Reducing Cache Power with Low-Cost, Multi-bit Error-Correcting Codes

Chris Wilkerson, Alaa R. Alameldeen, Zeshan Chishti,
Wei Wu, Dinesh Somasekhar, and Shih-Lien Lu

Intel Labs
Hillsboro, Oregon, USA

{chris.wilkerson, alaa.r.alameldeen, zeshan.a.chishti, wei.a.wu, dinesh.somasekhar, shih-lien.l.lu} @intel.com

ABSTRACT

Technology advancements have enabled the integration of large on-die embedded DRAM (eDRAM) caches. eDRAM is significantly denser than traditional SRAMs, but must be periodically refreshed to retain data. Like SRAM, eDRAM is susceptible to device variations, which play a role in determining refresh time for eDRAM cells. Refresh power potentially represents a large fraction of overall system power, particularly during low-power states when the CPU is idle. Future designs need to reduce cache power without incurring the high cost of flushing cache data when entering low-power states.

In this paper, we show the significant impact of variations on refresh time and cache power consumption for large eDRAM caches. We propose Hi-ECC, a technique that incorporates multi-bit error-correcting codes to significantly reduce refresh rate. Multi-bit error-correcting codes usually have a complex decoder design and high storage cost. Hi-ECC avoids the decoder complexity by using strong ECC codes to identify and disable sections of the cache with multi-bit failures, while providing efficient single-bit error correction for the common case. Hi-ECC includes additional optimizations that allow us to amortize the storage cost of the code over large data words, providing the benefit of multi-bit correction at same storage cost as a single-bit error-correcting (SECDED) code (2% overhead). Our proposal achieves a 93% reduction in refresh power vs. a baseline eDRAM cache without error correcting capability, and a 66% reduction in refresh power vs. a system using SECDED codes.

Categories and Subject Descriptors

B.3.4 [**Memory Structures**]: Reliability, Testing, Fault-Tolerance.

General Terms

Design, Reliability, Power.

Keywords

ECC, Multi-Bit ECC, DRAM, eDRAM, refresh power, Vccmin.

1. INTRODUCTION

Advances in technology scaling have led to dramatic yearly improvements in on-die cache capacity. New process technologies have also enabled integrating DRAM on a logic process, leading to the use of embedded DRAM (eDRAM) to build on-die caches that are much denser than SRAM-based caches (e.g., IBM Power 7 [14]). However, a side effect of technology scaling is the increasing susceptibility of cache structures to device variations [1, 27], where a few weak cells can constrain the operating range of the whole cache.

In traditional SRAM caches, intrinsic variations force operation at high voltages due to a few weak cells that fail at lower voltages, and impede efforts to reduce power [29, 33]. Likewise, in eDRAM caches, device variations affect the retention time of individual DRAM cells, with a few particularly weak bits determining the refresh time of the whole cache. A high refresh rate significantly increases cache power.

Reducing power consumption is a first-order design constraint for modern processors. In pursuit of improved power and energy efficiency, processors implement a number of idle states to support lower power modes. Reducing the power consumed during idle states is particularly important because the typical CPU spends the vast majority of its time in idle state. Many desktop applications, such as word processors and spreadsheets, spend much of the time waiting for I/O and tend to require the CPU to operate only 10-20% of the time during use. Studies done on the Intel® Core™/Core™ 2 Duo show that an idling processor will consume an average of 0.5W-1.05W [24] depending on the processor and frequency of idle state exits caused by events like OS interrupts. Our analysis projects that a future processor with 128MB of eDRAM cache will consume 926mW just refreshing the eDRAM. Based on these power numbers, we project that the power consumption of large memory structures, like eDRAM caches, will be the biggest contributor to overall idle power.

One popular method to reduce cache power is to power-gate large blocks of memory at the cost of losing its state [6]. But as cache density increases, the performance and power costs of this approach also increase. One of the important goals in implementing idle states is reducing power consumption, while minimizing the transition latency into and out of the idle state. In the Intel® Core™ 2, a transition out of the C4 idle state can take about 100-200us [9]. In contrast, if the state in a 128MB eDRAM cache were sacrificed to save power, it would take 4.2ms to re-fetch the 128MB of lost data assuming full usage of the 30GB/s bandwidth provided by system memory. In future products with denser eDRAM caches, navigating the tradeoffs between idle exit

latency and idle power consumption will become increasingly difficult. As the capacity of embedded memory grows, the performance and power cost of flushing this memory also grows. A key challenge for future product designers is to enable flexible memory structures that can operate at very low idle power, without dramatically increasing transition latency to and from the idle power state due to data loss.

In this paper, we evaluate a modern processor with a 128MB eDRAM cache. We show that refresh time plays the key role in determining the eDRAM's power. We first explore the role of variation-related cell failures in determining refresh time. We then evaluate the potential for using error-correcting codes (ECC) to mitigate refresh-related failures. Augmenting eDRAM with error-correcting codes enables reliable cache operation with longer refresh periods, thereby lowering system power.

We propose Hi-ECC, a practical, low-latency, low-cost, error-correcting system that can compensate for high failure rates in eDRAM caches. Hi-ECC implements a strong BCH code with the ability to correct 5 and detect 6 errors (hereafter referred to as a 5EC6ED code). A traditional approach using strong ECC suffers from two prohibitive overheads that limit its applicability. First, building a low-latency decoder for multi-bit ECC codes is extremely costly. Second, the storage overhead of ECC bits is high (around 10% for a 5EC6ED ECC code for a 64 byte line). Hi-ECC proposes architectural solutions to both problems. It uses a simple ECC decoder optimized for the 99.5% of the lines that require little or no correction, and provides a high latency alternative for lines that require complex multi-bit correction. To minimize the performance impact of processing high latency multi-bit corrections, Hi-ECC disables lines with multi-bit failures. Finally, Hi-ECC leverages the natural spatial locality of the data to reduce the cost of ECC storage. We make the following main contributions:

1. We demonstrate that device variations lead to significant increases in cache refresh rates.

2. We propose Hi-ECC, a practical system for using strong error correcting codes that avoids decoder complexity and latency.

3. We show how Hi-ECC can be extended to reduce the storage overhead of the error-correcting codes by amortizing the cost of the code over larger data words. This allows implementing Hi-ECC with a 2% storage overhead, comparable to that of a single error correcting code (SECDED) over 64 byte lines.

4. For a system with a 128 MB eDRAM cache, we show that Hi-ECC can reduce cache refresh power by 93% compared to an eDRAM with no error correction capability, and 66% compared to an eDRAM with SECDED, all for about the same storage overhead as a SECDED code. When accounting for dynamic power, an optimized Hi-ECC reduces total power by 61% relative to SECDED.

The remainder of this paper is organized as follows. In Section 2, we review some of the design tradeoffs for eDRAM caches, including a discussion of retention failures and previous work on mitigating them. Section 3 describes our proposed Hi-ECC architecture, and is followed by a review of the mathematical properties of BCH codes that Hi-ECC relies on in Section 4. We

describe our evaluation methodology and results in Section 5 and conclude in Section 6.

2. BACKGROUND

Embedded DRAM technology enables smaller memory cells as compared to SRAM cells, resulting in a three to four times increase in memory density [21]. The higher density of eDRAM makes it a promising candidate to replace SRAM as the last-level on-chip cache in future high performance processors. IBM has recently announced that its upcoming Power7 processor will use a 32 MB on-chip eDRAM cache [14]. As feature sizes continue to decrease, even larger eDRAM caches can be incorporated on chip. In this paper, we model a 128 MB eDRAM cache, two technology generations ahead of IBM Power7's eDRAM cache.

