Parity++: Lightweight Error Correction for Last Level Caches

Irina Alam, Clayton Schoeny, Lara Dolecek and Puneet Gupta
Department of Electrical and Computer Engineering, University of California, Los Angeles,
irina1@ucla.edu, cschoeny@ucla.edu, dolecek@ee.ucla.edu and puneetg@ucla.edu

***Abstract*—As the size of on-chip SRAM caches is increasing rapidly and the physical dimension of the SRAM devices is decreasing, reliability of caches is becoming a growing concern. This is because with increased size of caches, the likelihood of radiation-induced soft faults also increases. As a result, information redundancy in the form of Error Correcting Codes (ECC) is becoming extremely important, especially to protect the larger sized last level caches (LLCs). In typical ECCs, extra redundancy bits are added to every row to detect and correct errors. There is additional encoding (while writing data) and decoding (while reading data) procedures required as well. In caches, these additional area, power and latency overheads need to be minimized as much as possible. To address this problem, we present in this paper Parity++: a novel unequal message protection scheme for last level caches that preferentially provides stronger error protection to certain "special messages". This protection scheme provides Single Error Detection (SED) for all messages and Single Error Correction (SEC) for a subset of messages. Thus, it is stronger than just a basic SED parity and has much lower parity storage overhead (4X lower for a 64-bit memory) and lower error detection energy than a traditional Single Error Correcting, Double Error Detecting (SECDED) code. We also evaluate Parity++ with a memory speculation procedure that can be used with any ECC scheme to hide the decoding latency while reading messages when there are no errors.**

***Index Terms*—caches, lightweight error-correction, memory speculation**

I. Introduction

As demand and size of on-chip caches is increasing rapidly and the physical dimension and noise margins are decreasing, reliability of caches is increasingly becoming an important issue. As given in [1], [2], the vulnerability of SRAM caches to soft errors grows with increase in size. Also with reduction in physical dimensions of these devices, the critical charge required to flip the content of a cell due to a particle strike decreases. As a result, the soft error rate is higher for large capacity caches. The widely used technique to guarantee reliability of storage devices is using information redundancy in the form of Error Correcting Codes (ECC). In typical ECCs, extra redundancy bits are added to every row to detect and correct errors. There are additional encoding (while writing data) and decoding (while reading data) procedures required as well. Thus ECCs come with encoding and decoding mechanisms that incur additional overheads in terms of latency and energy. Both these overheads are critical for caches and hence, ECC protection was not widely used in caches till recently. However, due to the increased reliability concerns of large capacity caches and processor performance degradation due to occurrence of errors, cache protection using ECC schemes is becoming increasingly popular. Nevertheless, these additional area, power and latency overheads need to be minimized in caches as much as possible.

In this paper, we present Parity++: a novel unequal message protection scheme for last level caches that preferentially provides stronger error protection to certain "special messages". As the name suggests, this coding scheme requires one extra bit above a simple parity Single Error Detection (SED) code while providing SED for all messages and Single Error Correction (SEC) for a subset of messages. Thus, it is stronger than just basic SED parity and has much lower parity storage overhead (3.5X and 4X lower for 32-bit and 64-bit memories respectively) than a traditional Single Error Correcting, Double Error Detecting (SECDED) code. Error detection circuitry often lies on the critical path and is generally more critical than error correction circuitry as error occurrences are rare even with an increasing soft error rate. Our coding scheme has a much simpler error detection circuitry that incurs lower energy and latency costs than the traditional SECDED code. Thus, Parity++ is a lightweight ECC code that is ideal for large capacity last level caches. We also evaluate Parity++ with a memory speculation procedure [3] that can be generally applied to any ECC protected cache to hide the decoding latency while reading messages when there are no errors.

II. Background and Related Work

A. Error Correcting Codes

Error-correcting codes (ECCs) increase the resiliency of communication and storage systems by adding redundant bits (or symbols, but in this work we focus on the binary regime). A code $\mathscr{C}$ can be thought of as an injective mapping of *messages* of length k to *codewords* of length n. Let r be the number of redundant bits, i.e., $r=n-k$. A binary code is considered *linear* if the sum of any two codewords in $\mathscr{C}$ is also a codeword in $\mathscr{C}$.

A linear block code is described by either its $(k \times n)$ *generator matrix* $\mathbf{G}$ or its $(r \times n)$ *parity-check matrix* $\mathbf{H}$, with the relation $\mathbf{G}\mathbf{H}^T = \mathbf{0}$. A particular message $\mathbf{m}$ is encoded to its corresponding codeword $\mathbf{c}$ by multiplying it with the generator matrix as follows: $\mathbf{m}\mathbf{G} = \mathbf{c}$. Each row of $\mathbf{H}$ is a parity-check equation that all codewords must suffice, thus $\mathbf{H}\mathbf{c}^{\mathrm{T}} = \mathbf{0}$. We define the *received vector* at the output of the channel as $\mathbf{y} = \mathbf{c} + \mathbf{e}$, in which $\mathbf{e}$ is the error-vector representing which bits have been flipped. The receiver calculates the *syndrome*, $\mathbf{s} = \mathbf{H}\mathbf{y}^{\mathrm{T}}$, and if $\mathbf{s} \neq \mathbf{0}$, then it is known that the received vector is not a valid codeword. At this point, the decoder can either attempt to determine the most likely originally transmitted codeword or it can simply raise a flag that an error was detected (depending on the system goals and design). We say a code is *systematic* if a message is directly embedded in the codeword, i.e., each message bit is equal to a specific codeword bit.

