100K randomly chosen **integers** with randomly chosen 2-bit error injections. Most of the candidates in the coset are very close numerically to the correct value as is evidenced by the large concentration at and to the left of normalized logdistance 0. *Note the x axis is the log of the normalized distance, i.e.* $log\,(\mid original - candidate \mid /original)$.



(a) Distribution of distances from the correct value to all members of the coset for **32-bit floating-point**.



(b) Distribution of distance from the correct value to all members of the coset for **64-bit floating-point**.

Fig. 6: Results for **floating-point** numbers. Most of the candidates in the coset are very close numerically to the correct value as is evidenced by the large concentration at the normalized logdistance 0. *Note the x axis is the log of the normalized distance, i.e.* $log\,(\mid original - candidate \mid /original)$. There are a few instances of distances further to the right of 0 than in the above case for integers, which generally correspond to bit-flips that impact the *exponent* rather than the *mantissa*.

takes the candidate list as input, and chooses the best of these, according to the criteria in use. We ran two sets of tests using different criteria: 1) the local average of the surrounding cells' heights, and 2) the global conservation of mass.

Figure 8 shows a visualization of the wave heights over the region of interest in the timestep before faults are injected. This correct timestep serves as visual comparison to the incorrect results in the following timestep, after the fault injection.

Figure 9 shows the result of the tests using the local averaging criterion. Figures 9a and 9c show whether or not the given cell's selected candidate is correct, for the Hamming and Hsiao encodings. The handler performs well for some cells, but for others, it miscorrects the wave height. Figures 9b and 9d shows the resulting "corrected" grid, containing both correct and miscorrected wave heights.

Even though incorrect, the miscorrected cells still have wave heights relatively close to the correct heights. On average, miscorrected values differ on average by 0.123 units, representing approximately 1.78% error for Hamming, and 0.114 units and 1.67% error for Hsiao, as shown in Figure 11a.

Figure 11a shows that there are miscorrections of more than 1 unit from the correct original value. In Figure 12, we show a cell that corresponds to a miscorrection of 2.5 units, and show the values of the surrounding neighbors. The average of the neighbors is 14.224. Candidates calculated from the syndrome and the corresponding weight-2 members of the coset are shown to the left in Figure 12. In this case, the element from the list that is closest to the average is 13.046,

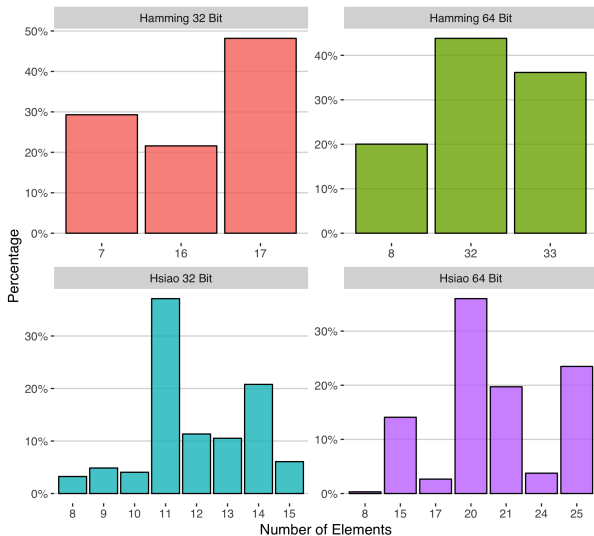


Fig. 7: The number of uniquely-sized candidate lists and their lengths. We see that the number of candidates values that must be considered is relatively small, and as such, a handler should be able to quickly consider all remaining possibilities.