One of the main problems with eDRAM cells is that they lose charge over time due to leakage currents. The retention time of an eDRAM cell is defined as the length of time for which the cell can retain its state. Cell retention time depends on the leakage current, which, in turn, depends on the access device leakage. To preserve the state of stored data, eDRAM cells need to be refreshed on a periodic basis. In order to prevent failures, the refresh period needs to be less than the cell retention time. Because eDRAM uses fast logic transistors with a higher leakage current than conventional DRAM, the refresh time for eDRAM is about a thousand times shorter than conventional DRAM. For example, Barth, et al. [2] report the refresh time of a small eDRAM to be 40us as compared to 64ms refresh time in commodity DRAM [22]. This low refresh time poses serious problems for eDRAM because it not only increases the idle power, but also leads to reduced availability. Previous work has shown that variations in threshold voltage cause retention times of different DRAM cells to vary significantly [8, 10, 11, 18]. These variations are caused predominantly by random dopant fluctuations and manifest themselves as a random distribution of retention times amongst eDRAM cells. We use the data published in [18] to model the retention time distribution in Figure 1.
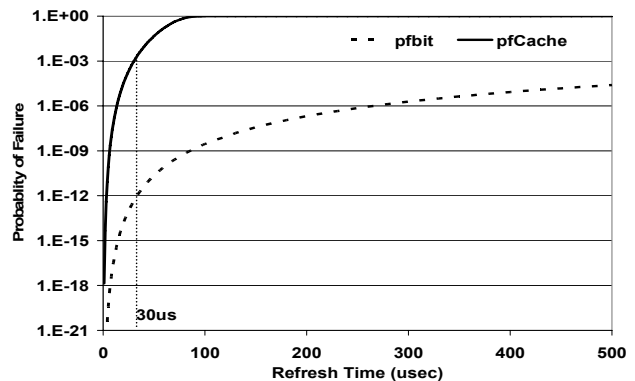

Figure 1: eDRAM retention time distribution

The pfbit curve in Figure 1 represents the probability of a retention failure in a single bit cell (derived from Figure 3 in [18]) and the pfCache curve represents the failure probability of a 128MB eDRAM cache for different refresh times. Our model assumes that bit retention failures are distributed randomly throughout the eDRAM cache, consistent with [18]. A cache containing even a single failure must be discarded. Therefore, the

probability of failure for the entire cache is (1 – probability of success), where the probability of success is the probability that each bit in the cache stays failure-free. We assume that the pfCache must be kept at less than 1 out of 1000 to achieve acceptable manufacturing yields [33]. Under these assumptions, data in Figure 1 shows that the refresh time for a baseline 128MB eDRAM cache is 30 microseconds, close to the 40 microseconds refresh time reported in [2] for a 13.5Mbit eDRAM macro.

Refresh mechanisms in eDRAM designs typically use a single worst case refresh period dictated by the cell with the lowest retention time. As eDRAM capacity increases in future generations, eDRAM idle power, dominated by refresh, will grow. Some previous papers have proposed hardware mechanisms to exploit retention time variations by refreshing different DRAM cells at different refresh rates [16, 26]. Venkatesan, et al. [32] proposed a software mechanism that allocates DRAM pages with longer retention time before allocating pages with shorter retention time, and then chooses a refresh period that is determined only by the populated pages instead of the entire DRAM. These approaches require additional storage to track retention times and rely on memory tests to identify marginal bit cells. In [33] Wilkerson et al propose the bit-fix algorithm, another testing-based approach, to address the problem of high failure rates in the context of Vccmin reduction in SRAM caches instead of DRAM refresh time. Since test time grows proportionately with the capacity of the memory being tested, increasing cache capacities may limit the applicability of all testing-based approaches.

Ghosh and Lee [10] recently proposed a SmartRefresh technique to reduce refresh power by adding timeout counters in each DRAM row and avoiding unnecessary refreshes for those rows which were recently read or written. However, SmartRefresh is ineffective during the idle mode when the cache is not being accessed, and therefore does not improve idle power.

Another promising approach to increase DRAM refresh times is the use of error-correcting codes (ECC) to dynamically identify and repair bits that fail [8, 15]. This approach sets refresh time irrespective of the weakest bits, using ECC to compensate for failures. With this approach, a stronger error-correcting code, with the ability to correct multiple bits, implies increased refresh time and reduced power. However, strong ECC codes have a high storage and complexity overhead which limit their applicability. In the following two sections, we propose an architectural mechanism that uses strong ECC codes with a low storage and complexity overhead.

3. STRONG ECC ARCHITECTURE

When designing a large eDRAM cache, a designer strives to minimize eDRAM power consumption in the low-power operating modes without penalizing performance in the normal operating mode. To achieve this objective, we propose Hi-ECC, which implements a multi-bit error-correcting code with very small area, latency, and power overheads.

We propose a system with a large (128MB) eDRAM last level cache. In a baseline configuration with no error correction capability, the time between refreshes for such a cache will be 30 microseconds, leading to a significant amount of power consumed even when the processor is idle. Refresh power can be reduced by

flushing and power gating the cache during the low-power operating modes. This, however, causes a significant performance penalty when the processor wakes up from the idle mode since it will need to reload the cache, thereby incurring a large number of cold start misses. Alternatively, we can lower refresh power consumption by decreasing the refresh frequency (i.e., increasing time between refreshes). However, as we show in Figure 1, decreasing refresh frequency implies the need to tolerate a higher number of failures for each cache line. Implementing a strong error-correcting code with the capability to correct multiple errors is necessary to achieve this goal.

At the core of Hi-ECC is a strong 5EC6ED (five bit correction, six bit detection) BCH code. We explain implementation details for BCH codes in Section 4. Traditional implementations of a 5EC6ED BCH code would suffer from two key drawbacks: high storage overhead for the code itself, and high decoder complexity and latency. In this section, we describe how Hi-ECC addresses both of these drawbacks. Since our implementation requires architectural changes that would increase dynamic power, we also propose an architectural optimization to lower the impact on the cache dynamic power.

3.1 Reducing Storage Overhead

The storage required for a 5EC6ED code for a 64B cache line is 51 bits, a 10% overhead. Since the cache occupies a large portion of the die area (50% or higher), augmenting the eDRAM cache with 5EC6ED code will significantly increase the die area and cost. In contrast, a single error correcting, double error detecting (SECDED) code for a 64B line requires 11 bits, an overhead of around 2%. Our goal is to implement the 5EC6ED code with the same storage overhead as the SECDED code.

To achieve this goal, we leverage two important properties. First, the size of an ECC code relative to that of the data word diminishes as the size of the data word grows, as we show in Section 4.3. While a SECDED code for a 64B line has an 11-bit overhead (2%), a SECDED code for a 1KB line has a 15-bit overhead (0.18%). Second, the efficacy of a code only diminishes slightly as the size of the data word increases. In Figure 2, we show the failure probability (i.e., the probability that the line will have more failures than those correctable by ECC) for three different codes: SECDED on a 64B line, SECDED on a 1KB line, and double error correcting, triple error detecting code (DECTED) on a 1KB line. At very low refresh times, failure rates of the SECDED 64B line and the SECDED 1KB line are very close. DECTED-1KB (with only a 29-bit overhead, 0.35%) has a lower failure probability than both SECDED codes, except at high refresh times, such as 500us, where it is very close to the SECDED-64B code. By choosing a stronger code and amortizing the cost of the code over larger cache lines, we can improve our ability to tolerate failures with a very small storage overhead.

