Efficient RAS support for 3D Die-Stacked DRAM

Hyeran Jeon
University of Southern California
hyeranje@usc.edu

Gabriel H. Loh
AMD Research
gabriel.loh@amd.com

Murali Annavaram
University of Southern California
annavara@usc.edu

Abstract

Die-stacked DRAM is one of the most promising memory architectures to satisfy high bandwidth and low latency needs of many computing systems. But, with technology scaling, all memory devices are expected to experience significant increase in single and multi-bit errors. 3D die-stacked DRAM will have the added burden of protecting against single through-silicon-via (TSV) failures, which translate into multiple bit errors in a single cache line, as well multiple TSV failures that may lead to an entire channel failure. To exploit wide I/O capability of 3D DRAM, large chunks of data are laid out contiguously in a single channel and an entire cache line is sourced from a single channel. Conventional approaches such as ECC DIMM and chipkill-correct are inefficient since they spread data across multiple DRAM layers to protect against failures and also place restrictions on the number of memory layers that must be protected together. This paper adapts several well known error detection and correction techniques while taking into account 3D DRAM's unique organization. First, we decouple error correction from detection and perform error detection using a novel two level 8-bit interleaved parity to handle die-stacked DRAM-specific failure modes such as TSV failures. We then adapt RAID5-like parity, a technique developed for hard disks which also layout large chunks of data contiguously, for recovering from a wide range of errors from single-bit errors to channel-level failures without the need to splice data across multiple layers. As further optimizations, a two-step decoupled error correction code update process is used to improve write speed, and an error detection code cache is used for improving read/write performance without compromising reliability. The proposed approaches effectively reduce the FIT rate with 15.7% area and almost negligible performance overhead.

1. INTRODUCTION

With the shrinking feature size, reliability is a growing concern for memory. The 2D DRAM failure modes are dominated by single bit failures, but failures at the granularity of word, column, row, bank, to an entire chip failure have been observed. According to the recent field study [23], half of all the failures are single-bit errors but multi-bit errors such as row, column and bank level failures also happen with over 10% possibility each. More interestingly, the study found that majority of the DRAM failures are caused by permanent faults. In a conventional DRAM, ECC has been widely used to defend against errors. Typically, a DRAM module consists of multiple 4- or 8-bit-wide DRAM chips and bits from all the chips on a single DRAM module are combined to form a single data word along with the error correcting code (ECC) associated with the data word. We call the data + ECC as the *ECC word* as illustrated in Figure 1(a). Depending on the strength of error correction, any N-bit error can be corrected. Most commonly, DRAMs employ single-error correcting, double-error detecting (SECDED) codes. Any single-bit errors can be corrected within an ECC word.

IBM proposed chipkill-correct [5] to cover the chip level failures. Dell PowerEdge 6400 and 6450 servers [13] implement the chipkill-correct in which all the bits of each ECC word are scattered into multiple DRAM chips. Given such data scattering, a single chip failure becomes a single bit error within each ECC word and thereby it is correctable by the SECDED code. Another chipkill-correct design is implemented in Sun UltraSPARC-T1/T2 and the AMD Opteron [28]. Instead of scattering every bit of each ECC word, they use longer ECC word to provide more ECC bits so that each ECC word can correct more than a single-bit failure. They use a symbol based ECC that can correct a single symbol failure and detect up to double symbol failures (SSC-DSD). In the symbol based ECC, any bit errors of a single symbol are correctable which means, once the bits within a symbol are all sourced from a DRAM chip, the single chip failure can be tolerated.

Recently, DRAM designers are moving toward 3D die-stacking. Multiple DRAM chips are vertically integrated into a single die-stack. Die-stacked DRAM provides notable advantages such as high bandwidth, low latency, and small footprint. Die-stacked DRAM's capacity has been increasing from hundreds of megabytes initially to a few gigabytes now [1]. Due to initial capacity limitations, the die-stacked DRAM has been actively studied as a last level cache [30, 15, 20]. However, with silicon interposer-based solutions [11, 18] that enable the integration of multiple memory stacks within the same package, die-stacked DRAM can be used as the system's main memory for a variety of market segments. Recently, NVIDIA revealed their GPU roadmap which shows the planned use of die-stacked DRAM as main memory by employing multiple stacked DRAM layers around the GPU [24]. Similarly, in real-time embedded devices and certain cloud servers with known memory capacity needs, die-stacked DRAM's high bandwidth, low latency and small footprint make it an attractive option. These domains also tend to have higher *reliability, availability and serviceability* (RAS) requirements. When die-stacked DRAM is placed on the same package as the CPU, the importance of the serviceability is further escalated because even with a small number of hard faults in a DRAM chip, the entire package of CPU+DRAM would need to be replaced, which is going to be a lot more expensive than replacing an external

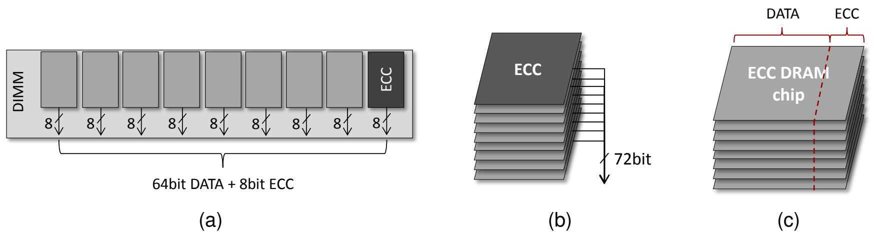


Figure 1: Organization of (a) a conventional ECC memory with 9 x8 DRAM chips, (b) a possible conversion to die-stacked DRAM with 9th layer for ECC, and (c) another possible solution to use ECC DRAM chip

DIMM in a conventional memory organization.

1.1. Why Is Reliability Support For Die-stacked DRAM Challenging?

Die-stacked DRAM has unique memory organization to take advantage of its superior bandwidth. The die-stacked DRAM typically has much wider channel width, 128 bits per chip as opposed to 4/8 bits per chip in planar DRAM. One or two ranks form a layer in 3D DRAM and each layer is allocated its own independent channel(s) to access data from each layer. The upcoming JEDEC High-Bandwidth Memory(HBM) [8] standard provides 1024 bits of data over 8 channels. The minimum burst length is said to be 16B-32B. Hence, 3D DRAMs can potentially bundle 128-256 bits of data from a wide channel every cycle. To take advantage of the wide I/O capability large chunks of data is contiguously laid out in a single chip. Instead of spreading cache line data across multiple chips, as has been done in traditional DRAMs, 3D DRAMs place an entire cache line worth of data on a single DRAM chip. Traditional approaches for planar DRAM protection, such as ECC-DIMM and chipkill-correct, do not lend themselves well to protect 3D DRAMs. The details are discussed in the following paragraphs.

Possible but Inefficient Solutions: Figure 1(b) and (c) show two possible conversions of the conventional ECC memory to die-stacked DRAM. One approach is to add an ECC layer within a DRAM stack as illustrated in Figure 1(b). Alternatively, one could stack ECC DRAM chips that have 12.5% wider data rows as in Figure 1(c). In this approach one has to read all the stacked layers to read a few bits (4 or 8 bits) from each layer, including the ECC layer, to construct a single ECC-word. Hence, when ECC-DIMM technique is directly applied to 3D DRAMs, then contiguous allocation of data on a single chip is not viable. Even a single word may require activating all layers in a 3D stack. While this configuration is a simple extension of traditional DRAMs, there is a significant bandwidth loss because only 8-bit data is transferred through each 128-bit-wide channel in any given cycle, which is further exacerbated by the minimum burst length. Since the minimum bust length is at least 16 bytes, reading just 8-bit data from a single burst is a significant bandwidth loss. In addition to wasting bandwidth, activating all layers within a stack to service even a single cache line can lead to significant power overhead to read a single cache line. It would be much better to exploit the 128-bit channel width of each layer to read an entire cache line from a single layer. Since there is one ECC layer per a fixed number of DRAM layers, there is a scalability concern as well. For example, the 12.5% area overhead for SECDED code requires one ECC layer for each 8 data layers. Hence, if DRAM manufacturers want to add just one additional data layer above 8 data layers to provide higher density for a given market segment, a second ECC layer must be added.