A useful parameter of a linear code is its *minimum distance*, d_{min}, which is the minimum Hamming distance between any two (non-identical) codewords. Additionally, since a linear

code must include the **0** codeword, the minimum distance of a linear code is simply the minimum weight of any (non-zero) codeword in the code:

$$d_{min} = \min_{\substack{\mathbf{c_1},\mathbf{c_2}\in\mathscr{C};\\ \mathbf{c_1}\neq\mathbf{c_2}}} [d_H(\mathbf{c_1},\mathbf{c_2})] = \min_{\substack{\mathbf{c}\in\mathscr{C};\\ \mathbf{c}\neq\mathbf{0}}} [wt(\mathbf{c})].$$

A linear code guarantees correction of up to $t = \lfloor \frac{1}{2}(d_{min}-1) \rfloor$ bit-errors, or detection of up to $(d_{min}-1)$ bit-errors (without any correction guarantees). For even values of d_{min}, a linear code simultaneously guarantees correction of up to t bit-errors and detection of up to $(t+1)$ bit-errors. Further explanation of the fundamental properties of codes can be found in classic textbooks [4], [5].

B. SRAM Reliability and Error Detection and Correction in Caches

As mentioned before, SRAM reliability concerns are growing. Although the soft error rate of SRAM cell has almost been constant at 10^{-3} FIT/bit [6], [7], the likelihood of a particle striking the array is increasing with increase in size. Most of the recent processors with large capacity caches have ECC protected L2 and/or L3 caches. Some of the common and recent examples include Qualcomm's Centriq 2400 processor [8], AMD's Athlon [9] and Opteron [10] processors as well as IBM Power 4 [11] processors. Most of the commercially available processors use traditional (72,64) SECDED [12] code on each 64-bit word in the cache line. A lot of past works have suggested decoupling error detection and correction mechanisms so as to reduce the complexity and overhead of error detection since that is more critical than error correction. In [13], the authors suggest using SRAM for only error detection and storing the ECC correction bits within the memory hierarchy to reduce the overhead. In another work on ECC in caches, the authors of [14] suggest protecting only those cache lines that have been recently used. Thus, they trade-off protection with area and energy. Some past works like [15] have also focused on ECC protection schemes for L1 cache.

C. Application Characteristics

Data or instructions in applications are generally very structured. Frequencies of instructions in most applications follow power law distribution [16]. This means that some instructions get more frequently accessed than the rest. If the opcode (that primarily determines the action taken by the instruction) in a certain instruction set architecture (ISA) is, for example, the first x bits, then the relative frequency of the opcodes of the common instructions are high. This means most instructions in the memory would have the same prefix of x-bits. Table I shows the fraction of the two most frequently occurring opcode over each of the benchmark suites. The benchmarks were compiled for 32-bit RISC-V (RV32G) [17] instruction set v2.0 were the least significant 7 bits are designated as the opcode. This is true not just for instructions but also for data. In most applications, the data in the memory is usually low-magnitude signed data of a certain data type. However, these values get represented inefficiently, for e.g., 4-byte integer type used to represent values that usually need only 1-byte. Thus, in most cases, the MSBs would be a leading-pad of 0s or 1s. Table I shows that, for a wide range of data sets, most stored data starts with a leading pad of zeros. Our approach of utilizing these characteristics in applications complements recent research on data compression in cache and main memory systems such as frequent value/pattern compression [18], [19], base-delta-immediate compression [20] and bit-plane compression [21]. However, our main goal is to provide stronger error protection to these special messages that are chosen based on the knowledge of data patterns in context.

TABLE I: Fraction of Special Messages per Benchmark Within Suite

	Top Two Most Freq Opcodes (Instruction Memory)		First 6 bits are 0 (Data Memory)	
Benchmark Suite	**Max**	**Mean**	**Max**	**Mean**
AxBench	0.51	0.46	0.92	0.86
SPEC CPU2006	0.56	0.37	0.99	0.89

III. Lightweight Error Correction Code

A. Theory

The code we developed in this work, which we call Parity++, is a type of *unequal message protection* code, in that we *a priori* designate specific messages to have extra protection against errors as can be seen in Figure 1. As in [22], there are two classes of messages, normal (non-special) and special, and they are mapped to normal (or non-special) and special codewords, respectively. When dealing with the importance or frequency of the underlying data, we refer to the messages; when discussing error detection/correction capabilities we refer to the codewords.