Our Hi-ECC design implements a 5EC6ED code on each 1KB line (5EC6ED-1KB), requiring an additional 71 bits (0.87% overhead) for each line to store the code. In Figure 3, we compare the refresh time of a 128MB cache augmented with Hi-ECC to the baseline configuration with no error correction capability as well as a configuration using a SECDED code for each 64B sub-block (SECDED-64B). Like previous work that focused on SRAM [33], we assume that the refresh time will be chosen such that no more than 1E-03 (i.e., 1/1000) of the caches will fail. The baseline configuration with no failure mitigation must operate at the

baseline refresh time of 30us. Adding a SECDED code allows a 5X increase in refresh time to 150us. Hi-ECC allows us to increase the refresh time to 440us (almost a 15X reduction in refresh frequency compared to the baseline).
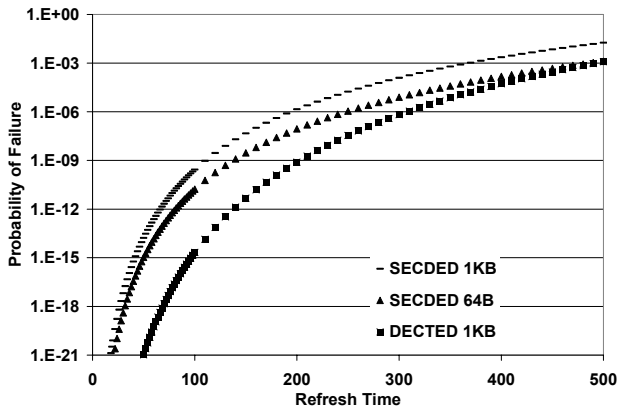

Figure 2. Comparing bit failure probabilities for three code/line size combinations. DECTED on 1KB lines achieves higher refresh time than SECDED on 64B lines
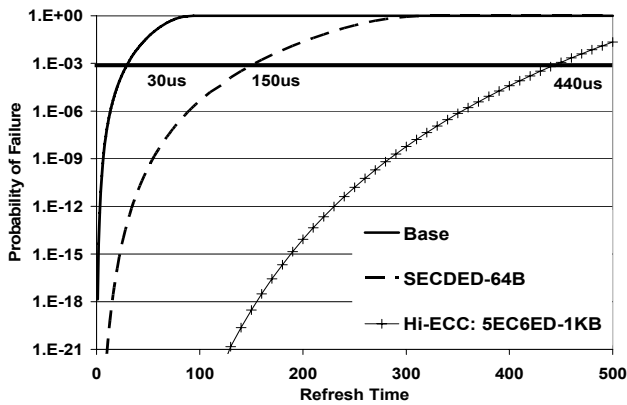

Figure 3. Hi-ECC achieves a higher refresh time than SECDED at the same failure probability

3.2 Reducing Latency

A hardware implementation of a 5EC6ED code is very complex and imposes a long decoding latency penalty, proportional to both the number of error bits corrected and the number of data bits (Section 4.1). If the full strength encoding/decoding was required for every cache access, this could significantly increase cache access latency. However, our proposal leverages the fact that error-prone portions of the cache can be disabled, avoiding the high latency of decode during typical operation.

The Hi-ECC technique relies on a simple, one cycle ECC block to correct a single bit error, and an un-pipelined, high-latency ECC processing block to correct multiple bit failures in a cache line [7, 20]. When a line is read from the cache, a simple decoder generates the syndrome for the line, which includes information on whether it has zero, one, or a higher number of errors (Section 4.2). If the line has zero or one bit failures, the simple ECC decoder can perform the correction in a single cycle. Figure 4 shows a high-level block diagram for Hi-ECC. The block referred

to as Quick-ECC contains the syndrome generation logic and the error correction logic for lines with zero or one failures. The Quick-ECC block also classifies lines into two groups based on the syndrome: those that require complex multi-bit error correction and those that have zero or one errors. Lines that require multi-bit error correction are forwarded to a high latency (potentially hundreds of cycles) ECC processing unit that performs error correction using either software or a simple state machine. This allows us to simplify the design at the expense of increased latency for lines with two or more failures. Lines that require one or less error corrections can be immediately corrected and forwarded to the unit requesting the line.
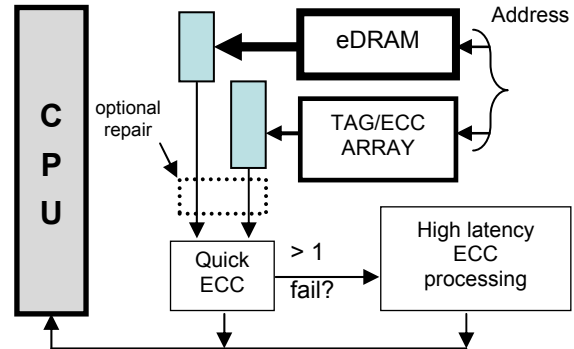

Figure 4: Block diagram for Hi-ECC

The high latency of handling multi-bit failures could significantly reduce performance. To avoid incurring this latency, problematic lines could be completely disabled or a mechanism such as bit-fix [33] could be integrated as shown by the dotted box labeled *optional repair* in Figure 4. This guarantees that the performance penalty of multi-bit decoding is incurred only once, the first time a failure is identified. The frequency of failures plays a role in the disable strategy that we choose. Low multi-bit failure rates motivate a simple approach such as disabling cache lines containing multi-bit failures. On the other hand, cache line disable will result in unacceptable cache capacity loss if multi-bit failure rates are high. In this case, a more complex mechanism such as bit-fix might be used to minimize the capacity lost to disabling.

Figure 5 shows the probability that N (X-axis) or more lines have multi-bit failures for a 128MB eDRAM cache at the refresh time we propose (440us). On average, a 128MB eDRAM will have 750 1KB lines with multi-bit failures that need to be disabled, (0.573% of all lines). As highlighted in the figure, the probability that 900 or more lines (0.7% of all cache lines) will exhibit multi-bit errors is 6.77×10^{-8}. For comparison, Hi-ECC augmented with a simplified version of bit-fix [33] that repairs a single additional bit per cache line requires an additional 13 bits per line (0.13% overhead). However, this 13-bit overhead enables efficient correction of lines with 2-bit errors and reduces the number of lines that need to be disabled. Disabling lines with only 3 or more errors reduces the average number of disabled lines from 750 to 28 (0.02% of all lines), with a probability of 5.95×10^{-8} that 60 or more lines contain three or more errors. Although adding bit-fix reduces wasted cache capacity, the improvement over simple line disable is marginal for the failure rates in our model and doesn't justify the additional latency and complexity. As a result, the rest

of this paper will focus on the Hi-ECC approach that relies solely on line disable for lines with multi-bit (two or more) errors.

Due to the implementation of our eDRAM cache, there are restrictions to how many and which lines can be disabled. Since our cache is a 16-way set-associative, and since disabling all lines in a particular cache set could be catastrophic for some workloads, we limit the maximum number of lines that can be disabled in a particular set to 14 of the 16 ways. We also limit the maximum number of failing lines to 900 to quantify overhead. With a refresh time of 440us, the probability of at least one of the lines containing more than 5 failing bits is 6.21×10^{-4} (Figure 3), the probability of more than 900 multi-bit failures (disabled lines) is 6.77×10^{-8} (Figure 5), and the probability of more than 14 multi-bit failures in a single set is 1.12×10^{-61}. This indicates that disabling cache lines with multi-bit failures will have little effect on the overall probability that our cache meets the quality requirements at 440us. The total storage overhead for our approach is 1.58% including a 0.88% overhead for the code and a single per-line disable bit, and a 0.7% overhead for the 900 disabled lines.
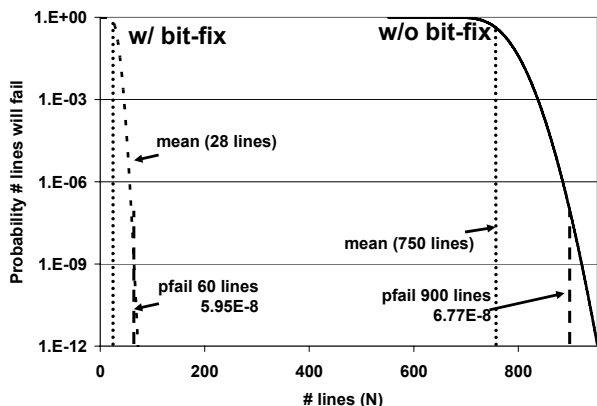

Figure 5. The distribution of failing lines for a 128MB Cache with 1KB lines with (w/) and without (w/o) bit-fix.

3.3 Reducing Dynamic Power

Our Hi-ECC proposal uses larger cache line sizes to reduce the area cost of strong ECC codes. However, larger line sizes introduce some additional challenges. Although 1KB is a reasonable line size for a large embedded memory, our baseline configuration has a much smaller L2 cache with a 64B cache line (referred to as sub-block). Some implementation issues arise when we read from or write to our large L3 eDRAM cache due to the mismatch between its 1KB line size and the 64B sub-blocks used by other caches.

