Parity Helix: Efficient Protection for Single-Dimensional Faults in Multi-dimensional Memory Systems

Xun Jian
University of Illinois
at Urbana-Champaign
xunjian1@illinois.edu

Vilas Sridharan
RAS Architecture
Advanced Micro Devices, Inc.
vilas.sridharan@amd.com

Rakesh Kumar
University of Illinois
at Urbana-Champaign
rakeshk@illinois.edu

ABSTRACT

Emerging die-stacked DRAMs provide several factors higher bandwidth and energy efficiency than 2D DRAMs, making them excellent candidates for future memory systems. To be deployed in server and high-performance computing systems, however, die-stacked DRAMs need to provide equivalent or better reliability than existing 2D DRAMs. This includes protecting against channel and die faults, which have been observed in existing 2D DRAM production systems.

In this paper, we observe that memory subsystems can be viewed as a multi-dimensional collection of memory banks in which faults generally affect memory banks that lie along a single dimension. For instance, in die-stacked DRAMs, a die consists of a group of DRAM banks that lie in a horizontal plane while a channel consists of a vertical group of banks spanning across multiple dies. We exploit this fault behavior to propose Parity Helix to efficiently protect against single-dimensional faults in multi-dimensional memory systems. Parity Helix shares the same error correction resources across all dimensions to minimize error correction overheads. For die-stacked DRAMs, our evaluation shows that compared to a straightforward extension of previous schemes, Parity Helix increases memory capacity by 16.7%, reduces memory energy per program access by 21%, on average, and by up to 45%.

1. INTRODUCTION

Memory is fast becoming the power, and therefore, performance bottleneck of modern and emerging computer systems. For example, memory consumes 30% of total power in HPC systems, on average, and up to 40% of total power in data centers [1, 2]. If the conventional DDR3 memory technology were deployed in future exascale systems, memory power consumption will constitute 2.5X the total system's power budget [1]. Many architecture and technology level solutions have been proposed to improve the energy efficiency of the memory system [2, 3, 4]. One promising solution is to stack multiple DRAM dies on top of one another using through silicon vias (TSVs) [1]. Die-stacked DRAMs improve memory bandwidth by 4-20X and energy efficiency by 3X compared to latest generation 2D memories, such as DDR4 and GDDR5 memories [5, 6]. As a result, die-stacked DRAMs may replace 2D DRAMs as the main building block of memory systems in the near future.

Many recent studies show that DRAM faults are a common occurrence in production systems [7, 8, 9, 10]. An uncorrectable error in memory can lead to a system crash, which degrades both performance and availability. As such, many supercomputers and data centers deploy memory error resilience techniques to protect against DRAM faults [7, 8, 9, 10]. A large body of recent works also explore how to provide efficient error resilience for 2D DRAMs [11, 12, 13]. Die-stacked DRAMs will also need error resilience techniques that provide similar reliability in order to be deployed in these reliability-critical systems.

A fault mode that is commonly covered by memory error resilience techniques for data centers and supercomputers, such as the well-known Chipkill Correct [7, 8, 9], is the DRAM chip failure. Due to the large number of DRAM chips in large-scale systems, DRAM chip failures are common. Chipkill Correct protects against the complete failure of DRAM chips in conventional memory systems with 2D DRAMs. Several recent works have started exploring error resilience for die-stacked DRAMs [14, 15, 16]. While prior works protect against up to channel faults, DRAM die faults have not been addressed.

One challenge with protecting against DRAM die faults and channel faults in die-stacked DRAMs is that a DRAM die consists of a group of DRAM banks that lie in a horizontal plane while a channel consists of a vertical group of DRAM banks that span across multiple DRAM dies; as such, a die fault affects a horizontal group of banks while a channel fault affects a vertical group of banks. A straightforward resilience scheme to protect against both fault modes is to deploy dedicated error correction resources to protect against a fault along each physical dimension. However, maintaining dedicated error correction resources for each physical dimension can incur high power and capacity overheads.