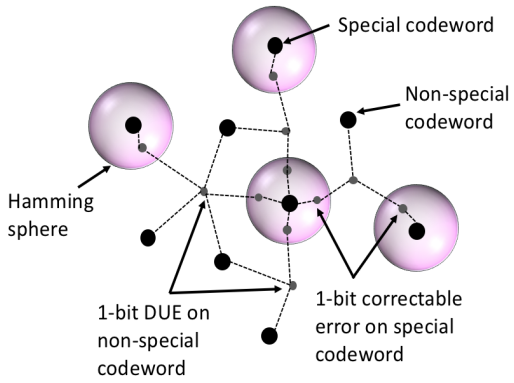

Fig. 1: Conceptual Illustration of Parity++ for 1-bit error

Codewords in Parity++ have the following error protection guarantees: normal codewords have single-error detection; special codewords have single-error correction. Let us partition the codewords in our code $\mathscr{C}$ into two sets, $\mathscr{N}$ and $\mathscr{S}$, representing the normal and special codewords, respectively. The minimum distance properties necessary for the aforementioned error protection guarantees of Parity++ are as follows:

$$\min_{\mathbf{u},\mathbf{v}\in\mathscr{N},\mathbf{u}\neq\mathbf{v}} d_H(\mathbf{u},\mathbf{v}) \geq 2, \tag{1}$$

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

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

A second defining characteristic of the Parity++ code, is that the length of a codeword is only two bits longer than a message, i.e., $n = k+2$. Comprehensive comparisons between Parity++ and other popular ECCs are included in some of the subsequent sections.

For the context of this paper, let us assume that our Parity++ always has message length k as a power of 2. The overall approach to constructing our code is to create a Hamming subcode of a SED code [23]; when an error is detected, we decode to the neighboring special codeword. The overall code has $d_{min}=2$, but a block in $\mathbf{G}$, corresponding to the special messages, has $d_{min}\geq 3$. For the sake of notational convenience, we will go through the steps of constructing the $(34,32)$ Parity++ code (as opposed to the generic $(k+2,k)$ Parity++ code).

We begin by creating the generating matrix for the Hamming code whose message length is at least as large as the message length in the desired Parity++ code; in our case, we use the $(63,57)$ Hamming code. Let α be a primitive element of GF(2^6) such that $1+x+x^6=0$, then our generator polynomial is simply $g_S(x)=1+x+x^6$ (and we construct our generator matrix using the usual polynomial coding methods). We then shorten this code to $(32,26)$ by expurgating and puncturing (i.e., deleting) the right and bottom 31 columns and rows. Now, we add a column of 1s to the end, resulting in a generator matrix, which we denote as $\mathbf{G}_S$, for a $(33,26)$ code with $d_{min}=4$.

For the next step in the construction of the generating matrix of our $(34,32)$ Parity++ code, we add $\mathbf{G_N}$ on top of $\mathbf{G}_S$, where $\mathbf{G_N}$ is the first 6 rows of the generator matrix using the generator polynomial $g_N(x)=1+x$, with an appended row of 0s at the end. Note that $\mathbf{G_N}$ is the generator polynomial of a simple parity-check code. By using this polynomial subcode construction, we have built a generator matrix with overall $d_{min}=2$, with the submatrix $\mathbf{G}_S$ having $d_{min}=4$. At this point, notice that messages that begin with 6 0s only interact with $\mathbf{G}_S$; these messages will be our special messages. Note that Conditions 1 and 3 are satisfied; however, Condition 2 is not satisfied. To meet the requirement, we add a single non-linear parity-bit that is a NOR of the bits corresponding to $\mathbf{G_N}$, in our case, the first 6 bits.

The final step is to convert $\mathbf{G}_S$ to systematic form via elementary row operations. Note that these row operations preserve all 3 of the required minimum distance properties of Parity++. As a result, the special codewords (with the exception of the known prefix) are in systematic form. For example, in our $(34,32)$ Parity++ code, the first 26 bits of a special codeword are simply the 26 bits in the message (not including the leading run of 6 0s).

At the encoding stage of the process, when the message is multiplied by $\mathbf{G}$, the messages denoted as special must begin with a leading run of $\log_2(k)+1$ 0's. However, the original messages we deem to be special do not have to follow this pattern as we can simply apply a pre-mapping before the encoding step, and a post-mapping after the decoding step.

In our $(34,32)$ Parity++ code, observe that there are 2^{26} special messages. Generalizing, it is easy to see that for a $(k+2,k)$ Parity++ code, there are $2^{k-\log_2(k)-1}$ special messages.

B. Error Detection and Correction

We separate the received–possibly erroneous–vector $\mathbf{y}$ into two parts, $\bar{\mathbf{c}}$ and η, with $\bar{\mathbf{c}}$ being the first $k+1$ bits of the codeword and η the additional nonlinear redundancy bit ($\eta=0$ for special messages and $\eta=1$ for normal messages). There are three possible scenarios at the decoder: no (detectable) error, correctable error, or detected but uncorrectable error.