Most writes to the large L3 eDRAM cache will be in the form of smaller 64B sub-blocks generated at lower-level caches or fetched from memory. To modify a 64B sub-block in a 1KB line, we need to perform a read-modify-write operation since we need to compute the ECC code. First, the old 64B block that is being overwritten must be read, along with the ECC code for the entire line. We then use the old data, old ECC, and new data to compute the new ECC for the whole 1KB line. We then write the new 64B sub-block and the new ECC. However, we do not need to read the

whole 1KB line to compute the new ECC, as explained later in Section 4.3.

The purpose of most L3 reads will be to provide cache lines for allocation in lower-level caches. Processing any sub-block requires the ECC code to be processed with the entire data word (1KB cache line) that it protects. Since each 64B sub-block must be checked, each reference to a 64B sub-block must be accompanied by a reference to the surrounding 64B sub-blocks. This implies that any L3 read will access all 16 sub-blocks in the 1KB line, as well as the ECC code that they share. As an example, if we need to read eight out of the 16 sub-blocks in one 1KB line, we must read all 16 sub-blocks eight times, for a total of 128 sub-block reads. This large number of additional reads causes a substantial increase in dynamic power consumption and a drastic reduction in the useful bandwidth delivered by the memory.

To address the extra power overhead for L3 reads, we consider the fact that the vast majority of eDRAM failures are retention failures. Since the retention time of our baseline eDRAM is 30us, and each read automatically implies a refresh, we know that retention failures will not occur for 30us after a line has been read. Our proposal leverages this property and also the temporal and spatial locality of the data to minimize the number the superfluous reads. Using a structure we refer to as the *Recently Accessed Lines Table* (RALT), we attempt to track lines that have been referenced in the last 30us.

The first read to a line causes all sub-blocks in the line to be read and checked for failures. The address of the line is then placed in the RALT to indicate that it has recently been checked and will remain free from retention failures for the next 30usec. As long as the address of the line is held in the RALT, any sub-block reads from the line can forgo ECC processing and thus avoid reading the ECC code and other sub-blocks in the line. To operate correctly, the RALT must ensure that none of its entries are more than 30us old. To guarantee this, each 30us is divided into four equal periods (P0, P1, P2, P3). Entries allocated in the RALT during each period are marked with a 2-bit identifier to specify the allocation period. Transitions between periods, P0 to P1 for example, will cause all RALT entries previously allocated in P1 to be invalidated.

Each entry in the RALT consists of the following fields: a line address to identify the line the entry is associated with; a valid bit, a 2-bit period identifier field to indicate in which of the four periods the line was allocated (P0, P1, P2, P3); and a 16-bit parity consisting of one parity bit for each 64B sub-block in the line. The RALT is direct mapped, but supports a CAM invalidate on the 2-bit period field to allow bulk invalidates of RALT entries during period transitions.

Figure 6 compares the implementation of the baseline L3 protection scheme (top) with that of Hi-ECC (bottom). The baseline scheme uses a separate tag for each 1KB line and a separate SECDED code for each 64B sub-block. To read a 64B block, first the ECC and the block itself are read, then the ECC is processed. In our Hi-ECC technique, the first time a sub-block is read the entire ECC code is read along with each sub-block in the 1KB line to allow ECC processing for a single 64B block. We update the RALT with the line address of the referenced line, a 2-bit period ID, and a single parity bit for each sub-block. After the

first hit to a line, future accesses to the same 1KB line within the next 30us should hit in the RALT. Figure 7 demonstrates a RALT hit. In most cases, only the requested 64B sub-block is read. Parity for the 64B sub-block is computed and compared to the parity stored in the RALT. If the parity matches, we infer that sub-block is valid and forward it to the requesting cache or processor. A parity mismatch is treated as a RALT miss where the whole 1KB line needs to be read.
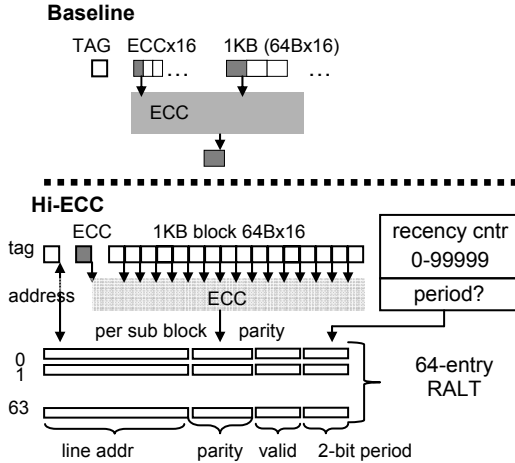

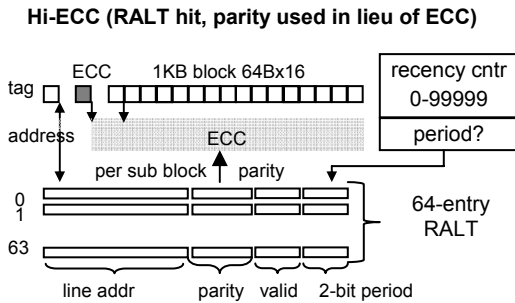**Figure 6: Initial read and update of the Recently Accessed Lines Table (RALT).**

Figure 7: Hits to the Recently Accessed Lines Table (RALT)

3.4 Summary

In this section, we described our L3 cache architecture supporting a strong ECC code. We use a 5-bit correcting code over a 1KB cache line to minimize the area overhead. We check for zero or one-bit errors in the common case to minimize latency and complexity. We use an additional structure, the RALT, to track recently accessed lines so that we avoid reading the whole 1KB line on every cache read, thus minimizing dynamic power. This design allows us to reduce the cache refresh power with minimal area and performance overhead.

4. MULTI-BIT ECC FOR EDRAM CACHES

Error-correcting codes (ECC) have been used extensively in memory and storage devices to tolerate both soft and hard errors. On-chip caches and memory chips typically use simple and fast ECC such as SECDED (single error correction, double error detection) Hamming codes [19], whereas slower devices such as

flash memories use multi-bit ECCs with strong error correcting capabilities (e.g., Reed-Solomon codes [19]). The higher decoding latencies of the strong ECC mechanisms do not pose a problem for mass storage devices, because the encoding/decoding latency is insignificant as compared to intrinsic device access time. However, as a result of technology scaling, the on-chip memory arrays are becoming more susceptible to multi-bit errors, and strong ECC codes are becoming desirable for these fast memories as well.

Besides the latency overhead, the storage overhead of more ECC check bits is another obstacle to using strong codes for on-chip caches. In [34], for example, the authors propose to reduce the cost of storing error-correcting codes for the last level cache (LLC) by partitioning the code between the LLC and memory. Many previously proposed techniques have exploited the tradeoff between code complexity and check bit overhead to mitigate the higher latency of multi-bit ECC. These techniques combine simple ECC with bit rearrangement mechanisms such as bit interleaving [28], address skewing [12] or 2-dimensional product codes [5, 17], to correct certain multi-bit error patterns by dispersing multi-bit errors into multiple single-bit errors. While these techniques enable fast correction, they require more check bits and provide insufficient protection from random multi-bit errors [28].

The Hi-ECC architecture, highlighted in the previous section, is a practical, low-latency, low-cost, multi-bit error-correcting system that can compensate for high failure rates in eDRAM caches. Hi-ECC implements a strong BCH code with the ability to correct 5 and detect 6 errors (5EC6ED). In the remainder of this section, we will introduce the multi-bit BCH algorithm, analyze its circuit complexity and latency, and explain in more detail how our selective correction technique mitigates both the high latency and high cost of strong ECC codes.

4.1 Multi-bit BCH Code

BCH codes are a large class of multi-bit error-correcting codes which can correct both highly concentrated and widely scattered errors [28]. In general, each BCH code is a linear block code defined over a finite Galois Field $GF(2^m)$ with a generator polynomial, where 2^m represents the maximum number of code word bits.