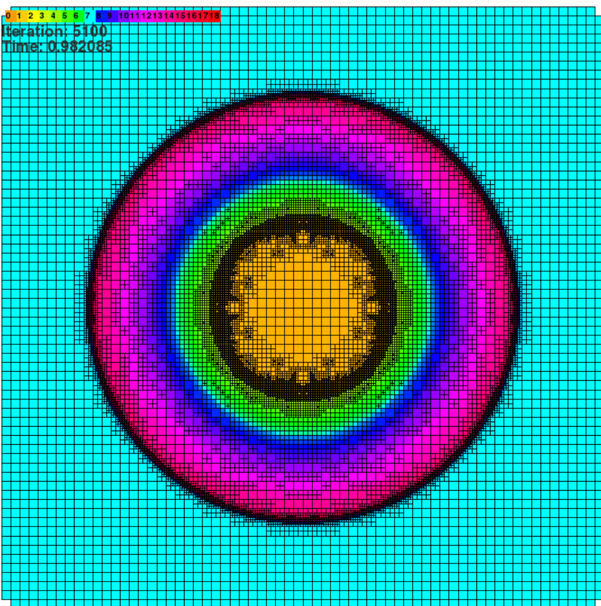


Fig. 8: CLAMR's grid depicting wave height just before fault injection. This represents the original uncorrupted data. The grid is initially 64x64 cells wide with three levels of possible mesh refinement enabled. Here there are a total of approximately $25K$ individual cells across all three levels.

not the correct answer of 15.556. The non-linearity of the surface of the wave on this set of cells combines with the abundance of elements close to the correct answer to cause the "nearest to neighbor average" to perform poorly.

In an attempt to improve upon this performance, we looked at additional application-specific information. For example, the simulation must satisfy the law of conservation of mass, so that is another possible criterion. We tested our handler with this criterion, so that upon detection of a DUE, the handler picked the candidate consistent with the mass conservation invariant, to the closest extent possible within the limits of typical floating-point rounding error. While CLAMR makes use of enhanced floating-point summation techniques such as Kahan summation [17], our helper function could be improved by following suit, as discussed in Section VIII. Nonetheless, any technique that makes use of a fixed number of bits will impose limits to precision and the associated truncation errors.

In Figure 10 we present the much improved results of experiments using conservation of mass as the criterion. There are many fewer miscorrections, as shown by the smaller number of blackened cells in Figures 10a and 10c as compared to Figures 9a and 9c. Furthermore, the magnitude of the error in the miscorrections is very much smaller, with average error values 9.60×10^{-15} units and $2.41 \times 10^{-13}\%$ error and 7.93×10^{-15} and $1.67 \times 10^{-13}\%$ for Hamming and Hsiao, respectively. These are truly negligible errors, and would likely be reduced even further by using enhanced precision techniques in our helper function.

VI. Comparison to Other Software Fault-Tolerance Strategies

The use of information produced by the hardware EDAC reduces calculation, and gives this software-based technique a great advantage over some other software-based methods.

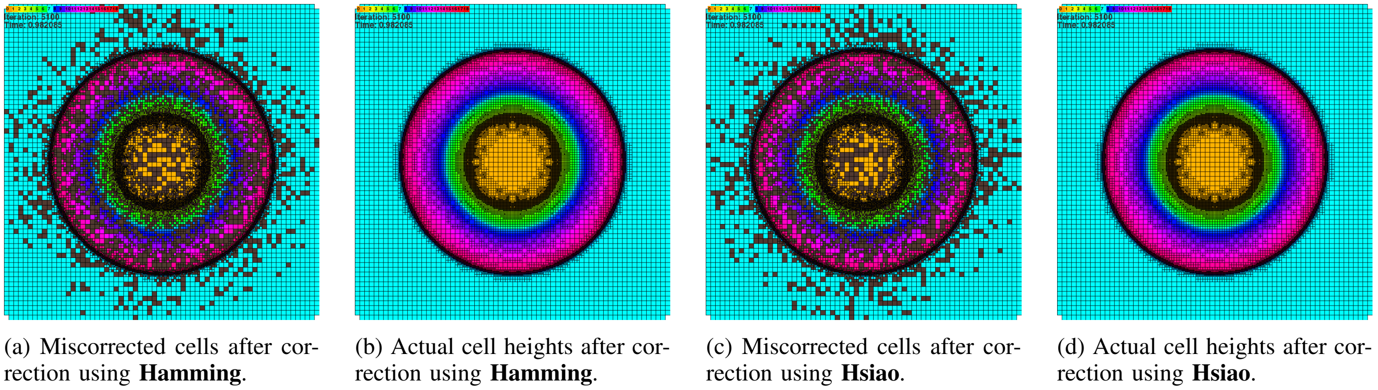
CLAMR and other simulations often make use of invariants like conservation of mass for error detection. However, they must then periodically check that these remain invariant, adding overhead to the already complex computations they perform. In our strategy, we do not waste time periodically checking invariants, but only take action when a DUE is detected asynchronously by the EDAC method when the hardware issues an interrupt.