In this paper, we propose Parity Helix, a general

978-1-4673-9211-2/16/$31.00

memory resilience technique to efficiently protect a multi-dimensional memory system against single-dimensional faults. We refer to a multi-dimensional memory system as a memory system with a multi-dimensional collection of memory banks; for example, a die-stacked DRAM can be considered as a three-dimensional collection of banks. We refer to a single-dimensional fault as a fault that affects only memory banks located within a subset of physical dimensions in a multi-dimensional memory system; we refer to the subset of physical dimensions that a fault affects collectively as a single logical fault dimension. Die faults and channel faults in a die-stacked DRAM are examples of single dimensional faults. Parity Helix is most effective for multi-dimensional memory systems where each fault typically affects memory banks located only within a subset of the physical dimensions in the memory system. Parity Helix is inspired by the helix - a properly constructed helix can intersect a horizontal plane and a vertical line segment in at most a single point in space. Parity Helix constructs every parity group (i.e., a parity line and the data lines it protects) in a helix-like fashion to ensure that at most a single line per parity group is affected even in the event of the complete failure of all the banks within a single fault dimension.

When applying Parity Helix to die-stacked DRAMs, our evaluations show that Parity Helix reduces memory energy per data access by 21%, on average, and by up to 45% compared to the baseline scheme of protecting against both die fault and channel fault by maintaining dedicated error correction resources for each dimension. In addition, Parity Helix increases available data capacity per die-stacked DRAM by 16.7% compared to this baseline with a $< 4\%$ increase in uncorrectable error probability. Further, Parity Helix incurs less than 1% performance overhead compared to prior schemes that correct channel faults but not die faults. Finally, while our evaluations focus on die-stacked DRAMs, Parity Helix can be easily applied to other memory technologies.

We make the following contributions in this paper:

- We propose Parity Helix, a technique to protect against any single-dimensional fault in a multi-dimensional memory system.
- We evaluate Parity Helix in the context of die-stacked DRAMs; it is the first scheme to protect against up to a complete DRAM die fault in die-stacked DRAMs.

2. DIE-STACKED DRAM BACKGROUND

A die-stacked DRAM, or simply a *stack*, consists of multiple DRAM dies stacked on top of one another using TSVs, which provides both structural support and communication. Since each stack consists of multiple DRAM dies, a stack contains a large number of DRAM banks. The large number of DRAM banks in a stack are grouped into multiple independent channels, as illustrated in Figure 1. Each channel consists of the same

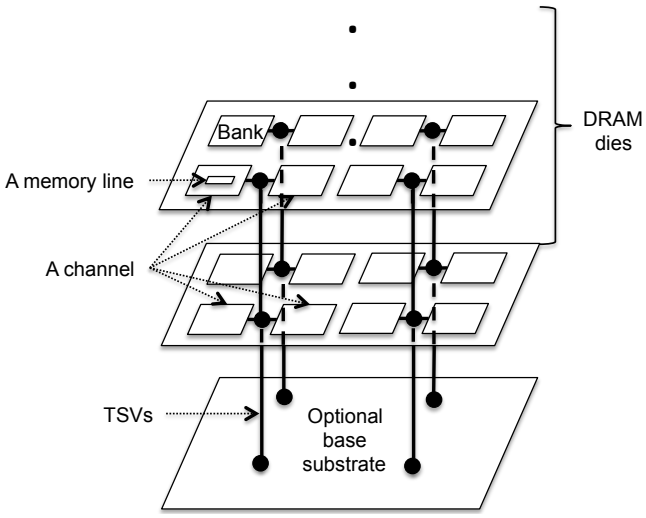

Figure 1: Die-stacked DRAM architecture.

DRAM banks in every DRAM die in the stack[1]; the DRAMs banks within a channel share control and datapath with one another but not with banks in other channels. Within a bank, DRAM cells are organized in rows and columns, identically to conventional memory.