An alternative solution to avoid the multiple stack or layer approach is to increase the width of each data row by the size of SECDED code (12.5%) such that the ECC and the data can be read from a single layer as illustrated in Figure 1(c). However, this design is not desirable as it requires the memory vendors to either design two types of chips (ECC and non-ECC), or endure the cost of 12.5% area overhead by using ECC chips even when cost sensitive market segments do not support ECC. Note that for DIMM-based memory, there is no such design concern because the single-chip design can be used in both ECC or non-ECC DIMMs by placing eight or nine chips (for x8 chips) on a DIMM; the design and manufacturing costs for two different DIMMs is significantly cheaper than for two different silicon designs.

The symbol-based ECC itself has several inefficiencies that make it particularly challenging to apply to 3D DRAMs. Due to the algorithmic constraints, the symbol-based ECCs used in the commodity DRAM modules restricts to use x4 or x8 memory chips even though embedding smaller number of chips of wider interface provides better power efficiency [25, 28]. Even if these constraints are ignored, SECDED can recover only from a single-bit failure and is not capable of dealing with multi-bit failures that are becoming more prominent. Actually, a recent study showed that 2.5% and 5.5% of the DRAM failures are multi-bank and multi-rank failures, respectively [23]. Hence, we need to explore alternative approaches to deal with multi-bit errors as well as channel-level failures in a unified manner. 3D DRAMs have the added burden of dealing with single through-silicon-via (TSV), and multiple TSV failures. A single TSV failure can lead to four bits of error per 64 byte cache line, since a single cache line is source from a single DRAM chip over four consecutive cycles. More generally, 3D DRAM requires at least $\frac{bits\_in\_cache\_line}{interface\_width}$ bits of error detection and correction capability. Multiple TSV failures lead to an entire channel failure. Hence, protecting against multi-bit errors requires providing stronger protection codes, such as DECTED (double error correcting and triple error detection). But, the area overhead of ECC grows super-linearly as the strength of the protection increases [9]. For instance, a DECTED code has more than 20% overhead.

Summary: A new error detection/correction scheme is needed for die-stacked DRAM with the following properties: 1) the scheme can protect against multi-bit errors as well as protect against channel-level failure, 2) the scheme should enable us to take advantage of wide I/O and hence should not require data to be finely spliced across multiple layers to provide protection 3) the scheme can scale irrespective of the number of layers in the DRAM stack 4) the scheme should be power efficient and hence should access as few DRAM layers as possible for the common case of error detection which is done on every memory access. Obviously, these features are going to come at some expense. In our proposed scheme, the latency for error correction will be significantly higher than ECC, although error detection latency is not compromised.

1.2. Contributions

This paper adapts several well known error detection and correction techniques while taking into account 3D DRAM's unique memory organization. First, to reduce the performance overhead for the regular error-free accesses, we decouple error detection from error correction. The decoupled error detection and correction eliminates the computation overhead of complex error correction code out of the critical path of the execution. Then, we propose to extend the traditional 8-bit interleaved parity to handle die-stacked DRAM-specific failure modes such as TSV failures. While standard 8-bit interleaved parity can detect 8 consecutive bit errors, it does not protect against a single TSV failure. Hence, we adapt the traditional interleaved parity to create a two-level 8-bit interleaved parity to protect against a TSV failure.

We adapt RAID5-like parity [4], a techniques developed for hard disks which also layout large chunks of data contiguously, for recovering from a wide range of errors from single-bit errors to channel-level failures without the need to splice data across multiple layers.

We then co-locate error detection and correction codes with the data on the same die-stack to improve scalability. Thus error detection and correction capabilities can be incrementally added to each layer in a die-stack without any restrictions on the minimum number of stacked layers.

As further optimizations, a two-step decoupled error correction code update process is used to improve write speed, and an error detection code cache is used for improving read/write performance without compromising reliability.

The proposed approaches effectively reduce the FIT rate with 15.7% area and almost negligible performance overhead even assuming an aggressive bit-error rate for a sixteen-channel 3D DRAM.

The remainder of this paper is organized as follows. Section 2 describes the proposed architecture. Section 3 shows the optimization techniques for the proposed architecture. Section 4 discusses the error coverage as well as area overhead. Section 5 shows our evaluation methodology and results. Section 6 discuss related work and we conclude in Section 7.

2. PROPOSED ORGANIZATION

2.1. Baseline Die-stacked DRAM Structure

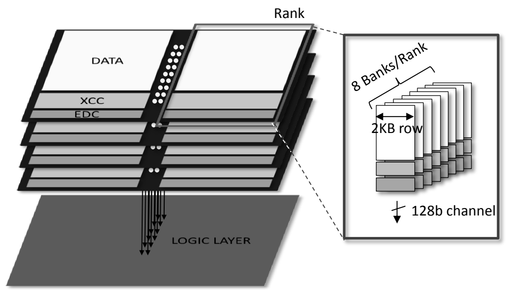


Figure 2: Illustration of proposed memory configuration

The baseline DRAM stack has eight ranks organized as four layers, with two ranks per layer, in a die-stack. Each rank has eight banks and is provided with its own 128-bit channel. The minimum burst length is assumed to be 32B. These parameters reflect the JEDEC High-Bandwidth Memory(HBM) [8] standard that provides 1024 bits of data over eight channels. For simplicity of discussion, we assume that the cache line size is 64B in this paper, although the size of the cache line can be changed with minimal changes to the description. Thus, a 64B cache line can transferred through a 128-bit channel in four memory transfers (i.e., two memory cycles assuming a double-data rate (DDR) interface). The width of each row in a bank is 2KB. Row interleaving is used for the baseline address mapping (e.g., 2KB of consecutively addressed data is placed in Rank#0, the next 2KB in Rank#1, and so on). Because each rank has its own channel, we sometimes use the terms rank and channel interchangeably.

2.2. Data & ECC Co-location

As explained in the previous section, adding extra layer or increasing row size by ECC code length is not desirable in die-stacked DRAM. For better scalability, we propose to store the redundant data necessary for error detection and correction in the DRAM cells with the data as illustrated in Figure 2.

Redundant data necessary for error correction is stored in each bank at the end of the data row. Each bank consists of three different regions: Data, XOR correction code (XCC), and error detection code (EDC). XCC maintains all the redundant data necessary for correcting errors, while EDC maintains the data necessary for quickly detecting an error both in the data as well as in the XCC regions, and the data region contains the normal values stored in memory. The size of EDC and XCC regions is changeable depending on the code type. For example, in this paper, we evaluate 2-level 8-bit interleaved parity per 64B cache line for EDC and RAID5-like XCC for error correction.