Furthermore, such approaches may detect that an invariant assertion no longer holds, but they do not correct the error causing this, since they are unable to locate it in the data structure. In this case, a checkpointed application would have to restart from the last known good state. In contrast, our strategy combines EDAC information about the physical location of the DUE in hardware with the application-specific domain knowledge, so not only detects, but also locates and corrects these types of errors, and there is no need for a restart.

While software-based fault-tolerant mechanisms exist for some problems, like those initially studied in [18], they also require additional overhead and require periodic synchronous checksumming or checksum recalculations. The use of information from the EDAC method greatly reduces this overhead.

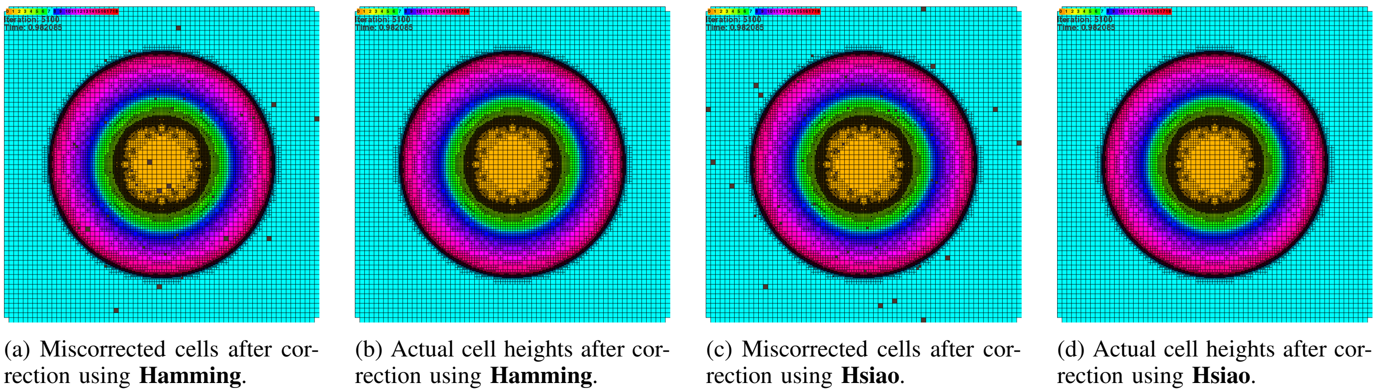
VII. Future Work

Our results are promising as a first step; however, further study is required in several areas, in order to best deploy this scheme onto a working system. Future work will include the systems work needed for this end. It will also include further investigation into application-specific criteria for a range of