Unlike conventional DRAMs, where each memory line is sourced in parallel from different DRAM banks in different DRAM dies, an entire memory line is sourced from a single bank in a single DRAM die in a stack. This access behavior resembles that of disks, where an entire OS page is typically sourced from a single disk.

Because each line is sourced from a single DRAM bank in a stack, a single fault in a stack can corrupt an entire memory line. As such, a disk-like resilience approach, such as RAID, that can protect against entire line failures is needed [19, 20]. Several recent works on error resilience for die-stacked DRAMs have proposed RAID-like schemes for die-stacked DRAMs. Citadel protects up a complete bank fault (i.e., it can protect against complete failure of every line within a single bank) in a stack by constructing parity lines from lines in different banks and then distributing the parity lines across all the banks in the stack in a manner similar to RAID5 [15] . Sim *et al.* protect against up to a complete channel fault by storing a duplicate of a line of one channel in another channel, in a manner similar to RAID1 [14]. Jeon *et al.* protect up to a complete channel fault [16]; they construct parity lines from lines in different channels and distribute the parity lines across all the channels in the stack in a manner similar to RAID5.

In the context of data centers and supercomputers, large-scale field studies have observed DRAM die failures in 2D DRAMs [8, 9, 10]. Chipkill-Correct, a memory error resilience technique commonly deployed in data centers and supercomputers [7, 8, 9], is used to pro-

[1]This organization is supported by all current die-stacked DRAM standards including HBM [17] and HMC [18]. HBM also supports a second organization where each channel consists only of DRAM banks within the same DRAM die. Our work studies the first organization due to its wider support.

Table 1: Fault coverage summary.

	Complete Bank	Complete Channel	Complete Die
[15]	✓		
[14, 16]	✓	✓	
Parity Helix	✓	✓	✓

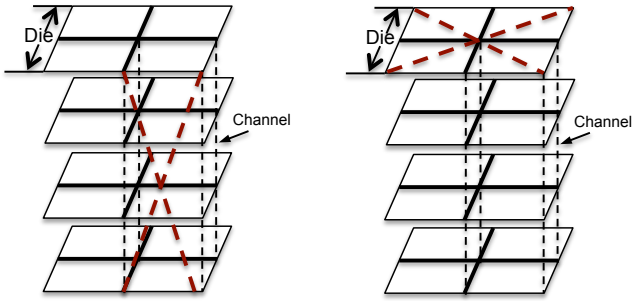

Figure 2: Left: a channel fault. Right: a DRAM die fault. Die-stacked DRAMs are vulnerable to faults along different dimensions.

tect against up to a complete DRAM die fault in conventional memory systems. For die-stacked DRAM memory systems to achieve similar reliability as conventional memory systems, die-stacked memory systems also need to be protected against DRAM die faults. Die-stacked DRAMs may be even more susceptible to DRAM die faults than 2D DRAMs. DRAM dies in a stack are much thinner than 2D DRAM dies to allow the integration of TSVs [21]. The TSVs are also made from different materials from the DRAM dies and, therefore, have different thermal expansion coefficients from the DRAM dies [21]. Both factors make DRAM dies in a stack more vulnerable to cracking under thermal-mechanical stress [21, 22, 23]. In this paper, we explore how to efficiently protect against a DRAM die fault in addition to protecting against faults covered by prior works. Table 1 summarizes the fault coverage of this work with respect to prior works on error resilience for die-stacked DRAMs.

3. MOTIVATION

As shown in Figure 1, a DRAM die in a stack contains a horizontal group of DRAM banks while a channel contains a vertical group of DRAM banks. As such, a DRAM die fault affects DRAM banks that lie in a different physical dimension from banks affected by a channel fault, as illustrated in Figure 2. This makes protection against a DRAM die fault and against a channel fault challenging. For example, RAID across all the channels in a stack (i.e., protecting the channels using a parity channel) does not protect against a DRAM die fault because a DRAM die fault affects multiple channels in the stack. Similarly, RAID across all the DRAM dies in a stack does not protect against a channel fault because the fault affects multiple DRAM dies.