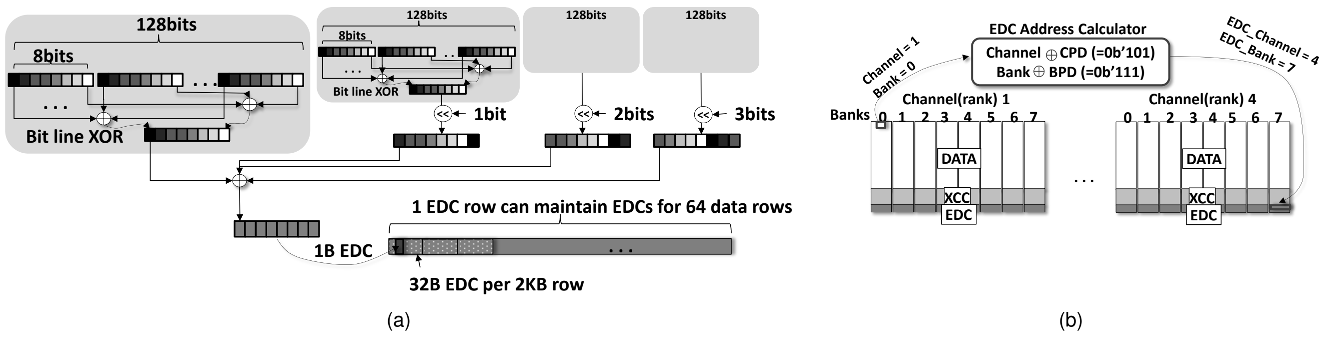


Figure 3: EDC computation and placement: (a) Two level 8-bit interleaved parity calculation to scale up the detection capability to the single TSV failure (b) Permutation based EDC placement to avoid bank and channel conflict

2.3. Error Detection Code (EDC)

Error Detection Code Selection: Even when DRAMs encounter increasing number of errors, error free operation will be the common case. Combining error detection and correction as a single code typically requires complex computation which may fall in the critical path of reading data. Recognizing this concern, many prior approaches have argued for separating error detection from error correction [16, 28, 25]. In this paper, we adopt the approach of decoupling detection from correction.

The wide I/O 3D DRAMs that are being targeted in this paper provide an entire cache line width (64 bytes) of data from a single channel. We exploit this wide read capability while exploring various error detection codes by providing an error detection code for an entire cache line. We evaluated a wide range of error detection codes: 8-bit interleaved parity, cyclic-redundancy codes (CRC), Fletcher and Adler checksums, 1's complement addition and 2's complement addition [10]. Each of these approaches was aimed at generating an error detection code for an entire cache line. For instance, we computed one 8-bit parity for an entire 64 byte cache line, or one 16-bit CRC for 64 byte cache line.

Of all the approaches, we found that the 8-bit interleaved parity is the most lightweight both in terms of its computational demand as well as the power consumption. However, in terms of error detection capability, CRC is the strongest error detection code [10]. For instance, 16-bit CRC can detect 3 random bit errors, irrespective of the bit positions of these errors within a cache line, as well as 16-bit burst errors. On the other hand, traditional 8-bit interleaved parity can detect 8-bit burst errors within a cache line but it cannot detect multiple errors that occur in the same bit position in every 8-bit data unit. For example, when two bits in the same position in every 8-bit unit of a data are flipped, these errors cannot be detected by 8-bit interleaved parity because those bits are XORed together when generating the parity. In 3D DRAM, a single TSV failure will generate multiple errors in a cache line and each error is precisely located at the same location in each 128-bit read. This 3D DRAM-specific error is precisely the type of error that an 8-bit interleaved parity fails to detect. Hence, when we applied 8-bit parity to 3D DRAM, any single TSV failure will go undetected. To take advantage of the 8-bit interleaved parity's performance and energy efficiency while scaling the detection capability up to a single TSV failures which are unique to 3D DRAM, we propose a two level 8-bit interleaved parity calculation as shown in the Figure 3(a).

Instead of XORing entire 64B cache line in 8-bit units together as in the traditional 8-bit interleaved parity, we partition the calculation into two levels. In the first level, we calculate four intermediate 8-bit interleaved parities for each 128-bit data unit, which is the minimum transfer size of wide I/O 3D DRAM. Then, in the second level, the intermediate parities of the first, second, third, and fourth 128-bit data are rotated by zero, one, two, and three bits respectively. The four rotated intermediate XORs are then XORed together to make the final 8-bit interleaved parity. By doing the rotation, the bits that might be flipped due to the TSV failure are XORed into different bit positions in the final 8-bit interleaved parity. Hence, we can detect any single TSV failure with this modified two-level 8-bit interleaved parity. While this error detection code is the preferred option, we evaluated traditional 8-bit interleaved parity, 16-bit CRC, SECDED in our results section. Our goal is to provide quantitative data to 3D DRAM designers to evaluate performance and error capability tradeoffs of some well known prior techniques alongside the two-level 8-bit interleaved parity.

Error Detection Code Location: One option for storing EDC is to place the EDC at the end of each data row (i.e., a 2KB row consists of 31 X 64B cache lines, followed by one 32B EDC). However, this organization creates address decoding complexity because a non-power-of-two (POT) computation is required to find the column and row indexes for a given cache line address. To avoid the row and column decoder logic modification, we store the EDCs in separate region starting at the end of the XCC region. Each EDC row of 2KB can hold the EDCs for 64 data rows when two-level 8-bit interleaved parity is used. Thus, 1.5% of the rows in every bank are assigned as the EDC region.

While separating EDC simplifies address decoding, placing EDC in the same bank with the data results in a bank conflict.

Note that after each cache line is read from the data row, the corresponding EDC must also be read to verify the cache line data. Therefore, we use a permutation-based EDC mapping. Zhang *et al.* proposed a permutation-based page interleaving scheme to reduce bank conflicts [29]. The idea is to spread rows across multiple banks by XORing some partial bits of L2 tag with bank index. Our goal is actually much simpler: always use different banks for storing data and the corresponding EDC. The banks where data and the corresponding EDC reside can be statically determined. As such, a much simplified approach of using a fixed distance permutation is used. While any permutation distance from 1 to $\#banks - 1$ can be used, in this paper a permutation distance of seven is used. To obtain the bank number where EDC will be stored, the permutation distance (seven in our case) is XORed with the bank index of the data. For example, when seven (0b'111) is used for the permutation distance, and the data is stored in bank two (0b'010), the corresponding EDC bank is determined to be five (0b'101) which is the XORed result of two and seven. With the fixed distance permutation of seven, data and the corresponding EDC are guaranteed to be stored in different banks.

To take further advantage of potential channel-level parallelism, we also opted to store the EDC in different channels. If data and EDC are in the same channel, then they both must be accessed sequentially as each row is transferred through the same channel interface. If EDC and data are placed in different channels, then both data and EDC can be read in parallel. We use the channel permutation distance (CPD) to XOR with the channel number of the data to decide on the channel number for storing the EDC. Thus, the bank and channel number of EDC will be determined by XORing the data's bank index with bank permutation distance (BPD), and channel number with channel permutation distance (CPD), respectively. The block labeled *EDC Address Calculator* in Figure 3(b) illustrates the channel and bank number computation of an EDC data corresponding to one cache line. Because we are only adding an XOR operation in the memory access path without changing the existing row and column decoder, the additional delay is negligible compared to overall memory access latency.

Address Translation for Accessing Error Detection Code: The path for computing the EDC bank and channel for a data address is shown in Figure 4(a). The physical address of the data is split into Row, Bank, Channel, Column and Byte addresses. The EDC base row register stores the first row number where the EDC region begins in any bank. The row number of the address is first right-shifted by $log_2(\#data\_rows\_per\_EDC\_row)$. For instance, in Figure 3, one EDC row can store the EDCs for 64 data rows (32B of EDC per each data row) when 8-bit interleaved parity is used for EDC. Then, the data row number is right shifted by $log_2(64) = 6$ and added to the base register to get the EDC row number. The EDC bank and channel numbers are simply the bank and channel indexes XORed with BPD and CPD, respectively. The column address computes the column within an EDC row where the EDC of a given cache line is stored. There are three shift, two XOR, two addition, and one logical-AND operations. But as shown in the figure, most of these operations are carried out in parallel.