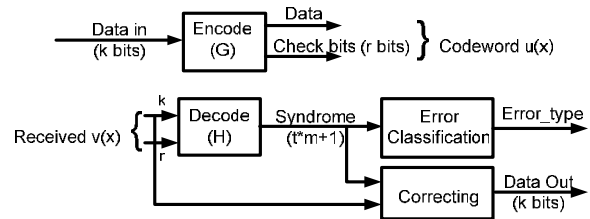**Figure 8. Overview of a BCH-based error-correcting scheme**

Figure 8 shows a high-level block diagram for BCH error correction. The ECC logic includes two main components, (i) ECC encoding and (ii) ECC decoding.

(i) ECC encoding: The encoding logic takes the k-bit input data word d and uses a pre-defined encoder matrix G to generate the corresponding code word u ($u = d \times G$). Since BCH is a

systematic code, the original *k*-bit data is retained in the code word, and is followed by *r* check bits.

(ii) ECC decoding: The decoding logic detects and corrects any errors in the stored code word to recover the original value of data. The decoding logic can be further divided into three components, (a) syndrome generation, (b) error classification and (c) error correction.

(a) Syndrome generation: Let *v* be a code word with error *e*, such that $v = u + e$. The decoder first computes a syndrome *S* by multiplying *v* with the transpose of a pre-defined *H*-matrix ($S = v \times H^T$). The detailed derivation of H-matrix is beyond the scope of this paper and can be found in [13, 28]. However, it is relevant to mention that the G and H matrices are constructed in such a way that $G \times H^T = 0$. The general form of H-matrix is as follows:

$$H = \begin{bmatrix} Parity \\ H_1 \\ H_3 \\ ... \\ H_{2t-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & ... & 1 \\ 1 & \alpha & \alpha^2 & \alpha^3 & ... & \alpha^{(n-1)} \\ 1 & \alpha^3 & \alpha^6 & \alpha^9 & ... & \alpha^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & ... & \vdots \\ 1 & \alpha^{(2t-1)} & \alpha^{2*(2t-1)} & \alpha^{3*(2t-1)} & ... & \alpha^{(2t-1)*(n-1)} \end{bmatrix} \quad (1)$$

In the finite field GF(2^m), each element α^i can be represented as a polynomial of α with a degree less than *m*, or simply a vector with *m* binary coefficients of the polynomial. Therefore, the H matrix can be expanded into a binary matrix with ($t*m+1$) rows, where *t* is the maximum number of errors that the code can correct. Since $S = v \times H^T$, *S* also has $t*m+1$ bits, which can be divided into multiple components [*Parity, S_1, S_3, ..., S_{2t-1}*].

(b) Error classification: The error classification logic uses the syndrome *S* to detect if the code word has any errors. Since:

$$S = v \times H^T = (u + e) \times H^T = (d \times G + e) \times H^T = e \times H^T \quad (2)$$

Therefore, in case of zero errors, $S = 0$ and the following equation would hold true:

$$Parity = S_1 = S_3 = ... = S_{2t-1} = 0 \quad (3)$$

(c) Error correction: If the above equation is not satisfied then the error correction logic uses the syndrome to pinpoint the locations of corrupted bits. Let the error locations in *e* be denoted as [$j_1, j_2, ... , j_t$], then each syndrome component S_i can be specified as:

$$S_i = \alpha^{j1*i} + \alpha^{j2*i} + ... + \alpha^{jt*i} \quad (4)$$

The correction logic implements the following three steps:

Step 1: Determine the coefficients of error location polynomial $\sigma(x)$, where $\sigma(x)$ is defined such that the roots of $\sigma(x)$ are given by the inverse of error elements α^{j1}, α^{j2}, ... , α^{jt} respectively,

$$\sigma(x) = 1 + \sigma_1 x + ... + \sigma_t x^t = (1 - \alpha^{j1} x)(1 - \alpha^{j2} x)...(1 - \alpha^{jt} x) \quad (5)$$

Step 2: Solve the roots of $\sigma(x)$, which are the error locations. When polynomial $\sigma(x)$ is determined, each field element α^j is substituted into the polynomial. Those elements which make the polynomial equal to zero are the roots.

Step 3: Calculate the correct value for data bits. This is done by simply flipping the bits at error locations.

Previous studies have shown that the decoding procedure of multi-bit BCH is tedious [3, 20] and its complexity grows rapidly with the increase in the number of bit corrections. Error correction is the most complex and time consuming component [31]. Step 1 of error correction is based on a *t*-step iterative algorithm, where each iteration involves a Galois Field inversion, which alone takes *2m* operations [4]. The implementation of Step 2 can either take *n*-cycles with one circuit, or a single cycle with *n* parallel circuits [7]. Either way, the base circuit is O($t*m^2$). Overall, both the decoding latency and area complexity is proportional to *(t*m)* [31]. Next, we describe a mechanism for lowering the latency and complexity in the common case of zero or one-bit errors.

4.2. Reducing Decoder Overhead
The latency analysis in the previous subsection is based on the worst case scenario of correctable errors, i.e. *t*-bit errors. However, we observe that for the case of zero or one errors, the detection and correction logic can be a lot simpler and faster:

(i) Detection of zero error: No errors in the code word lead to a syndrome value of zero, as shown in equation (3). This case can be detected by performing a logical OR of all the syndrome bits. This requires *ceil(log₂tm)* 2-input gate delays.

(ii) Single-bit error detection and correction: In the case of a single-bit error, the syndrome exactly matches the H-matrix column that corresponds to the error bit. Therefore, one can detect and pinpoint a single-bit error by comparing each column of the H-matrix with the syndrome. Notice that this correction is significantly faster than the general case of *t*-bit correction (with $t > 1$) because it does not require Step 1 and most of the Step 2 of the error correction logic. In fact, we do not even need to match all the syndrome components with entire H-matrix columns. We just need to compare S_1 to each column in H_1 (defined in equation (1)) and verify that the following equation is satisfied:

$$[(parity=1) \& (S_1^3 == S_3) \& (S_1^5 == S_5) \& ... \& (S_1^{2t-1} == S_{2t-1})] == 1 \quad (6)$$

To minimize latency, we assume that the comparison of S_1 with H_1 and all the comparisons in equation (6) can proceed in parallel.

Based on the above observations, we propose an ECC decoding mechanism that can correct multi-bit errors with a low latency overhead in the common case. The main idea is to differentiate the slow decoding process of multi-bit errors from the case of zero or one errors. In our proposed Hi-ECC mechanism, fast hardware correction is used for the common case of zero or one errors, whereas software or state machine-based correction is used to deal with multi-bit errors, as highlighted in Section 3.2.

4.3 Reducing ECC Storage Overhead
One of the main drawbacks of multi-bit ECC is that correcting more errors requires higher redundancy, which leads to high check bit overhead. To correct *t*-bit errors in *k*-bit input data, a BCH code typically requires $r = t * ceil(log_2 k) + 1$ check bits [28]. In order to reduce the check bit overhead, we make the observation that due to the logarithmic relationship between *r* and *k*, the number of check bits increases much slower than the size of data. Therefore, we can reduce the ECC check bit overhead by increasing the size of data word, as explained in Section 3.1.

Figure 9 shows the check bit overhead for three types of BCH codes correcting different data sizes. For each data size, we show

the total number of check bits (solid lines) and code percentage overhead (dashed lines). The figure shows that the check bit overhead drops significantly with the increase in data size. For example, for 5EC6ED code based on 16B data word, almost 25% of the code word is ECC. However, for a 4KB word, the number of check bits doubles, but the overhead reduces to 0.3%.
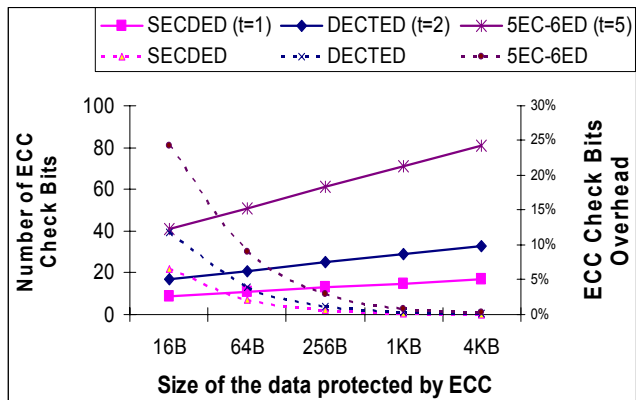

Figure 9. Comparing the storage overhead for check bits for different error-correcting codes. Solid lines represent #check bits (left axis), and dashed lines represent the percentage overhead (right axis).