One potential solution is to adapt a stronger RAID

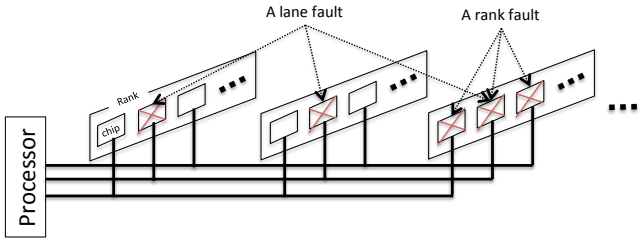

Figure 3: Conventional DRAMs can also experience faults along different dimensions.

implementation, such as RAID6, to die-stacked DRAMs. RAID6 protects against up to two disk failures in a group of disks [24]. There are numerous implementations of RAID6, such as EVENODD, Row-Diagonal Parity, and Horizontal-Diagonal Parity [25, 26, 27], but they all use two parity blocks to protect each data block. RAID6 can be adapted to die-stacked DRAMs by protecting each data line in a stack with two parity lines, where one parity line is constructed from lines in different channels while the other parity line is constructed from lines in different DRAM dies. In this scheme, the first parity line helps to protect against a channel fault while the second parity line helps to protect against a die fault. However, protecting each data line with two parity lines can incur high memory power and capacity overheads (see Section 6). Instead, we seek to provide similar reliability as the RAID6 adaptation while protecting each data line with a single parity line.

Besides die-stacked DRAMs, other memory technologies also experience faults along different dimensions. For example, many conventional memory systems with 2D DRAMs are also multi-dimensional; a rank of chips and a lane of chips (i.e., the same chip in different ranks of DRAM chips that share the same I/O bus bits) in a conventional memory system occupy different dimensions (see Figure 3). A lane of 2D DRAM chips in a conventional memory system can fail together (e.g., due to a stuck-at-1 or stuck-at-0 I/O pin failure [8]), while all the chips in the same rank can also fail together (e.g., due to a Row-Hammer fault [28]) since these chips are controlled by the same command signals. Our proposed solution to protect against any single-dimensional fault in die-stacked DRAMs is easily generalizable to other memory contexts, as will be discussed in Section 7.

4. PARITY HELIX

We propose Parity Helix to efficiently protect against faults that affect a subset of physical dimensions in a multi-dimensional memory system by sharing the same error correction resources across all dimensions to minimize the overheads of the error correction resources. Parity Helix is similar to RAID5, which uses ECC bits dedicated to each line to detect errors and to optionally correct small errors but uses parity lines to correct large errors that affect up to an entire line. Parity Helix differs from RAID5 in how the parity lines are constructed.

4.1 Parity Construction

We define a *sector* as a group of row-wise adjacent line locations in memory. A *data sector* contains data while a *parity sector* contains the bitwise XOR of the contents of a set of data sectors. We divide memory into disjoint and equally-sized groups of sectors called *partitions*, such that each partition is uniquely identifiable by N coordinates, each corresponding to one of the N logical fault dimensions in memory. We call the union of the same sector in every partition a *memory slice*; in other words, each sector within a memory slice is uniquely identifiable by the N coordinates of the partition that the sector belongs to. Within each memory slice, we construct *stripes* of memory, each consisting of a set of data sectors and a single parity sector.