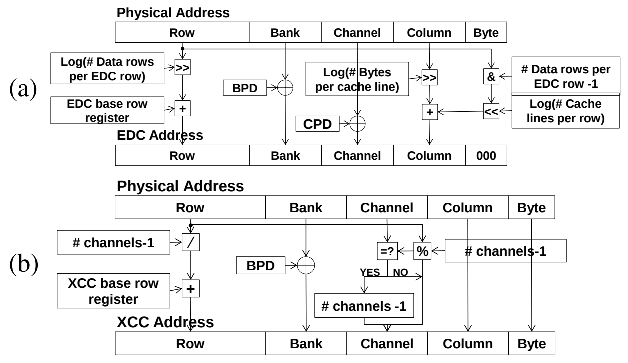


Figure 4: Address translation for (a) EDC and (b) XCC

2.4. XOR Correction Code (XCC)

Error Correction Code Selection: The majority of error-free requests can be handled by EDC as mentioned in the previous section. When an error is detected, we use an XOR-based correcting code (XCC) to correct the error. For this purpose, we rely on RAID5-like parity [4], which was originally proposed to protect data from catastrophic hard disk failures. In 3D DRAM, there is a strong correspondence to disk failures when multiple TSVs fail thereby leading to an entire channel failure. In a DRAM having N channels, $\frac{1}{N-1}$ of the rows in every bank is assigned for storing XCC. To aid the description of how XCC is computed, Figure 5 shows the proposed XCC computation framework when there are eight data channels. Among all the rows in each bank, $\frac{1}{7}$ of the rows are allocated for the XCC region. In each ith row across the eight channels, the rows from seven channels having the same color are XORed together to make a XCC row of the same color. The XORed data is stored in the XCC region in the channel whose data is *not* XORed. For instance, the data in channel 1-row 0 (labeled as 1-0 in the figure), channel 2-row 0 (labeled 2-0), ..., channel 7-row 0 (labeled 7-0) are XORed and the data is stored in XCC region of channel 0. Similarly, channel 1 stores the XORed data from 0-1, 2-1, ..., 7-1 which excludes 1-1. After iterating through seven data rows, there will be data left in one channel per each row that is not XORed anywhere. These un-XORed rows (which are colored black in the figure) from seven diagonal data rows across seven channels will be separately XORed to be stored to the 8^{th} channel in the XCC region.

Error Correction Code Location: Our proposed approach for computing XCC is similar to the RAID5 [4] parity computation. The difference is that the XCC data is stored in the channels alongside data but in a distinct XCC region. In a die-stacked DRAM having N channels, $\frac{1}{N-1}$ of the rows in every bank is reserved for XCC region. In an eight-channel 3D DRAM, each XCC row covers seven rows of data and

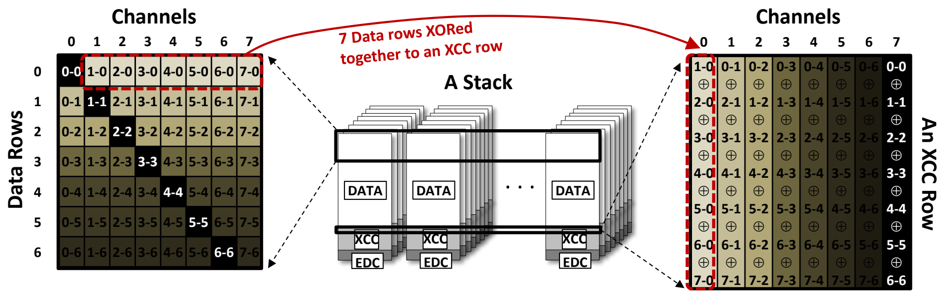


Figure 5: XCC: In a system with N channels, each XCC row covers $N-1$ data rows

hence, the area overhead for XCC is 14.2%. Together with the EDC overhead, which as mentioned earlier, occupies 1.5% of the data rows, the overall overhead due to error detection and correction code in an eight-channel 3D DRAM is 15.7%.

Address Translation for Accessing Error Correction Code: The address translation logic to identify the location of XCC is shown in Figure 4(b). As shown in Figure 5, the XCC data for every seven data rows are placed in a single XCC row. Hence, the row number of the XCC data is the quotient of $\frac{row\_number\_of\_data\_address}{7}$. The XCC row number is then added to an XCC base row register. Then the bank index of the XCC is computed by XORing BPD with bank index of the data, just as in the EDC address computation. The unmodified column and byte addresses of the data are used in the XCC address computation. The only complexity is in identifying the proper XCC channel index. For this purpose, if the channel number is equal to the row number of the data, then the XCC is placed in channel #7. Otherwise, the channel number of the XCC is same as *row_number modulo* 7.

The XCC address computation requires a non-POT computation (modulo 7 for an eight-channel DRAM design). We explored alternate designs that use a POT calculation, but we decided to trade off POT computation for improved area and power efficiency of the design shown above. Note that the XCC address do not need to be calculated for read requests. It is only needed when there is a write operation (off the critical path) or during error recovery (rare). On the other hand, the EDC address is needed for every read and write and hence we optimized EDC address mapping for faster implementation but, compromised with a slightly slower modulo 7 operation for XCC.

3. OPTIMIZING THE COMMON CASE

3.1. Decoupled XCC Update

One potential disadvantage of our proposed design is the write overhead. Whenever a cache line is modified, the old value of the cache line must be XORed out from the corresponding XCC and then the new data should be XORed into the XCC. Therefore, whenever a write request on a cache line arrives, the following steps must be performed sequentially: (1) the old value of the cache line is read from the memory, (2) the corresponding XCC value is read, (3) the old value is XORed out of the XCC, (4) then the new data is XORed into the XCC, (5) the XCC is then written back to memory. Thus, every write incurs an additional two XOR operations, two reads, and one write.

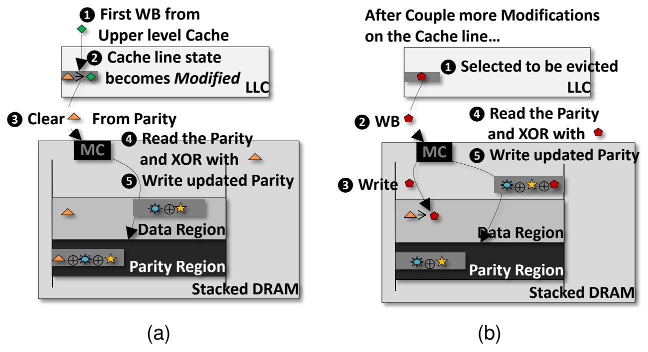


Figure 6: Decoupled XCC update : (a) Step 1 clears old data from XCC during the first modification of last level cache line and (b) Step 2 updates the XCC when the dirty cache line is written back to memory. Step 1 and 2 can be decoupled.