First, due to the Parity++ construction, every valid codeword has even weight. Thus, if $\bar{\mathbf{c}}$ has even weight, then the decoder concludes no error has occurred, i.e., $\bar{\mathbf{c}}$ was the original codeword. Second, if $\bar{\mathbf{c}}$ has odd weight and $\eta=0$, the decoder attempts to correct the error. Since $\mathbf{G}_S$ is in systematic form, we can easily retrieve $\mathbf{H}_S$, its corresponding parity-check matrix. The decoder calculates the syndrome $\mathbf{s_1}=\mathbf{H}_S^T\bar{\mathbf{c}}$. If $\mathbf{s_1}$ is equal to a column in $\mathbf{H}_S$, then that corresponding bit in $\bar{\mathbf{c}}$ is flipped. Third, if $\bar{\mathbf{c}}$ has odd weight and either $\mathbf{s_1}$ does not correspond to any column in $\mathbf{H}_S$ or $\eta=1$, then the decoder declares a DUE (detected but un-correctable error).

The decoding process described above guarantees that any single-bit error in a special codeword will be corrected, and any single-bit error in a normal codeword will be detected (even if the bit in error is η).

Let's take a look at two concrete examples for the $(10,8)$ Parity++ code. Without any premapping, a special message begins with $\log_2(3)+1=4$ zeros. Let our original message be $\mathbf{m}=(00001011)$, which is encoded to $\mathbf{c}=(1011010110)$. Note that the first 4 bits of $\mathbf{c}$ is the systematic part of the special codeword. After passing through the channel, let the received vector be $\mathbf{y}=(1001010110)$, divided into $\bar{\mathbf{c}}=(1001010110)$ and $\eta=0$. Since the weight of c is odd and $\eta=0$, the decoder attempts to correct the error. The syndrome is equal to the 3rd column in $\mathbf{H}_S$, thus the decoder correctly flips the 3rd bit of $\bar{\mathbf{c}}$.

For the second example, let us begin with $\mathbf{m}=(11010011)$, which is encoded to (0011111101). After passing through the channel, the received vector is $\mathbf{y}=(0011011101)$. Since the weight of $\bar{\mathbf{c}}$ is odd and $\eta=1$, the decoder declares a DUE. Note that for both normal and special codewords, if the only bit in error is η itself, then it is implicitly corrected since $\bar{\mathbf{c}}$ has even weight and will be correctly mapped back to $\mathbf{m}$ without any error detection or correction required.

C. Architecture

For a cache with error detection and correction (EDAC) mechanism, there is additional error detection/correction latency. Error detection latency is more critical than error correction as occurrence of an error is a rare event when compared to the processor cycle time and doesn't fall in the critical path. The data/instruction being read from the cache goes through the ECC error detection engine first. If there are no errors then the decoded message moves ahead. In case of an error, the received message goes through an additional correction engine to retrieve the correct message and then the message can be used in the rest of the computation flow.

When using Parity++, the flow almost remains the same. Parity++ can detect all single bit errors but has correction capability for "special messages". When a single bit flip occurs on a message, the error detection engine first detects the error and stalls the pipeline. If the non-linear bit says it is a "special message"(non-linear bit is '0'), the received message goes through the Parity++ error correction engine which outputs the corrected message. This marks the completion of the cache access. If the non-linear bit says it is a non-special message (non-linear bit is '1'), then a DUE is declared and it is checked if the cache line is clean. If so, the cache line is simply read back from the lower level cache or the memory and the cache access is completed. However, if the cache line is dirty and there are no other copies of that particular cache line, it leads to a crash or a roll back to checkpoint. Note that both Parity++

and SECDED have equal decoding latency of one cycle that is incurred during every read operation from an ECC protected cache. The encoding latency during write operation does not fall in the critical path and hence, is not considered in our analyses.

Next in this paper we present a memory speculation scheme that helps to hide the latency incurred by the error detection engine when there are no errors.

1) Memory Speculation: Figure 2 shows the flow of a read operation when the memory speculation scheme is used. The basic idea behind this speculation scheme is to predict the original message from the encoded codeword without having to go through the decoding/error detection circuitry in order to hide the additional latency incurred by the decoding/detection mechanism. While the decoding happens, the predicted instruction/data can move forward to the next stages in the pipeline. If the predicted value is correct, then no action is required and pipeline goes ahead as usual without any additional stalls. In case an error is detected, the mis-predicted instruction or all the dependent instructions that received the mis-predicted data needs to be squashed. This prediction scheme for ECC protected caches is similar to what was proposed in [3] for stronger error protection in on-chip memories.

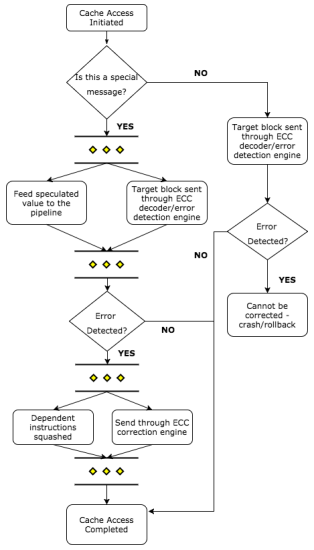

Fig. 2: Flow of read operation in cache with memory speculation and Parity++ protection schemes

This speculation scheme is most effective when the encoded ECC codewords are systematic. When systematic, the original message can be easily retrieved by truncating the additional redundant bits that are generally added to the end of the actual message in case of no errors in the received codeword. Instead of waiting for the decoding to get done, the original message can be speculated by truncating the redundant bits. Thus, the computation moves ahead with the predicted data/instruction without any stalls while the decoding for error detection happens in parallel. A major difference between SECDED and our scheme, Parity++ is that all codewords under SECDED are systematic while only the special messages for Parity++ are systematic. As a result, for Parity++, speculation is used only if the message is special. If not, computation is stalled for one cycle while decoding/error detection happens. Special messages can be distinguished from non-special messages using the non-linear bit.