One key aspect of Parity Helix is the appropriate assignment of sectors to stripes within a memory slice. Parity Helix assigns each sector to the k^{th} stripe in the sector's memory slice, where:

$$k(c_1, c_2, ..., c_N) = (c_1 + c_2 + ... + c_N) \ mod \ p \qquad (1)$$

In Equation 1, c_1,c_2,...,c_N stand for the coordinates of the partition that a sector belongs to while p stands for the protection strength. Specifically, p represents the number of faulty adjacent partitions that Parity Helix can correct. Equation 1 ensures that all coordinates that differ in a single coordinate value by less than p evaluate to a different value of k. Note that any group of p adjacent partitions along the same dimension always differ by less than p in exactly one coordinate value among the N coordinate values; as such, Equation 1 assigns all adjacent p sectors along the same dimension into different stripes. This ensures that a fault that affects up to p partitions in the same dimension will affect at most a single sector per stripe; such a fault is, therefore, correctable by the parity in the single parity sector of each stripe.

Because k in Equation 1 ranges from 0 to $p-1$, there are p stripes per memory slice. Therefore, the total number of sectors in a stripe is:

$$S_{stripe} = memory \ slice \ size/p = s_1 \cdot s_2 ... \cdot s_N/p. \qquad (2)$$

In Equation 2, s_i is the range of the coordinate values for the i^{th} dimension. Since there are p stripes per memory slice and $S_{stripe}-1$ data sectors per stripe, the total number of data stripes per memory slice is:

$$S_{ds} = p \cdot (S_{stripe} - 1) \qquad (3)$$

There is a parity sector in each stripe; so the capacity overhead of the parity sectors in a memory slice is:

$$parity \ sectors \ per \ memory \ slice/S_{ds} = p/S_{ds} = 1/(S_{stripe} - 1) \qquad (4)$$

Since p, the protection strength, should be no more than the size of the largest dimension in the memory system, Equation 2 implies that S_{stripe} scales approximately with the average size of the dimensions raised to the power of the total number of dimensions minus one; as such, the capacity overhead of Parity Helix (given in Equation 4) scales inversely proportionally with the average dimension size raised to the power of the total number of dimensions minus one. This is significantly lower than the capacity overhead of protecting each dimension with dedicated parities, which scales approximately proportionally with the total number of dimensions. Moreover, Parity Helix requires updating a single parity when modifying a unit of data. In comparison, protecting each dimension with a dedicated parity requires updating N parities when modifying a unit of data, resulting in higher power and performance overheads.

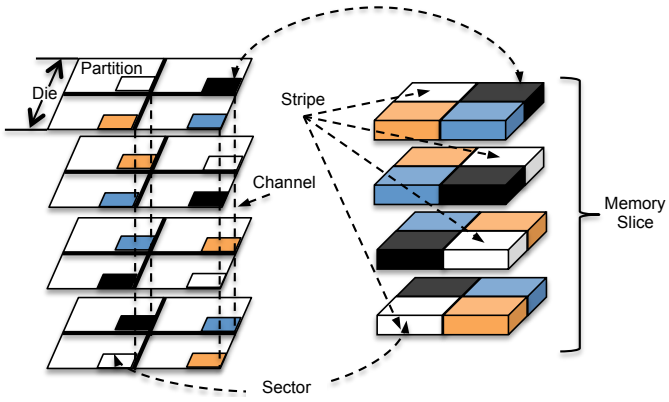

Figure 4: We divide memory into partitions, and partitions into sectors. The same sector in all partitions together form a memory slice. A helix-like collection of sectors in a memory slice form a stripe. Each color represents a different stripe in the figure.

We illustrate Parity Helix using an example memory system with a single stack with four DRAM dies and four channels. Since our goal is to protect against DRAM die faults and channel faults, which affect two different fault dimensions, we also partition the sectors in a stack along two dimensions. Since there are four DRAM dies and four channels in the example stack, the coordinate values of each dimension ranges from zero to three. Since a fault (e.g., a DRAM die fault or channel fault) in the example affects at most four partitions in a stack, p should be set to four to protect against up to a complete DRAM die or channel fault. The left half of Figure 4 illustrates a memory slice in the example stack. The right half of Figure 4 illustrates the stripes within a memory slice; it shows that each stripe rises up like a helix, the inspiration of our work. Equation 1 maps all sectors that belong to the same die (e.g., die c_1) in a memory slice to a different stripe and also maps all sectors that belong to the same channel (e.g., channel c_2) to a different stripe. As such, a single die fault or a single channel fault affects at most one sector per stripe, as illustrated in Figure 5, and is, therefore, correctable by the parity stored in the parity sector in each stripe.