The long sequential chain of operations can be divided into two sequences: 1) clearing the old data from the XCC and 2) adding the new data to the XCC. Figure 6(a) and (b), respectively, show the detailed steps of these two sequences. New data can be XORed only when the data is written back to memory from cache. However, the process of clearing the old data from the XCC can be initiated earlier. The most opportune earliest time to clear the old data from the XCC is when the associated cache line is firstly modified in the last level cache. When the cache line in the last level cache becomes dirty for the first time, the original clean copy of the data can be read from the cache and then forwarded to the memory controller to initiate the clearing of old data from the XCC. This approach works well because cache lines are typically written multiple times before being written back to memory. Hence, there is usually plenty of time between when the first write to a cache line occurs and when the dirty cache line is finally evicted to main memory. Thus, the two processes can be handled at different points in time. When a dirty cache line is eventually evicted, as the old data is already cleared out from the XCC, the new data can be XORed into the XCC faster than before. We call this approach *decoupled XCC update*. It is also possible to schedule the clearing of old data from XCC when memory is not busy with critical read operations.

In our implementation, the clearing of old values are given the lowest priority in the memory controller scheduler. Thus, the decoupled update reduces performance overhead without impacting critical reads.

The error-correction logic must also be aware of the decoupled update process. If an error is encountered after the old data is cleared, but before the new data have been written back, then the recovered data will be incorrect. Rather than burdening the memory controller with tracking exactly which cache lines are actually included in XCC and which lines are already cleared out but still have modified data in the on-chip caches, prior to using an XCC to reconstruct a cache line, we probe the last level cache to force-writeback any remaining modified lines covered by this XCC to memory. At most seven such probes are needed (because the XCC only covers seven cache lines); once the memory updates are complete, then the error correction can continue.

A similar approach was used in the phase change memory (PCM) [17] domain to improve performance by decoupling SET and RESET operations of a write request. They pointed out that the SET operation takes much longer than RESET. By eagerly performing the SET operation whenever the corresponding cache line is modified, when the actual memory write back happens, they only need to conduct the fast RESET operations. Another study used early write back [12] to enhance the performance especially for streaming applications that have enormous amounts of input data. By writing back dirty cache lines from the cache prior to the actual eviction time, the memory traffic becomes more balanced and thereby the overall performance is enhanced. Decoupled XCC update is inspired by these prior approaches although the purpose and the details differ due to its application to RAID5-like parity update and enforcing the correctness of the XCC calculation.

3.2. Caching Error Detection Code

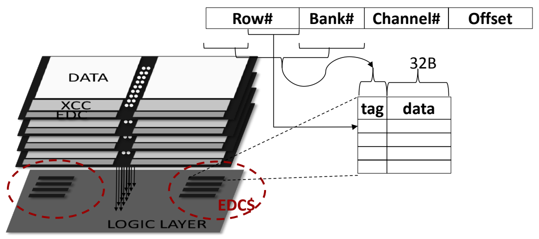


Figure 7: EDC Cache

The proposed approach to place EDC and data in different channels allows the requests to be parallelized. However, this design requires accessing at least two channels for each memory request, thereby halving the effective bandwidth of the memory system and increasing power consumption. Furthermore, the extra EDC access to a bank may conflict with other data accesses leading to more bank conflicts. To alleviate the bandwidth, power, and bank-conflict issues, we propose to use an EDC cache. Several studies showed that it is feasible to implement the cache on the logic layer of the die-stacked DRAM [14]. By using the similar technology, an EDC cache is implemented on the logic layer per channel. The EDC cache entry size is 32B such that one entry can cover all the cache lines in a single DRAM data row. Recall that each 2KB data row (32 cache lines in the proposed implementation) needs only 32B of EDC. Thus, on a first memory access request to any one cache line within the 2KB data row, the corresponding EDC is accessed and the entire 32B EDC data is fetched into the EDC cache. Subsequent memory access requests to other cache lines within the 2KB data row search the EDC cache before initiating an access to fetch EDC from memory. The EDC cache is indexed by using the LSBs of the data row number. The remaining bits in the row index field and the bank index field are used for tag. For fast lookup as well as the power efficiency, we configure the EDC cache to be direct mapped.

We also explored caching XCC. However, XCC cache is accessed during write operations only. A single 32B EDC cache line can store the error detection code of at least 64 cache lines, whereas an XCC cache entry must be a minimum of 64B wide (cache line width) and each XCC cache entry only covers 7 data cache entry when there are 8 channels in a stack. Hence, the coverage provided by each XCC cache entry is significantly lower than EDC cache entry. These limitations reduce the value of caching XCC. We also explored caching EDC and XCC together. Thus this cache can be configured as unified cache such that EDC and XCC are stored together. In the unified cache, the cache entry size is changed to 64B because an XCC is 64B wide. Therefore, an unified cache entry can store EDC for 128 cache lines data or XCC for 7 cache lines. As the cache is indexed by the data address, one bit indicator for distinguishing EDC from XCC is also needed.

3.3. Putting It All Together

As a summary, an example micro-architectural event sequence for a read transaction are as follows.Whenever the memory controller handles a *read* memory request, the EDC cache for the channel that the data resides in is looked up. If there is a EDC cache hit, the corresponding 1 byte EDC is read from the cache. Otherwise, the EDC address that corresponds to the memory access request is generated. Then, the EDC is read from the memory and then the EDC cache is replaced. In the meantime, the data is read from memory in parallel. Note that the data and the EDC are always in different channel. The error detection code for the data that is just read from memory is calculated. In the final step, the EDC read from the EDC cache and EDC calculated from the memory data read are compared. If there is no error, the memory controller returns the data to the last level cache. If there is an error, the XCC process is triggered. The data is rebuilt by XORing the XCC parity and the cache lines that are in the same row position of all the other channels except the channel that the XCC parity is stored.

4. COVERAGE AND OVERHEAD

In summary, we use error detection code that can detect multi-bit errors including a single TSV failure per 64B cache line. XCC can correct multi-bit errors and its coverage scales up to channel-level failure. The area overhead due to the redundant code is 15.7% in an eight channel die-stacked DRAM. The overhead can be further reduced when more channels are used. For example, in a 16 channel die-stacked DRAM, the area overhead is only 8.1% (but the error recovery cost increases). We also proposed an EDC cache per channel for better power efficiency. The tag size changes depending on the DRAM configuration. When a bank is 64MB, the number of banks in a channel is 8, and the row size is 2KB, 9 bits from MSB of data row index plus 3 bits from bank index are used for the tag. As a result, the area overhead due to EDC cache tag is 96B.

5. EVALUATION

5.1. FIT Analysis

To measure the error detection and correction capabilities of various coding schemes we relied on Monte Carlo simulations. We first generated various fault types using the probability distribution of fault types collected from real system observations [23]. The overall FIT rate of each fault type we used in our simulations are shown in the second column in Table 2. We also consider TSV-level failure. Because there is not a known FIT rate for the die-stacked DRAM specific failure mode, we assume that this new failure mode occurs with the lowest probability of all failure modes; we used 35 FIT per device. Clearly, single bit failure mode dominates the overall failures. However, all other failure modes cumulatively account for nearly as many failures as single bit failures. We conducted one million Monte Carlo simulations. For each simulation we essentially pick one of the various fault modes adhering to the probability distribution shown in Table 2. If a single-bit failure is selected for that simulation then only one bit picked randomly from an entire cache line is flipped. If a single-row or a single-bank failure is selected, since we do not know exactly how many bits within a row or bank may have been flipped, we select every bit in a cache line and then randomly flip that bit with a 50% probability. Note that this approach aggressively flips multiple bits with a probability that is higher than what would be seen in practice. Similarly, if a single-TSV failure is selected then we select one bit randomly from the first 128 bits of a cache line and then flip the same numbered bit in all the remaining three 128 bit chunks in the cache line.