2) Additional Cache Support for Speculation: Figure 3 depicts the additional circuitry that needs to be added to a traditional cache to support the memory speculation scheme with Parity++.

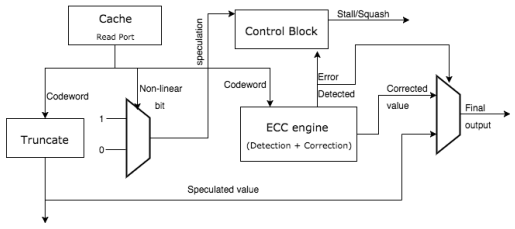

Fig. 3: Cache architecture to implement Parity++ with memory speculation

The non linear bit is first checked. If it is a special message, then speculation is triggered and the speculated value is forwarded to the next stage. This speculated value comprises of the lower 26-bits of the received codeword to which the special prefix is separately appended. Meanwhile, the decoding and the error detection circuitry works in parallel. If an error is detected, the control module initiates a squash operation to squash all the dependant instructions that used the mis-predicted data and the ECC correction engine provides the correct output. The control module also stalls the pipeline when the non linear bit indicates that the message is not special and hence, the codeword is not systematic. Therefore, speculation cannot be used and the pipeline needs to be stalled for one cycle till the original message is decoded. The stall latency is, of course, greater than one cycle when an error is detected and the ECC correction engine needs to be triggered. This additional control module is simple and has minimal overhead in terms of area and energy.

D. Coverage and Overheads

1) Detection/Correction Coverage: As given in Table II single-bit parity detects any single-bit error. Our Parity++ scheme keeps this single-bit error detection guarantee, and additionally provides single-bit error correction for special messages. Also, any 2-bit error on a special message in our Parity++ scheme is guaranteed detectable.

The coverage of SECDED and DECTED codes can be understood from their names. SECDED codes guarantee correction of any single bit error and detection of any double bit error; DECTED codes guarantee correction of any double bit error and detection of any triple bit error.

TABLE II: Error Detection and Correction Coverage for Parity++ along with some widely used ECC schemes

ECC scheme	Error Bits Detected	Error Bits Corrected
Parity- Single Error Detecting (SED)	1	0
Parity++	Special Messages - 2 Non-Special Messages - 1	Special Messages - 1 Non-Special Messages - 0
SECDED	2	1
DECTED	3	2

2) Storage Overhead: Single-error detection requires only a single parity bit; our Pairty++ scheme adds an additional parity-bit for a total of 2. The most efficient SEC code is the Hamming

code. Assuming our message length, k, is a power of 2, then the number of redundancy bits required for the (shortened) Hamming code is $\log(k)+1$. Since the Hamming code has a minimum distance of 3, we can create a SECDED code—the extended Hamming code—with the addition of a single parity bit, yielding a total of $\log(k)+2$ redundancy bits. Similarly, we can use a (shortened) extended BCH code as a DECTED code, with $2\log(k)+3$ redundancy bits. The parity storage overhead of these schemes for different cacheline sizes is given in Figure 4

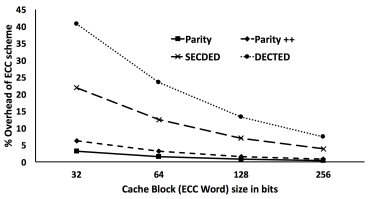

Fig. 4: Storage overhead of different commonly used ECC schemes along with our scheme Parity++

3) Latency and Energy Overhead: The encoding and decoding latencies when writing to/reading from the memory are almost identical for Parity++ and SECDED. They would both require an additional one cycle for each of the two operations. Error correction in case of Parity++ requires an extra matrix multiplication. However, this latency is not critical as occurrence of errors is a rare event compared to the cycle time of the processor. With the proposed memory speculation scheme, SECDED incurs no additional decoding latency when there are no errors. For Parity++ the one cycle extra decoding latency happens only when it is a non special message (only 20-25% of messages are typically non-special).