Using large cache lines introduces another potential problem: writes to sub-blocks within the cache-line may require the entire line to be read every time to regenerate the ECC bits. Fortunately, as a linear code, BCH inherits the additive property of linear systems, which ensures that ECC check bits can be updated using only the information of the modified chunk of data. Let the data word d (representing a cache line) be divided into multiple chunks $[d_{i-1}, d_{i-2},...,d_0]$. The G matrix used in ECC encoding can be divided into two parts as $G=[I_k, P]$, where P is the generator for ECC check word C, i.e., $C = d \times P$. If the jth chunk of data d_j is written with a new value d_{j_new}, then the new ECC would be:

$$C_{new} = d_{new} \times P = (d + [0,...,(d_{j\_old} + d_{j\_new}),...,0]) \times P \qquad (7)$$
$$= C + [0,...,(d_{j\_old} + d_{j\_new}),...,0] \times P$$

The above equation shows that the generation of new check bits requires only the old value of check bits, and the old and new values of the sub-block being modified.

4.4 Logic Overhead

In Section 3, we show that the area overhead for storing the multi-bit error-correcting code is 71 bits for every 1KB line. We estimate the total gate count for the error detection and correction based on Strukov's analysis [31]. Each 64B L3 read requires about 290 XOR gates for each of the 71 check bits for a total of 21k XOR gates. The ECC encoding process requires another 21k XOR gates. The one-cycle single error correction logic consists of three components: 1) error detection (70 ORs), 2) error classification (1k XORs and 1k ANDs) and 3) 1-bit correction (7k XORs and 7k ORs). The logic to handle multi-bit errors integrates the BM algorithm [20] with Chien's search algorithm [7] and uses ~ 47 registers, 28 GF additions, 28 GF multiplications and tens of logic gates, equivalent to fewer than 14k XOR gates but needs many cycles to perform the correction. Overall, the logic uses fewer than 100k XOR gates. Assuming each gate is equal in area

to six eDRAM cells, the total logic overhead is similar to that of 600k bits, or less than 0.1% of the total area for a 128MB cache.

5. EVALUATION

In this section we evaluate the impact of our Hi-ECC proposal on cache power and system performance. We demonstrate that Hi-ECC has almost no impact on performance, while significantly reducing refresh and total power. We first describe our simulation framework, benchmarks, and methodology. We then compare the performance and power of Hi-ECC and other alternatives.

5.1 Simulation Framework

Baseline simulation configuration. We use a cycle-accurate, execution-driven simulator running IA32 binaries. The simulator is micro-operation (uOp) based, executes both user and kernel instructions, and models a detailed memory subsystem. As a baseline, we model an out-of-order superscalar processor that is similar to the Intel® Core™ i7 processor with the addition of a large 128MB eDRAM last-level (L3) cache instead of a traditional SRAM cache. Our memory system includes a 32KB, 8-way set-associative L1 instruction cache, a 32KB, 8-way set-associative L1 data cache, a 256KB, 16-way set-associative unified L2 cache, and a 128MB eDRAM 16-way set-associative L3 (last-level) cache. L1 and L2 caches use 64-byte lines, and the baseline L3 cache uses 1KB lines with no error correction capability. Our processor runs at 2 GHz, and we assume a 40-cycle latency for an L3 hit.

Benchmarks. In our experiments, we simulate nine categories of benchmarks. For each individual benchmark, we select multiple sample traces that well represent the benchmark behavior. Table 1 lists the number of traces and example benchmarks included in each category. We use instructions per cycle (IPC) as the performance metric. We calculate IPC of each category as the geometric mean of IPCs for all traces within that category. We normalize the IPC of each category to the baseline to show relative performance. We use activity factors for different cache structures as inputs to our power model to estimate power consumption. When modeling power consumption, we model a 16-core system running multi-program benchmarks.

Simulated configurations. We model four different configurations, starting with the baseline configuration (BASE) with 1KB L3 cache lines and a 40-cycle L3 hit latency. The second configuration (SD) models an eDRAM augmented with a single error correcting, double error detecting (SECDED) ECC code, and has an additional 2-cycle latency for each L3 hit to model ECC checking, i.e., a 42-cycle total latency. The last two configurations are variations on our Hi-ECC mechanism with and without a RALT. Recall that Hi-ECC uses the multi-bit ECC code to identify and remove lines with high failure rates. Consequently, the latency of multi-bit error processing only impacts processor initialization latency and doesn't impact runtime performance. Furthermore, the 1% reduction in cache capacity (line disable) has a negligible performance impact. As a result, in our performance analysis we focus on the impact of increased L3 latency introduced by Hi-ECC. The Hi-ECC configuration without the RALT (HE) suffers a small penalty for each write (since it must be preceded with a read) and large 32-cycle additional latency penalty for each read (since each 2-cycle 64B read is accompanied by reads to the other 15 sub-blocks in

the line to allow for ECC processing). The Hi-ECC including the RALT (HER) configuration attempts to avoid the additional read latency penalty. With the RALT, the 32-cycle additional penalty typically only applies to the first read in a 1KB line. Subsequent references to other sub-blocks in the line that hit in the RALT will check the sub-block against the parity stored in the RALT. We assume the RALT will be accessed in parallel to the eDRAM to minimize additional latency, but we model the parity check as additional 2-cycle penalty, similar to the SECDED ECC check.

Table 1. Evaluation benchmarks

Category	# traces	Example benchmarks
Digital home (DH)	46	H264 decode/encode, flash
SPECINT2006 (ISPEC)	12	www.spec.org
SPECFP2006 (FSPEC)	22	www.spec.org
Games (GM)	10	Doom, quake
Multimedia (MM)	49	Photoshop, raytracer
Office (OFF)	38	Spreadsheet/word processing
Productivity (PROD)	34	File compression, Winstone
Server (SERV)	15	SQL, TPC-C
Workstation (WS)	24	CAD, bioinformatics
ALL	250	

5.2 Results

Performance. We compare the performance of four configurations with different L3 hit latencies across all nine benchmark categories. On average, the SECDED configuration loses less than 0.1% of its performance compared to our baseline with no error correction. With no RALT, Hi-ECC configuration loses 0.48% of the baseline performance. When we add a RALT with 64 entries, performance loss goes down to 0.35%. With a 512-entry RALT, performance loss is below 0.1%, the same as that of the SECDED configuration. We do not show detailed performance results since they are almost indistinguishable from the baseline. These results show that Hi-ECC can achieve performance comparable to the baseline configuration.

Power modeling. We compared the refresh power, dynamic power, and total power for our four configurations. We used an analytical model for power that uses activity factors from our simulation experiments (e.g., number of L3 reads/writes and number of RALT hits/misses per second) to obtain different power components. We assume a 16-core system for power modeling, since the dynamic

power consumed by a single core is almost insignificant compared to refresh power. We provide more details on our power model in Appendix A. We estimate that the baseline refresh power is approximately 926mW. We also estimate that the dynamic power is 421mW per L3 sub-block read, 425mW per L3 sub-block write, 22mW per RALT access, and 5152mW for reading an entire 1K line with 16 sub-blocks.

Refresh power. We scale the refresh time from 30us for the baseline to 150us and 440us for SECDED and Hi-ECC, respectively. This scaling leads to a similar scaling for refresh power, from 926mW for the baseline to 185 and 63mW for SECDED and Hi-ECC, respectively. SECDED achieves a 5X reduction in refresh power compared to our baseline, and Hi-ECC achieves almost a 3X reduction compared to SECDED (almost 15X compared to the baseline).