Table 1 shows the percentage of faults that are detected and corrected using various error detection and correction scheme combinations. As SECDED code has not only the error detection capability but also the correction capability, we evaluated the SECDED coding in two different combinations: SECDED for both detection and correction, and SECDED for detection only + XCC for correction. For the other error detection codes, XCC is used for error correction. We also

Failure Mode	EDC	ECC	Detection	Correction
Single Bit	8-bit interleaved	XCC	100	100
	two level 8-bit interleaved	XCC	100	100
	16-bit CRC	XCC	100	100
	SECDED	SECDED	100	100
		XCC	–	100
Single Column	8-bit interleaved	XCC	99.50	99.50
	two level 8-bit interleaved	XCC	99.50	99.50
	16-bit CRC	XCC	100	100
	SECDED	SECDED	99.27	16.81
		XCC	–	99.27
Single Row	8-bit interleaved	XCC	95.48	95.48
	two level 8-bit interleaved	XCC	98.72	98.72
	16-bit CRC	XCC	99.99	99.99
	SECDED	SECDED	49.99	0
		XCC	–	49.99
Single TSV	8-bit interleaved	XCC	0	0
	two level 8-bit interleaved	XCC	100	100
	16-bit CRC	XCC	99.99	99.99
	SECDED	SECDED	99.99	0
		XCC	–	99.99

Table 1: Error Detection and Correction Capability (%)

Failure Mode	Injected FIT	Result FIT	
		two level 8-bit inter. + XCC	16-bit CRC + XCC
Single-bit	320	0	0
Single-column	70	3.3E-04	6.6E-05
Single-row	80	1.0136E-03	7.918E-05
Single-bank	100	1.352E-03	1.06E-04
Single-TSV	35	0	3.3E-05

Table 2: FIT rate per device used in the evaluation. 10x higher FIT than the actual FIT rate collected from field test [23] is used. A die-stacked DRAM specific failure mode, TSV failure, is added in this paper

evaluated the single bank failure mode but omitted the result from Table 1 because the result is similar to the single row failure mode.

Among the evaluated coding schemes, SECDED provides lowest error detection and correction capability. For instance, in single-column failure mode multiple bits may be flipped within a single cache line and hence SECDED detects an error 99.27% of the time, but it can only correct those scenarios where only one bit has flipped, which is 16.81% in our Monte Carlo simulations.

Two-level 8-bit interleaved parity performs slightly worse than 16-bit CRC when dealing with multi-bit failures, as explained earlier in the introduction. However, single-TSV failures are handled 100% by our proposed two-level 8-bit interleaved parity while 16-bit CRC provides 99.99% detection capability.

The injected FIT and the modified FIT after running the error detection and correction codes are shown in Table 2. Combined with XCC as a corrector, when two-level 8-bit interleaved parity is used as a detector, the single-bit and TSV failures are perfectly eliminated. In all other cases 16-bit CRC + XCC reduces the overall FIT rate by about 10 times more compared to two-level 8-bit interleave parity . However, due to the long distance (128bit) between the faulty bits as well as the number of faulty bits that is greater than the 16-bit CRC's hamming distance, when it comes to the single TSV failure mode, 16-bit CRC+XCC combination could not

Processors	
Core	4 cores, 3.2 GHz out-of-order, 4 issue width
L1 cache	4-way, 32KB I-Cache + 32KB D-Cache (2-cycle)
L2 cache	8-way, 256KB (4-cycle)

Stacked DRAM	
Size	8GB
Bus frequency	800 MHz (DDR 1.6GHz), 128 bits per channel
Channels/Ranks/Banks	16/1/8, 2048 bytes row buffer
tCAS-tRCD-tRP	25-17-8
tRAS-tRC	25-29

DRAM Energy Parameter (nJ)			
Activation	2.77	Read	4.74
Write	4.74	Precharge	2.43
TSV per access	1.04		

Table 3: Simulation Parameters

Name	LLC MKPI	IPC	Name	LLC MKPI	IPC
leslie3d	13.41	0.75	bzip2	0.19	0.63
soplex	20.27	1.20	gcc	3.88	0.76
libquantum	15.93	0.20	milc	9.93	1.17
lbm	28.99	0.44	namd	0.16	0.59
mcf	17.70	0.12	gromacs	0.12	1.28

Table 4: Characteristics of SPEC2006 benchmarks

perfectly remove the errors. We conclude that either two-level 8-bit interleaved parity or 16-bit CRC can be used for error detection and XCC can be used for error correction in a die-stacked DRAM. If TSV failures are major concern, then two-level 8-bit interleaved parity is better option. In the rest of the results section we use two-level 8-bit interleaved parity to evaluate the performance and energy overheads of using error protection in die-stacked DRAMs.

5.2. Performance and Energy Overhead Evaluation

5.2.1. Settings and Workloads

For evaluating the performance and energy overheads of our proposed scheme, macsim, a cycle-level X86 simulator [6] and SPEC2006 benchmark suite are used. We assume a 8GB die-stacked DRAM having 8 layers. Each layer has two ranks of 512MB each. As described in prior section, each rank is given its own channel. The timing components such as RCD, RAS, RC, and CAS are calculated by using CACTI-3DD [3]. The detailed simulation parameters are described in Table 3. In the prior sections, we described the idea based on a stacked DRAM that has 4 layers and 8 ranks assuming a 4GB DRAM stack. Since we are simulating a larger 8GB die-stacked DRAM a few minor modifications are made to XCC. Instead of XOR data from 7 channels as described in our previous section, we XOR data from 15 channels to create one XCC row.

Workload	Memory Intensive	Memory Non-instensive
H1	lbm, leslie3d, soplex, libquantum	x
H2	lbm, leslie3d, libquantum, mcf	x
H3	milc, soplex, libquntum, mcf	x
H4	mcf × 2, soplex, lbm	x
M1	lbm, leslie3d	bzip2, milc
M2	soplex, libquantum	namd, gromacs
M3	mcf, leslie3d	bzip2, gromacs
L1	x	milc, namd, gcc, bzip2
L2	x	gromacs, milc, bzip2, gcc

Table 5: Mix of workloads for multiprogramming simulation

One can also use two 4GB stacks with each stack configured as described in the prior section. However, as all the 16 channels operate concurrently in both configurations, we believe the evaluation results should be similar or worse than two 4-layer stacked DRAM configuration.

We implemented a new DRAM power measurement module to macsim by referring to DRAMSim [27]. The burst read and write access energy parameters are calculated by using CACTI-3DD [3]. The TSV energy overhead that is also calculated by CACTI-3DD is added to each access energy calculation. For the idle energy calculation, the parameters obtained from the Micron DRAM power calculator [7] are used. We assume that the die-stacked DRAM's idle power of each layer is similar to the same size 2D DRAM. As each layer of our configuration has 1GB memory, we borrowed the parameters of 1GB DDR2 DRAM and multiplied by the number of layers.

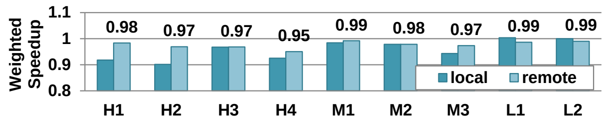
For the performance simulation, we used a representative 500 million instructions slice of each benchmark generated by using PinPoints [19]. Each benchmark's characteristics that are of interest to this study are described in Table 4. Each benchmark has a MPKI metric that measures the *misses per kilo instructions (MPKI)* of the last level cache. These misses are serviced by the die-stacked DRAM. We then made clusters of four benchmarks to simulate the multi-programming environment as shown in Table 5. Among $\binom{10}{4} = 210$ different clusters that grouped out of 10 benchmarks, we chose 9 clusters based on their memory intensity: 4 high, 3 medium, and 2 low memory intensive workloads. Weighted speed-up [22] is used for the performance metric, which is computed as:

$$Weighted\ Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{single}} \qquad (1)$$

To evaluate the performance impact of the proposed EDC/XCC schemes, faults are injected by assuming that each bit has 10^{-4} error probability every cycle. Since the actual FIT rate is extremely small (only 66 failures in a billion device hour [23]), we accelerated failure rates for measuring the performance impact of our proposed design.