The encoding energy overhead is almost similar for both Parity++ and SECDED. The decoding energy overheads are slightly different. For SECDED, the original message can be retrieved from the received codeword by simply truncating the additional ECC redundant bits. However, all received codewords need to be multiplied with the H-matrix to detect if any errors have occurred. For Parity++, the original message can be retrieved using truncation when it is a special messages. For the 20-25% non special messages, the non-systematic received codeword needs to be multiplied with a decoder matrix to get the original message. This decoder matrix multiplication, when sythesized using an industrial 45nm library has $\sim$4x higher energy overhead than the H-matrix multiplication of SECDED since the Parity++ decoder is larger than the SECDED H-matrix. However, for Parity++, the error detection scheme is much simpler. It is just a chain of XOR gates and the synthesized detection engine consumes $\sim$10x lower energy than the H-matrix of SECDED required for error detection. For Parity++, all messages go through the chain of XOR gates for error detection and only the non special messages need to be multiplied with the decoder matrix to retrieve the original message. Since the error detection in Parity++ is much cheaper in terms of energy overhead than SECDED and the non special messages only constitute about 20-25% of the total messages, the overall read energy in Parity++ turns out to be much lesser than SECDED. Also, with reduced array size for caches with Parity++ due to lower storage overhead, the leakage energy is also less than that in caches with SECDED.

IV. Experimental Methodology

We evaluated Parity++ over applications from the SPEC 2006 benchmark suite. Two sets of core micro-architectural parameters (provided in Table III) were chosen to understand the performance benefits in both a lightweight in-order(InO) processor and a larger out-of-order(OoO) core. Performance simulations were run using Gem5 [24], fast forwarding for 1 billion instructions and executing for 2 billion instructions.

The first processor is a lightweight single in-order core architecture with a 32kB L1 cache for instruction and 64kB L1 cache for data. Both the instruction and data caches are 4-way associative. The LLC is a unified 1MB L2 cache which is also 8-way associative. The second processor is a dual core out-of-order architecture. The L1 instruction and data caches have the same configuration as the previous processor. The LLC comprises of both L2 and L3 caches. The L2 is a shared 512kB SRAM based cache while the L3 is a shared 2MB cache which is 16-way associative. For both the baseline processors it is assumed that the LLCs (L2 for the InO processor and L2 and L3 for the OoO processor) have SECDED ECC protection.

The performance evaluation was done only for cases where there are no errors. Thus, latency due to error detection is taken into consideration but not error correction as correction is rare when compared to the processor cycle time and doesn't fall in the critical path. In order to compare the performance of the systems with Parity++ against the baseline cases with SECDED ECC protection, the size of the LLCs were increased by $\sim$10% due to the lower storage overhead of Parity as provided in Section III-D. We call this iso-area since the additional area coming from reduction in redundancy is used to increase the total capacity of the SRAM. The iso-area evaluation was done for both with and without memory speculation. The analysis was also done for the iso-capacity where the memory capacity of the systems with Parity++ and SECDED remain same and their performances are measured. As mentioned before, SECDED allows speculation in all cases and thus, incurs no additional read latency due to error detection when there is no error. But for Parity++, only the special messages are systematic and thus, for all non-special messages, there is an additional one cycle read latency due to the error detection circuitry. This additional latency for non-special messages was also taken into consideration for our simulations.

TABLE III: Core Micro-architectural Parameters

	Processor-1	Processor-2
Cores	1, InO (@ 2GHz)	2, OoO (@ 2GHz)
L1 Cache per core	32KB I$ 64KB D$ 4-way	32KB I$ 64kB D$ 4-way
L2 Cache	1MB (unified) 8-way	512KB (shared, unified) 8-way
L3 Cache	-	2MB (shared) 16-way
Cache Line Size	64B	64B
Memory Configuration	4GB of 2133MHz DDR3	8GB of 2133MHz DDR3
Nominal Voltage	1V	1V

V. Results and Discussion

In this section we discuss the performance results obtained from the Gem5 simulations (as mentioned in Section IV). Figures 5 and 6 show the comparative results for the two

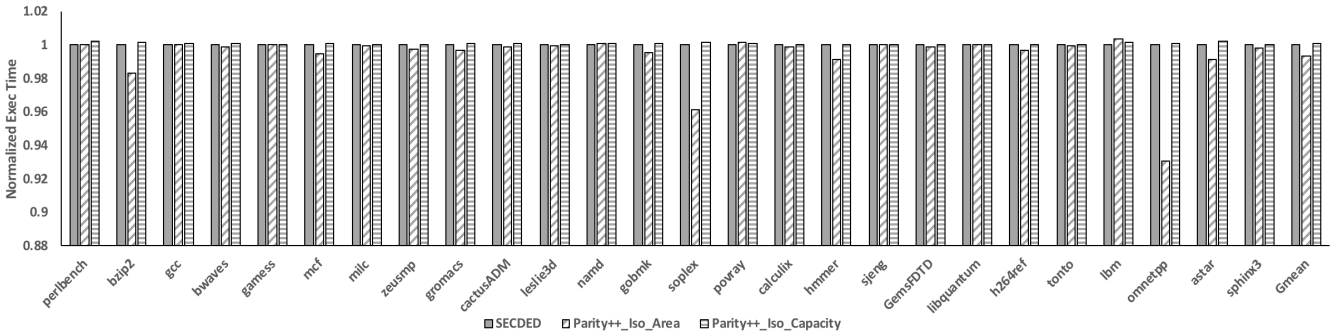

Fig. 5: Comparing Normalized Execution Time of Processor-I with SECDED and Parity++ (with memory speculation)