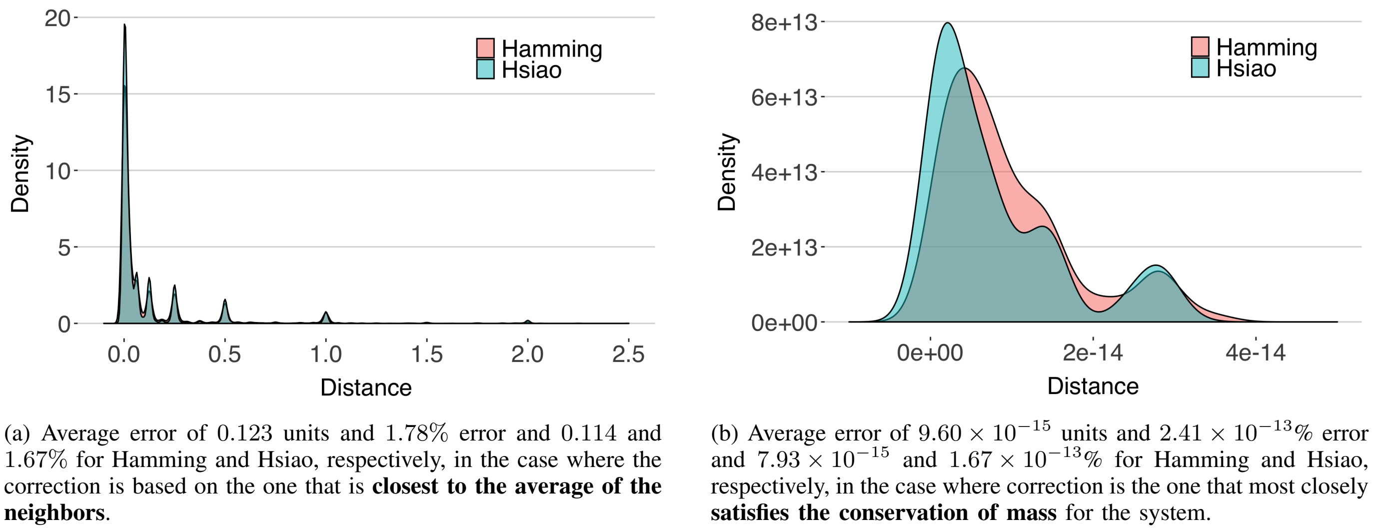
(a) Miscorrected cells after correction using **Hamming**.
(b) Actual cell heights after correction using **Hamming**.
(c) Miscorrected cells after correction using **Hsiao**.
(d) Actual cell heights after correction using **Hsiao**.

Fig. 9: The results of attempted contextual correction in CLAMR, using as context the **average of the surrounding neighbors**. While 53% of the $25K$ cells are miscorrected, (*shown as black cells*) they are miscorrected to values that are very close to the correct original values, as seen in Figure 11a.



(a) Miscorrected cells after correction using **Hamming**.
(b) Actual cell heights after correction using **Hamming**.
(c) Miscorrected cells after correction using **Hsiao**.
(d) Actual cell heights after correction using **Hsiao**.

Fig. 10: The results of attempted contextual correction in CLAMR, using as context the **conservation of mass for the system**. Here, only roughly 1% of the cells are miscorrected, and the amount of error is extremely small, as seen in Figure 11b.



(a) Average error of 0.123 units and 1.78% error and 0.114 and 1.67% for Hamming and Hsiao, respectively, in the case where the correction is based on the one that is **closest to the average of the neighbors**.

(b) Average error of 9.60×10^{-15} units and $2.41 \times 10^{-13}\%$ error and 7.93×10^{-15} and $1.67 \times 10^{-13}\%$ for Hamming and Hsiao, respectively, in the case where correction is the one that most closely **satisfies the conservation of mass** for the system.

Fig. 11: Distribution of errors in corrected cell heights as defined as $|$*corrected heights - original heights*$|$.

DOE applications, and development of such methods for other important EDAC schemes. Future theoretic work would include investigation into the combinatorics of the EDAC methods, as well as numerical techniques for summation.

There exists a tradeoff between the effort required to implement the contextual correction and the overall improved ap-

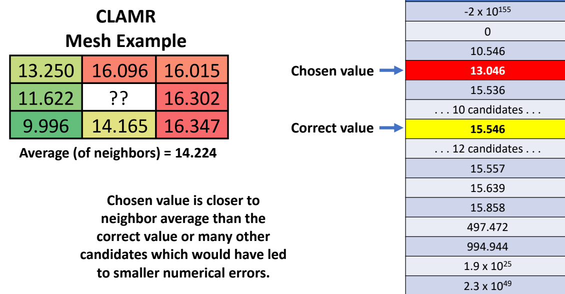


Fig. 12: Example DUE miscorrection in CLAMR

plication resilience. These corrections require domain-specific knowledge, so, additional study in the generalizability of this approach to multiple applications is needed.