5.2.2. Local EDC vs. Remote EDC

Before evaluating the proposed methods, we firstly investigate the performance overhead of placing EDC rows in a separate region from the data rows (labeled as *remote* in Figure 8(a)). We compare remote EDC with a configuration where EDC is embedded immediately next to the data (labeled as *local*). In this experiment, we disabled XCC since we are interested in measuring only error detection overhead. When the EDC and data are in the same row, then a single row buffer access is sufficient to get the data and perform error detection. Local EDC requires non-POT address decoding. However, in our experiment, we assume that there is no additional delay for non-POT computation. Because the error correction capability in both scenarios is the same, we assume BER (bit error rate) is zero for this evaluation experiment. Figure 8 shows the weighted speedup and the energy dissipation when local EDC and remote EDC are used. The weighted speedup and the energy consumption are normalized to scenario that uses no reliability support. These results simply measure the overhead of EDC computation and EDC checks on each memory access. Clearly, providing error protection reduces performance since in all cases both local and remote EDC approaches achieve a normalized performance that is less than one, compared to a no error protection baseline. However, enabling EDC degrades performance by up to 5% in the worst case.

(a) 

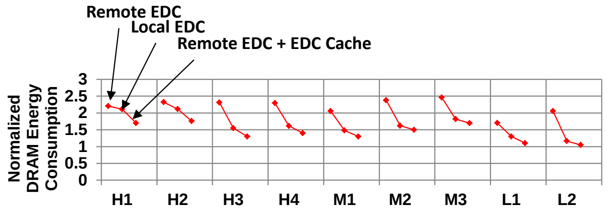
(b) 

Figure 8: (a) Performance and (b) energy comparison of local and remote EDC

Since remote EDC approach accesses two separate channels to access data and EDC concurrently it performs better than local EDC. However, the worst case energy consumption is 2.4 times higher than no error protection support, when remote EDC is used since two channels are accessed for each read. Local EDC opens only one data row for serving both data and the EDC. But when remote EDC is augmented with the EDC cache we can effectively reduce the energy overhead. As can be seen in Figure 8(b), the *Remote EDC + EDC Cache* reduces the energy consumption significantly and the overall energy consumption is lower than local EDC's energy consumption.

5.2.3. Impact of EDC and XCC

We also evaluate the performance impact of EDC and XCC. We measured the weighted speedup under four different scenarios: 1) no reliability support (labeled as *No RAS* in Figure 9), 2) EDC only without fault injection (labeled as *EDC@No Error*), 3) both EDC and XCC are activated without fault injection (marked as *EDC+XCC@No Error*), and 4) 3) with fault injection (marked as *EDC+XCC@BER*=10^{-4}). All the measurements are normalized by *No RAS*.

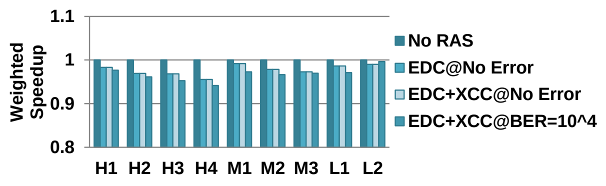


Figure 9: Impact of EDC and XCC on performance

EDC degrades the performance by about 5% as can be seen in the bars named *EDC@No Error* in Figure 9. *EDC+XCC@No Error* shows that the XCC update overhead is negligible since this bar is almost identical to *EDC@No Error*. When we inject errors at a significantly accelerated rate the XCC error correction overhead adds an additional 3% overhead on top of error detection overhead as shown by *EDC+XCC@BER*=10^{-4}. Clearly error correction overhead under realistic scenarios, where error rate is significantly lower than BER=10^{-4}, will disappear. Overall, the performance overhead of the EDC check that is operated in the majority of error free cases is the most significant. However, it is still less than 5% .

5.2.4. Impact of Optimizations

The performance impact of the two proposed optimizations is plotted in Figure 10(a). Recall that there are two optimizations that were proposed. One optimization is a decoupled XCC update approach and the second optimization is to cache EDC/XCC. Simulations are conducted by applying one of the proposed optimization techniques. *No Opt* is the case when the EDC and XCC are used for detecting and correcting the injected fault without any optimizations. *Decoupled XCC* and *EDC Cache* are the cases when the proposed decoupled XCC update and EDC cache optimization are applied one at a time, respectively. When evaluating cache optimizations a 2KB EDC cache is used per channel. The last bar is labeled as *Unified Cache*, where both EDC and XCC are stored in a unified cache with 2KB unified cache per channel. All the simulations are conducted by assuming that each bit has BER of 10^{-4} except for *No RAS*, which is the baseline. Figure 10(b) shows the corresponding energy overhead of all these schemes compared to *No RAS* baseline.

The error detection and correction overhead without any optimization techniques is 8%. Most of the protection overhead is suffered primarily by benchmarks with higher MPKI, as expected. For other benchmarks error protection does not impact performance. The overall performance gain due to the decoupled XCC update (indicated as *Decoupled XCC*) does not look significant. However, the write latency is significantly reduced as can be seen in Figure 10(c). Figure 10(c) shows the average write speedup of *Decoupled XCC* over *No Opt*. The decoupled XCC update can speedup write operations on average by 32% and up to 50% with no additional storage demands. However, write latencies are typically well masked by current architectures and hence the write speedup does not translate into corresponding speedup in the overall execution time in our benchmarks. Hence, the reduction in write latency may improve performance impact for other workloads that exhibit burst write activity.

The EDC cache, on the other hand, works well and recuperates almost all the performance loss due to protection overhead. As we already mentioned, the energy overhead of accessing EDC can also be significantly reduced with EDC cache. The unified cache, that caches both EDC and XCC, does not perform as well as EDC cache alone. The reason why EDC cache is more effective is the fact that a single byte of EDC can hold the parity associated with an entire cache line. Hence, a 2KB EDC cache can hold the error detection

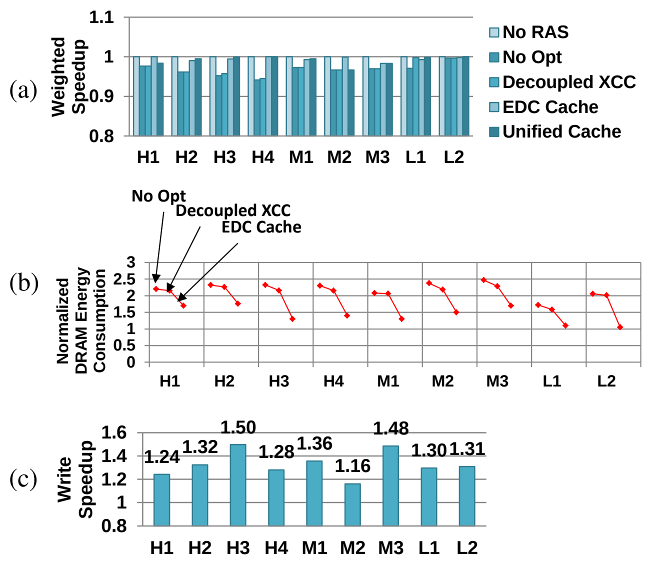


Figure 10: (a) Weighted speedup and (b) energy dissipation of various optimizations and (c) write speedup due to decoupled XCC update

parity for 2048 cache lines, which is 128KB of data. However, an XCC can cover only *the number of channels* − 1 rows and also XCC is accessed only when either the corresponding data is updated or when an error is encountered. Therefore, in many cases, the XCCs loaded to the EDC cache are replaced by the frequently accessed EDCs before it is accessed again. As a result, the unified cache perturbs the EDC cache hit ratio and hence it is better to not cache XCC entries.