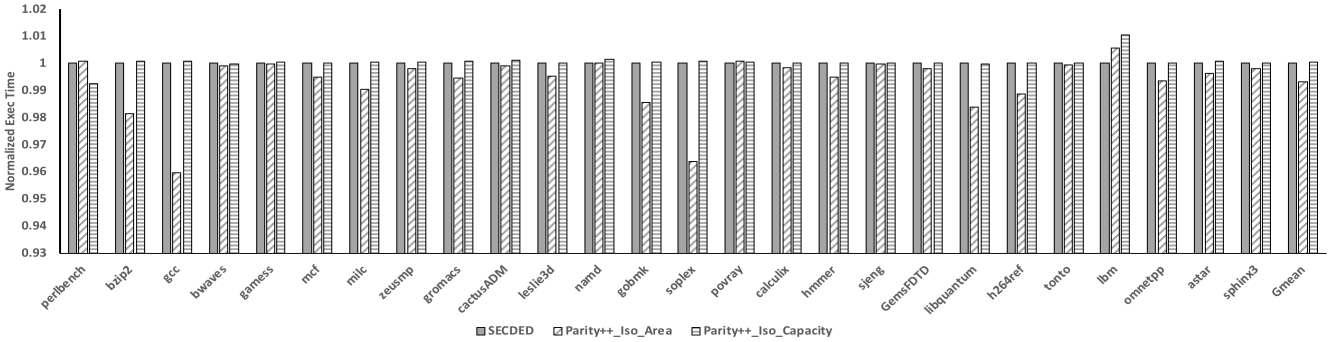

Fig. 6: Comparing Normalized Execution Time of Processor-II with SECDED and Parity++ (with memory speculation)

different sets of core micro-architectures across a variety of benchmarks from the SPEC2006 suite when using memory speculation. In both the evaluations, performance of the system with Parity++ was compared against that with SECDED. The evaluation was further split into iso-area and iso-capacity as explained in Section IV.

For both the core configurations, the observations for the iso-area case are almost similar. With memory speculation it is seen that with additional memory capacity for iso-area, the system with Parity++ has upto $\sim$4% better performance (lower execution time) than the one with SECDED. This improvement in performance happens in spite of the additional one cycle latency incurred on non special messages in the case of Parity++. The applications showing higher performance benefits are mostly memory intensive. Hence, additional cache capacity with Parity++ reduces overall miss rate to an extent such that the slight increase in average LLC hit time gets offset. For most of these applications, this performance gap widens as the LLC size increases for Processor-II. The applications showing roughly similar performances on both the systems are the ones which already have a considerably lower LLC miss rate. As a result, increase in LLC capacity due to Parity++ doesn't lead to a significant improvement in performance. The same evaluation was also done for the case where there is no memory speculation, i.e., both Parity++ and SECDED protected caches have additional hit latency of one cycle for all read operations (figure ommitted due to space constraints). The results show that with the exact same hit latency, Parity++ has upto 7% lower execution time than SECDED due to additional memory capacity.

A more significant result is the iso-capacity case with memory speculation. It is seen that even with additional one cycle latency for non special messages in Parity++, the performance of the system with Parity++ is at par with that of SECDED. This means that by using our lightweight error correction scheme, we manage to save about 5-9% last level cache area (excluding decoder and peripheral circuit area) with negligible hit in performance. Since the LLCs constitute more than 30% of the processor chip area, the cache area savings translate to a considerable amount of reduction in the chip size. This additional area benefit can either be utilized to make an overall smaller sized chip or it can be used to pack in more compute tiles to increase the overall performance of the system.

Parity++ in Lightweight Approximation-Friendly Embedded Memory

Instead of limiting ourselves to last level on-chip caches, we extended the evaluation to on-chip memories in embedded devices. Embedded systems at the edge of the Internet-of-Things (IoT) is driven by the need for low cost and low energy consumption. On-chip memories in these lightweight embedded systems consume a significant portion of system energy. As a result, having strong error correction schemes like SECDED or ChipKill [25] is too costly, in terms of overheads, for such devices. Based on the iso-capacity results, Parity++ (with 3.5X lower parity storage overhead than SECDED in a 32-bit memory) seems to be a good fit for SRAM based embedded memories. Since Parity++ helps in reducing area (in turn reducing SRAM leakage energy) and also has lower error detection energy, it provides a better protection mechanism in such devices than SECDED. It is also stronger than a single-error detecting (SED) Parity code and hence can reduce the number of crashes/hangs when there is a single bit flip in the memory.