Dynamic power. The additional activities due to extra reads, writes and ECC checks cause the dynamic power of the SECDED and Hi-ECC configurations to increase. Figure 10 shows the refresh power, dynamic (read and write) power, and total power across all benchmark categories for our four configurations. The figure demonstrates that SECDED has a marginal increase in dynamic power over our baseline due to ECC checks. However, the Hi-ECC configuration with no RALT (HE) incurs a significant increase in dynamic power due to reading the whole 1K cache line with 16 sub-blocks on every L3 read. If only refresh power is considered, Hi-ECC will consume 34% of the power consumed by SECDED. However, the increase in Hi-ECC's dynamic power pushes its total power to 58% of SECDED's total power. On the other hand, using a 512-entry RALT (HER) decreases the number of 1KB line accesses significantly, and the total power for Hi-ECC with RALT becomes 39% of the total power of SECDED (including the extra power due to RALT accesses).

6. CONCLUSIONS

In this paper, we have argued that a significant portion of idle power in future systems will be used to refresh eDRAM. To address this, we have developed Hi-ECC, a practical system for tolerating refresh-related failures in eDRAM-based caches. Hi-ECC reduces the cache refresh power by 93%, compared to an eDRAM with no error correction, and by 66% compared to an eDRAM with a single error-correcting code (SECDED). We accomplish this with only 2% storage overhead and without the loss of any cache state.
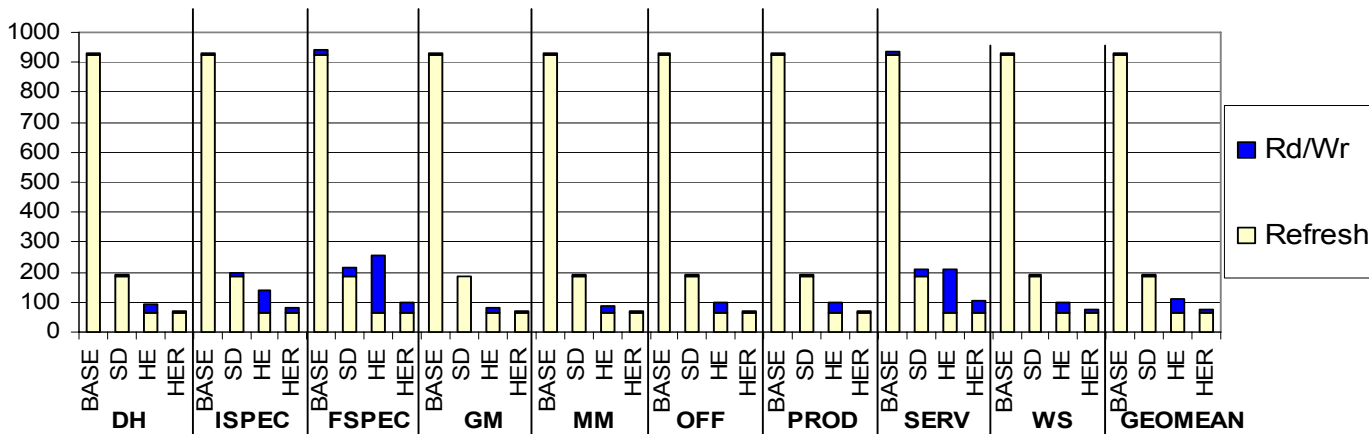

Figure 10. Refresh power, dynamic (Rd/Wr) power and total power in mW for different benchmark categories across all four configurations on 16 cores.

By employing a strong error-correcting code and augmenting it with other failure mitigation mechanisms, such as bit-fix, Hi-ECC offers a highly scalable solution for addressing very high failure rates. Furthermore, although this paper's evaluation focuses on eDRAM, elements of Hi-ECC are applicable to many different memory technologies and failure types, including Vccmin related SRAM failures. In addition, Hi-ECC's reliance on strong error-correcting codes instead of memory tests to identify bit failures makes it applicable to high capacity memories, such as bulk DRAM and phase change memories.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] A. Agarwal, et al., "Process variation in embedded memories: failure analysis and variation aware architecture," *IEEE Journal of Solid-state Circuits*, vol. 40, no. 9, pp. 1804-1814, Sep., 2005.

[2] J. Barth, et al., "A 500 MHz random cycle, 1.5 ns latency, SOI embedded DRAM macro featuring a three-transistor micro sense amplifier", *IEEE Journal of Solid State Circuits*, vol. 43, no. 1, pp. 86–95, Jan. 2008.

[3] E. R. Berlekamp, Algebraic coding theory, New York: McGraw-Hill, chapter 7, 1968.

[4] H. Brunner, A. Curiger and M. Hofstetter, "On computing multiplicative inverses in $GF(2^m)$," *IEEE Transactions on Computers*, vol. 42, pp. 1010-1015, Aug. 1993.

[5] H. O. Burton and E. J.Weldon, Jr., "Cyclic product codes," *IEEE Transactions on Information Theory*, vol. 11, no. 3, pp. 433–439, Jul. 1965.

[6] J. Chang, et al., "The 65-nm 16-MB shared on-die L3 cache for the dual-Core Intel® Xeon processor 7100 series," *IEEE Journal of Solid-state Circuits*, vol. 42, no. 4, pp. 846-852, Apr. 2007.

[7] R. T. Chien, "Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes," *IEEE Transactions on Information Theory*, vol. 10, no. 4, pp. 357-363, Oct. 1964.

[8] P. Emma, W. Reohr and M. Meterelliyoz, "Rethinking refresh: Increasing availability and reducing power in DRAM for cache applications," *IEEE Micro*, vol. 28, no. 6, pp. 47-56, Nov 2008.

[9] V. George, "45nm next generation Intel Core microarchitecture (Penryn)," Hot Chips 19, Stanford, CA, Aug. 2007.

[10] M. Ghosh and H. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs," in *Proceedings of the 40th International Symposium on Microarchitecture*, pp. 134–145, Dec. 2007.

[11] T. Hamamoto, S. Sugiura and S, Sawada, "On the retention time distribution of dynamic random access memory (DRAM)," *IEEE Transactions on Electron Devices*, vol. 45, no. 6, pp. 1300-1309, Jun. 1998.

[12] Mu Y. Hsiao, Douglas C. Bossen, "Orthogonal latin square configuration for LSI memory yield and reliability enhancement," *IEEE Transactions on Computers*, vol. 24, no. 5, pp. 512-516, May 1975.

[13] H. Imai and Y. Kamiyanagi, "A construction method for double error correcting codes for application to main memories," *Transactions of the IECE Japan*, vol. J60-D, pp. 861-868, Oct. 1977.

[14] R. Kalla, "Power7: IBM's next generation POWER microprocessor," Presentation at Hot Chips 21, Stanford, CA, Aug. 2009.

[15] J. Kim and M. Papaefthymiou, "Dynamic memory design for low data-retention power," in *Proceedings of the 10th International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation*, pp. 207-216, Sep. 2000.

[16] J. Kim, M. Papaefthymiou, "Block-based multiperiod dynamic memory design for low data-retention power," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp.1006-1018, Dec. 2003.

[17] J. Kim, et al., "Multi-bit error tolerant caches using two-dimensional error coding," in *Proceedings of the 40th Annual International Symposium on Microarchitecture (MICRO)*, pp. 197-209, Dec. 2007.

[18] W. Kong, et al., "Analysis of retention time distribution of embedded DRAM - A new method to characterize across-chip threshold voltage variation," in *Proceedings of IEEE International Test Conference (ITC 2008)*, pp. 1-7, Oct. 2008.

[19] S. Lin and D. J. Costello. Error control coding, Second Edition. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[20] J. L. Massey, "Step-by-step decoding of the Bose-Chaudhuri-Hocquenghem codes," *IEEE Transactions on Information Theory*, vol. 11, no. 4, pp. 580–585, Apr. 1965.

[21] R. Matick and S. Schuster, "Logic based eDRAM: origins and rationale for use," *IBM Journal of Research and Development*, vol. 49, no. 1, pp. 145 – 165, Jan. 2005.

[22] Micron Technology, Inc. "TN-41-01 : Calculating memory system power for DDR3", http://download.micron.com/pdf/technotes/ddr3/TN41_01DDR3%20Power.pdf

[23] K.Mistry, et al., "A 45nm logic technology with high-k+metal gate transistors, strained silicon, 9 Cu interconnect layers, 193nm dry patterning, and 100% Pb-free packaging," in *Proceedings of IEDM* 2007, pp. 247–250, Dec. 2007.