6. Related Work

There have been several studies to mitigate the negative impacts of conventional ECC for protecting DRAMs. Virtual-ECC [28] and LOT-ECC [25] brought up the inefficiency issue of the conventional ECC DRAM and proposed approaches that can reduce the performance and power dissipation overhead. They mainly decouple the error detection and correction and have the majority of error-free accesses handled with locally stored error detection code. Virtual-ECC [28] stores the correction code in an arbitrary memory rows with help of operating system. On the other hand, in LOT-ECC paper [25], error correction code is located in the same memory chips with the data but in the different region (one of the several bottom rows). They both successfully reduced the performance overhead for the normal accesses. LOT-ECC especially enhanced the rank level parallelism by squeezing the redundant code into a single rank. However, LOT-ECC still assumes ECC-DRAM-like configuration that uses 9 DRAM chips within a rank, which is impractical to apply to a die-stacked DRAM, and has a fairly high area overhead (26.5%). Virtual-ECC requires operating system support as well.

Udipi *et al.* propose to use RAID5 approach for the error correction [26]. They stripe the parity across the 9 DRAM chips in each rank. The local checksums are used to detect errors and the correction is conducted by XORing the parity and the corresponding data from the rest 8 chips. This approach is not directly applicable to die-stacked DRAM designs: employing additional DRAM chips just for ECC reduces the flexibility to add DRAM layers in die-stack. We lean on RAID5-like parity in our approach but we do not need additional ranks to support error protection. We further resolved the long write latency problem of RAID system by using decoupled parity update and EDC$.

There have not been many studies on reliability support for die-stacked DRAM. A study [21] proposed several approaches for efficient reliability support for 3D DRAM as a last level cache. However, the protection domain of cache and main memory is different as caches only care about modified data protection. Therefore, many techniques proposed in [21] cannot be used for 3D DRAM as a main memory. Another study [2] proposed to use different length ECCs for different layers. This proposal is motivated by the fact that different layers in a die-stacked DRAM have different levels of shielding effect. The outermost (the top) layer is likely to be impacted by more soft error attacks than the inner(lower) layers. But this approach is entirely orthogonal to our proposed approach.

7. CONCLUSION

Die-stacked DRAMs are actively being pursued as main memory replacements for many critical computing systems. But with technology scaling, all memory devices are expected to experience significant increase in single and multi-bit errors. 3D die-stacked DRAM will have the added burden of protecting against through-silicon-via (TSV) failures, which translate into multiple bit errors in a single cache line, as well as channel level failures. To take advantage of the wide I/O capability of 3D DRAM, large chunks of data is contiguously laid out in a single chip; 3D DRAMs place an entire cache line worth of data on a single DRAM chip. Traditional approaches for planar DRAM protection, such as ECC-DIMM and chipkill-correct, do not lend themselves well to protect 3D DRAMs. To address these concerns, this paper adapts several well known error detection and correction techniques while taking into account 3D DRAM's unique memory organization. We first adapt an 8-bit interleaved parity to handle die-stacked DRAM-specific failure modes such as TSV failures. We then, use RAID5 parity, a techniques developed for hard disks which also layout large chunks of data contiguously, for recovering from a wide range of errors from single-bit errors to channel-level failures without the need to splice data across multiple layers. We co-locate error detection and correction codes with the data on the same die-stack to improve scalability. As further optimizations, a two-step decoupled error correction code update process is used to improve write speed, and an error detection code cache is used for improving read/write performance without compromising reliability. The proposed approaches effectively reduce the FIT rate with 15.7% area and almost negligible performance overhead even assuming an aggressive bit-error rate for a sixteen-channel 3D DRAM.

References

[1] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die Stacking (3D) Microarchitecture," in *MICRO*, 2006, pp. 469–479.

[2] L.-J. Chang, Y.-J. Huang, and J.-F. Li, "Area and Reliability Efficient ECC Scheme for 3D RAMs," in *VLSI-DAT*, 2012, pp. 1–4.

[3] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *DATE*, 2012, pp. 33–38.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, vol. 26, pp. 145–185, 1994.

[5] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," in *IBM Technical Report*, 1997.

[6] HPArch, "Macsim simulator." Available: http://code.google.com/p/macsim/

[7] M. T. Inc., "DDR3 Power Calculator." Available: http://www.micron.com/products/support/power-calc

[8] JEDEC, "3D-ICs." Available: http://www.jedec.org/category/technology-focus-area/3d-ics-0

[9] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, "Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding," in *MICRO*, 2007, pp. 197–209.

[10] P. Koopman and T. Chakravarty, "Cyclic redundancy code (crc) polynomial selection for embedded networks," in *DSN*, 2004, pp. 145–.

[11] G. Kumar, T. Bandyopadhyay, V. Sukumaran, V. Sundaram, S. K. Lim, and R. Tummala, "Ultra-high I/O density glass/silicon interposers for high bandwidth smart mobile applications," in *ECTC*, 2011, pp. 217–223.

[12] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens, "Eager writeback - a technique for improving bandwidth utilization," in *MICRO*, 2000, pp. 11–21.

[13] D. Locklear, "Chipkill correct memory architecture," in *Dell Technical Report*, 2000.

[14] G. H. Loh, "A register-file approach for row buffer caches in die-stacked drams," in *MICRO*, 2011, pp. 351–361.

[15] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches," in *MICRO*, 2011, pp. 454–464.

[16] M. Manoochehri, M. Annavaram, and M. Dubois, "Cppc: correctable parity protected cache," in *ISCA*, 2011, pp. 223–234.

[17] M. K. Qureshi, M. M. Franceschini, A. Jagmohan, and L. A. Lastras, "PreSET: improving performance of phase change memories by exploiting asymmetry in write times," in *ISCA*, 2012, pp. 380–391.

[18] M. Santarini, "Stacked & Loaded Xilinx SSI, 28-Gbps I/O Yield Amazing FPGAs," *Xcell Journal*, pp. 8–13, 2011.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS*, 2002, pp. 45–57.

[20] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *MICRO*, 2012, pp. 247–257.

[21] J. Sim, G. H. Loh, V. Sridharan, and M. O'Connor, "Resilient die-stacked dram caches," in *ISCA*, 2013, pp. 416–427.

[22] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *ASPLOS*, 2000, pp. 234–244.

[23] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *SC*, 2012, pp. 76:1–76:11.

[24] TechSpot, "Future Nvidia 'Volta' GPU has stacked DRAM, offers 1TB/s bandwidth." Available: http://www.techspot.com/news/52003-future-nvidia-volta-gpu-has-stacked-dram-offers-1tb-s-bandwidth.html

[25] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, "LOT-ECC: localized and tiered reliability mechanisms for commodity memory systems," in *ISCA*, 2012, pp. 285–296.

[26] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *ISCA*, 2010, pp. 175–186.

[27] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: a memory system simulator," *SIGARCH Comput. Archit. News*, pp. 100–107, Nov. 2005.

[28] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," in *ASPLOS*, 2010, pp. 397–408.

[29] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *MICRO*, 2000, pp. 32–41.

[30] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *ICCD*, 2007, pp. 55–62.