We note that the mass conservation technique we have shown here outside of the CLAMR application uses simple summation methods as is used in most applications. CLAMR uses an enhanced precision technique [19] that would allow us to improve on these excellent results even further. The error correction handler can use enhanced precision sums as well. In a study by Anderson [20], this was shown to reduce the possible global sums, and thus the corrections to potentially two or three options that vary in the least significant bit or two. Though her work was for dot products, the global sum has been observed to be similarly restricted in possible values. Though some additional precision is lost in the subtraction of two similar large values, it is likely that this would drive the miscorrected value distribution down even more. Since regular sums are more commonly used and the results are already remarkable, we leave this improvement to future implementations.

Finally, some scientific applications are inherently resilient to small perturbations in their data, and as such, would be better candidates for this type of contextual correction in the presence of miscorrections. While we have done some initial work in this area of inherent resilience [21], a great deal of fundamental research remains.

VIII. Conclusions

In this work, we have discussed a method that uses information from the hardware-based EDAC method in combination with application-specific information to correct detectable but formerly uncorrectable errors in application data.

We investigated this in depth with excellent success on a proxy hydrodynamics code, statistically testing two different contextual criteria and two commonly used binary linear EDAC schemes. We have proposed feasible modifications to the operating system, and helper functions that will enable the system to make these application-contextual corrections.

The success of the application testing combined with currently achievable modifications to the system mean that this is a method that can actually be implemented on a real system to correct the double errors that formerly caused system crashes.

References

[1] J. Sarrao, "Exascale Computing Project: Status and Next Steps," US DOE Office of Science Advanced Scientific Computing Advisory Committee - Presentation and Report, Tech. Rep., 04 2018.

[2] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

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

[4] M. Y. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes," *IBM J. Res. Dev.*, vol. 14, no. 4, pp. 395–401, Jul. 1970.

[5] 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.

[6] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," 01 1997.

[7] V. Pless, *Introduction to the Theory of Error-Correcting*, 3rd ed. Wiley, 1998.

[8] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey, "Cell-based adaptive mesh refinement implemented with general purpose graphics processing units," Los Alamos National Laboratory – Eulerian Codes, Tech. Rep., 2012.

[9] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software-defined error-correcting codes," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, June 2016, pp. 276–282.

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

[11] N. Kim and K. Choi, "A design guideline for volatile stt-ram with ecc and scrubbing," in *2015 International SoC Design Conference (ISOCC)*, Nov 2015, pp. 29–30.

[12] "Configurable sysfs parameters for the x86-64 machine check code," https://goo.gl/xf9nVG, accessed: 2018-08-07.

[13] "BIOS and Kernel Developer's Guide (BKDG) for AMD Processors," https://goo.gl/t1SfcJ, Tech. Rep., 02 2015.

[14] Q. Guan, N. DeBardeleben, B. Atkinson, R. W. Robey, and W. M. Jones, "Towards building resilient scientific applications: Resilience analysis on the impact of soft error and transient error tolerance with the CLAMR hydrodynamics mini-app," in *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, 2015, pp. 176–179.

[15] B. Atkinson, N. DeBardeleben, Q. Guan, R. W. Robey, and W. M. Jones, "Fault injection experiments with the CLAMR hydrodynamics mini-app," in *25th IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Naples, Italy, November 3-6, 2014*, 2014, pp. 6–9.

[16] D. A. Randall, "The Shallow Water Equations," Department of Atmospheric Science Colorado State University, Fort Collins, Colorado 80523, Tech. Rep., 2008.

[17] W. Kahan, "Pracniques: Further remarks on reducing truncation errors," *Commun. ACM*, vol. 8, no. 1, pp. 40–, Jan. 1965.

[18] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, June 1984.

[19] R. W. Robey, J. M. Robey, and R. Aulwes, "In search of numerical consistency in parallel programming," *Parallel Computing*, vol. 37, no. 4, pp. 217 – 229, 2011.

[20] A. Anderson, "Achieving Numerical Reproducibility in the Parallelized Floating Point Dot Product," College of Saint Benedict/Saint John's University, Tech. Rep., 2014.

[21] L. Monroe, W. M. Jones, S. R. Lavigne, C. H. Davis IV, Q. Guan, and N. DeBardeleben, "On the inherent resilience of integer operations," in *Euro-Par 2016: Parallel Processing Workshops - Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers*, 2016, pp. 648–659.