In a memory system with multiple stacks, each stack can either be protected independently or all the stacks can be protected as a single logical stack. To illustrate

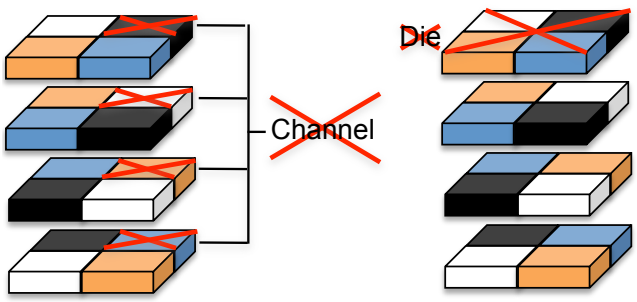

Figure 5: Left: a channel fault affects at most a single sector per stripe. Right: a DRAM die fault affects at most a single sector per stripe. Thus, the single parity sector per stripe protects against up to a complete channel or die fault.

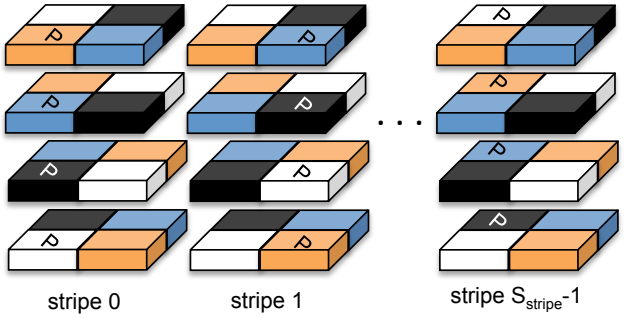

Figure 6: Parity layout. The parity sectors rotate over S_{stripe} memory slices. This distributes the parity sectors evenly across all partitions to eliminate bandwidth bottlenecks due to updating the parity sectors.

the latter case, consider a memory system with n stacks, each with c_1 channels and c_2 dies. All n stacks can be managed as a single logical stack with $n \cdot c_1$ channels and c_2 dies. By configuring protection strength to be $p = max(c_1, c_2)$, Parity Helix can protect up to any single complete channel or die fault among the n stacks.

4.2 Data Layout

Since any single DRAM fault (up to a complete DRAM die fault or a complete channel fault) affects at most a single sector per stripe, selecting which sector in a stripe to be the parity sector has no effect on the error correction coverage. As such, any one of the sectors in a stripe can be selected as the parity sector of the stripe for correctness. However, the selection of which sector in a stripe to be the parity sector can have significant impact on performance. A parity sector needs to be updated whenever a data line (i.e., a line location in a data sector) is written. Updating the parity sector requires a read-modify-write operation to the parity line (i.e., a line location within the parity sector) that corresponds to the line being written; the read-modify-write operation requires two line accesses. Therefore, updating the parity sectors can potentially become a bandwidth bottleneck if they are concentrated in a subset of the partitions in a stack.

To avoid bandwidth bottlenecks due to updating the parity sectors, we evenly distribute the parity sectors across all partitions in a stack similar to how RAID5 evenly distributes the parity blocks across all disks in a disk array [19]. This can be achieved by rotating the parity sector over all possible positions in a stripe. Since each sector in a stripe belongs to a different partition, alternating which sector in each stripe should be the parity sector evenly distributes the parity sectors across all partitions. This can be implemented by letting the same stripes within each group of S_{stripe} memory slices all have a different sector as the parity sector, as illustrated in Figure 6.