[24] A. Naveh, et al., "Power and thermal management in the Intel® Core® Duo processor," *Intel Technology Journal*, vol. 10, no. 2, May 2006.

[25] S. Natarajan, et.al., "A 32nm logic technology featuring 2nd-generation high-k + metal-gate transistors, enhanced channel strain and $0.171um^2$ SRAM cell size in a 291Mb array," in *Proceedings of IEDM* 2008, pp. 1-3, Dec. 2008.

[26] T. Ohsawa, K. Kai and K. Murakami, "Optimizing the DRAM refresh count for merged DRAM/Logic LSIs," in *Proceedings of the 1998 International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 82-87, August 1998.

[27] S. Ozdemir, et. al., "Yield-aware cache architectures," in *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*, pp. 15-25, Dec, 2006.

[28] T. Rao, E. Fujiwara, "Error-control coding for computer systems," Prentice-Hall, Inc., Upper Saddle River, NJ, 1989.

[29] D. Roberts, N. Kim and T. Mudge, "On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology," in *Proceedings of 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pp. 570-578, Aug. 2007.

[30] D. Somasekhar, et al., "Multi-phase 1GHz voltage doubler charge-pump in 32nm logic process," in *Proceedings of 2009 Symposium on VLSI Circuits,* pp. 196-197, Jun. 2009.

[31] D. Strukov, "The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories," in *Proceedings of 2006 Asilomar Conference on Signals Systems and Computers*, pp. 1183-1187, Oct. 2006.

[32] R. Venkatesan, S. Herr and E. Rotenberg, "Retention aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM," in *Proceedings of 12th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 155-165, Feb. 2006.

[33] C.Wilkerson, et. al, "Trading off cache capacity for reliability to enable low voltage operation," in *Proceedings of 35th International Symposium on Computer Architecture (ISCA-35)*, pp. 203-214, Jun. 2008.

[34] D. H. Yoon and M. Erez, "Memory Mapped ECC: Low-Cost Error Protection for Last Level Caches", in *Proceedings of the 36th International Symposium on Computer Architecture (ISCA-36)*, pp. 116-127, June, 2009.

APPENDIX A: EDRAM POWER MODEL

Figure 11 shows the topology of the bit-cell and the array organization for the 128MB cache used in this study. This cache is organized with 128x32 sub-arrays. Each sub-array is made out of 67 I/O columns. A pair of "data-in data-out" (16 cells) is referred to as an I/O column. Each I/O column has 16 columns and 256 rows. This array organization allows data to move out of the cache in multiple cycles. To deliver the 64B data word, 8 sub-arrays are activated at a time.

In order to account for all capacitance for power calculation, we need to estimate the silicon area. Area is calculated with a conservative cell size of $16F^2$ where F is ½ of the minimum pitch of the process technology cell. Note that [2] quotes cell sizes smaller than this study. Table 2 illustrates the scaling of the technology parameters to a projected 22nm node. VccE is the voltage of the stored cell, while Vcc is the supply voltage for logic circuits. Array area calculation is performed based on a sense-amplifier structure efficiency loss of 35%, a wordline decoder loss of 25% and an overall assembly loss of 90% in each of the X and Y dimensions.

In calculating the power consumption, we use the I/O column structure and the key technology metrics depicted in Figure 12. Note that the voltage swing on the word line is higher (1.8 V [2]) and is assumed to be generated by a 60% efficient charge pump [30]. Other design parameters include the usage of a small-signal sense amplifier with half-Vcc precharge, I/O column structure and a design with strobed data in the sense-amplifier which is pulled

out using column selection (YSEL devices) to a final CMOS driver. Dynamic power of the nets is estimated by first computing an effective switched capacitance *Cdyn* and then putting it into the equation: $Pdyn = ActivityFactor \times Cdyn \times Vcc2 \times F$. *Cdyn* accounts for global nets, clock circuit and repeater buffers. With this organization refreshing 8 sub-arrays requires only 256 refresh cycles each *Trc* clocks long and occurs at the retention rate. Thus, the amortized refresh power for 128MB over a retention period of 30 microseconds is 926 mW.
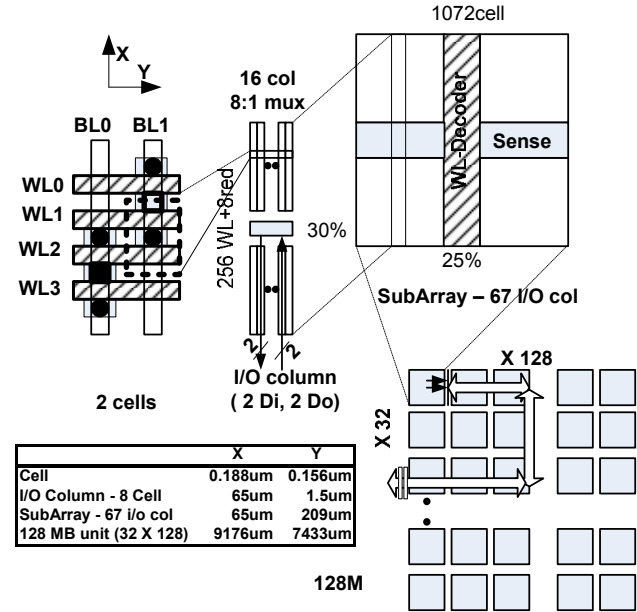

	X	Y
Cell	0.188um	0.156um
I/O Column - 8 Cell	65um	1.5um
SubArray - 67 I/o col	65um	209um
128 MB unit (32 X 128)	9176um	7433um

Figure 11. 32MB eDRAM organization

Node	Met Pitch	BL Pitch	Gate Pitch	Min Device	Sense Dev	VccE	Vcc
45 nm	160nm	320nm	192nm	135nm	3600nm	1V	1V
32 nm	112nm	224nm	134nm	96nm	2560nm	1V	0.9V
22 nm projected	78nm	156nm	94nm	66nm	1760nm	1V	0.8V

Table 2. Process technology parameters [23, 25]

Frequency	2 GHz
Trc	4 clk
Cwire	0.2 fF/um
Ccontact	0.6 fF/um
Cgate	1.5 fF/um
Idrive	1 mA/um
Drive Ratio	4 X
P/N ratio	1 X

Per Column	Capacitance	Voltage swing
Bit-Line	23.24 fF	0.5V
Word-Line	0.20 fF	1.8V
Sense-Amp enable	7.47 fF	1.0V
PCH	1.25 fF	1.0V
Ysel+Final	0.86 fF	1.0V (1 of 8)

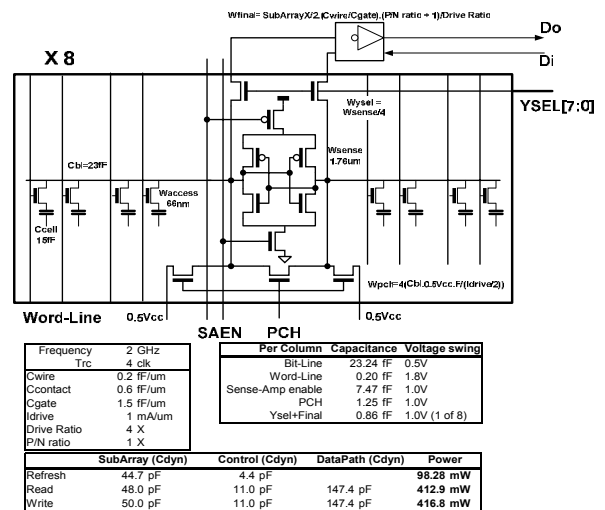|	SubArray (Cdyn)	Control (Cdyn)	DataPath (Cdyn)	Power
Refresh	44.7 pF	4.4 pF		98.28 mW
Read	48.0 pF	11.0 pF	147.4 pF	412.9 mW
Write	50.0 pF	11.0 pF	147.4 pF	416.8 mW

Figure 12. I/O Column structure and technology metrics for power modeling