Most of the applications that run on these low-cost IoT devices are approximation-tolerant. Hence, we analyzed the benefits of using Parity++ in such devices on 6 applications from AxBench [26], an approximate benchmark suite. The

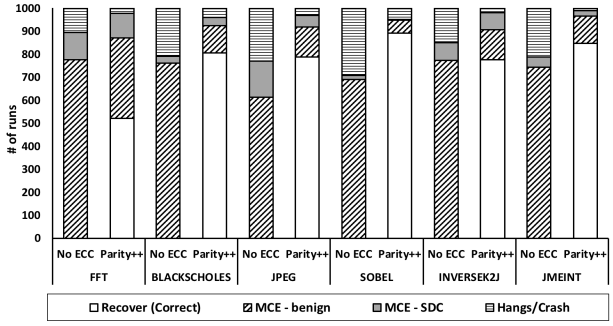

Fig. 7: Output Quality of AxBench benchmarks for memory with no ECC vs with Parity++

AxBench benchmarks were compiled for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [17] using the official tools [27]. Each benchmark was ran till completion 1000 times on top of the RISC-V proxy kernel [28] using the Spike simulator [29] that was modified to produce representative memory access traces. For each run, a single bit error was randomly injected on a demand data memory read. We compared Parity++ against the case when there is no ECC protection and the program continues with erroneous message. In case of non-special messages in Parity++, even though a single bit flip is detected, the program continued with the wrong message instead of crashing immediately since these applications are approximation-tolerant. The results are shown in Figure 7. It can be seen that Parity++ reduces intolerable Silent Data Corruption (SDC), that is, an SDC with more than 10% output error, by upto 84.2%(avg. 32.5%). It significantly reduces the number of crashes/hangs by upto 95.3%(avg. 85.6%). This means Parity++ not only improves the quality of output, the system will be much more resilient to hangs/crashes in case of unpredictable single bit flips during runtime.

VI. Conclusion

In this work, we present a novel lightweight error protection scheme, Parity++, for last level caches based on unequal message protection. From our analysis, we find that about 80% of messages/words have same prefix bits (leading 0's) and we denote these as special messages. For a 64 bit word, Parity++ uses only 2 additional redundant bits and provides SECDED protection for these special messages while providing only SED for the non-special messages. In iso-area evaluations, up to about 4% performance benefit can be obtained, while iso-capacity evaluations showed almost negligible ($<$0.2% in all but one case) performance degradation with $\sim$9% lower storage overhead than a traditional SECDED scheme which translates to about 5% cache area savings.

Acknowledgment

The authors thank the anonymous reviewers for their constructive feedback. This work was supported by the 2016 USA Qualcomm Innovation Fellowship. The authors thank Dr. Greg Wright from Qualcomm Research for his feedback and guidance of this work.

References

[1] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt, "Cache scrubbing in microprocessors: myth or necessity?," in *10th IEEE Pacific Rim International Symposium on Dependable Computing, 2004. Proceedings.*, pp. 37–42, March 2004.

[2] J. Yan and W. Zhang, "Evaluating instruction cache vulnerability to transient errors," in *Proceedings of the 2006 Workshop on MEmory Performance: DEaling with Applications, Systems and Architectures*, MEDEA '06, (New York, NY, USA), pp. 21–28, ACM, 2006.

[3] H. Duwe, X. Jian, and R. Kumar, "Correction prediction: Reducing error correction latency for on-chip memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 463–475, Feb 2015.

[4] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes*. Elsevier, 1977.

[5] S. Lin and D. J. Costello, *Error control coding*, vol. 2. Prentice Hall Englewood Cliffs, 2004.

[6] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of multi-bit soft error events in advanced srams," in *IEEE International Electron Devices Meeting 2003*, pp. 21.4.1–21.4.4, Dec 2003.

[7] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Transactions on Device and Materials Reliability*, vol. 5, pp. 397–404, Sept 2005.

[8] "Qualcomm Centriq 2400 Processor."

[9] J. Huynh, "White Paper: The AMD Athlon MP Processor with 512KB L2 Cache," tech. rep., May 2003.

[10] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, pp. 66–76, March 2003.

[11] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, pp. 5–25, Jan 2002.

[12] M. Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, pp. 395–401, July 1970.

[13] D. H. Yoon and M. Erez, "Memory mapped ecc: Low-cost error protection for last level caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 116–127, ACM, 2009.

[14] S. Kim and A. K. Somani, "Area efficient architectures for information integrity in cache memories," *SIGARCH Comput. Archit. News*, vol. 27, pp. 246–255, May 1999.

[15] N. N. Sadler and D. J. Sorin, "Choosing an error protection scheme for a microprocessor's l1 data cache," in *2006 International Conference on Computer Design*, pp. 499–505, Oct 2006.

[16] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software-defined ECC: Heuristic recovery from uncorrectable memory errors," tech. rep., University of California, Los Angeles, Oct. 2017.

[17] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.0," 2014.

[18] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pp. 258–265, 2000.

[19] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," 2004.

[20] 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 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 377–388, ACM, 2012.

[21] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 329–340, IEEE Press, 2016.

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

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

[24] 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," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

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

[26] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design Test*, vol. 34, pp. 60–68, April 2017.

[27] Q. Nguyen, "RISC-V Tools (GNU Toolchain, ISA Simulator, Tests) – git commit 816a252."

[28] A. Waterman, "RISC-V Proxy Kernel – git commit 85ae17a."

[29] A. Waterman and Y. Lee, "Spike, a RISC-V ISA Simulator – git commit 3bfc00e."