Distributing the parity sectors evenly across all partitions can complicate the mapping of physical lines (i.e., lines in the OS physical address space) to line locations within the stack. Since each partition now stores a mixture of both data and parities, one must avoid storing a physical line to a parity sector. We describe a physical line to stack location mapping that distributes the parity sectors evenly across all partitions. When designing such a mapping, we note that adjacent physical lines often exhibit spatial locality; therefore, mapping adjacent physical lines to the same stripe improves the locality of the parity lines in the memory buffer, and for resilience designs that cache ECC bits in the processor, improves the locality of the parity lines in the last-level cache as well. However, mapping adjacent physical lines to different sectors in the same stripe can reduce row buffer hit rate since different sectors in the same stripe belong to different channels and dies and, thus, different rows in memory. As such, we propose first filling up a single sector with the most adjacent physical lines before filling up the stripe with less adjacent lines to achieve both high row buffer hit rate and high spatial locality for the parity sectors.

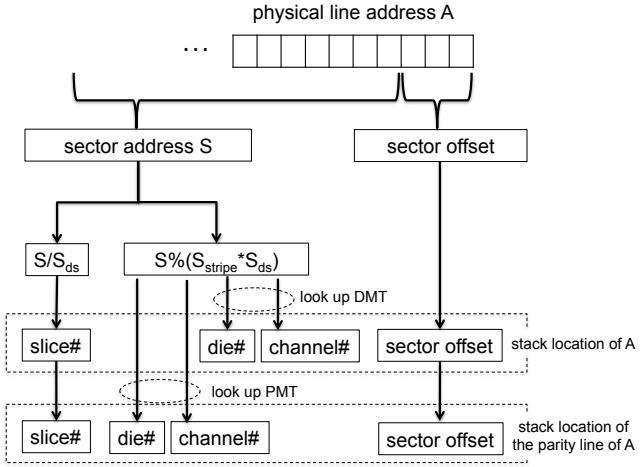

Figure 7: Steps for calculating the location of a physical line in a stack and the location of the parity line that protects the physical line.

Figure 7 summarizes the steps needed to locate a data line and its parity line in a stack given a physical line address A. To locate a data line given a physical line address A, we first compute S, the sector address for the line by taking the first $n - sector\_width$ bits in address A, where n is the total number of bits in address A and $sector\_width$ is the number of bits needed to represent the *sector size*, which is the number of lines per sector; the last $sector\_width$ bits in the line address, therefore, indicates which one of the sector size number of lines within the sector stores the physical line. Next, the memory slice that contains S is found through S/S_{ds}. Recall that a memory slice is the union of the same line locations in all the partitions in a stack; as such, A is mapped to the $(S/S_{ds})^{th}$ sector in the appropriate partition.

Finally, to determine the appropriate partition, which is uniquely identifiable by the die ID and channel ID of the partition, recall that the locations of the parity stripes rotate within groups of S_{stripe} memory slices (see Figure 6); therefore, the locations of the parity sectors start repeating after every S_{stripe} memory slices. As such, we record the channel ID and die ID for S in a data mapping table (DMT) that is indexed by $S \ mod \ (S_{stripe} \cdot S_{ds})$. The $(S \ mod \ (S_{stripe} \cdot S_{ds}))^{th}$ entry in the DMT stores the die ID and the channel ID of S. To maximize the spatial locality of the parity sectors, the DMT maps adjacent physical addresses into the same stripe by storing in adjacent DMT entries die IDs and channel IDs that point to the same stripe (see the paragraph below for details). Similarly, we use $S \ mod \ (S_{stripe} \cdot S_{ds})$ as an index into a parity mapping table (PMT) to identify the die ID and channel ID of the parity sector of the stripe to which A belongs. The DMT and PMT are implemented in hardware in the memory controller for fast lookup.

Populating the DMT takes the following steps. First, compute the stripe number k for all pairs of die ID and channel ID in a stack using Equation 1 and then sort the ID pairs by their respective k values from the least to the greatest. Next, replicate the sorted list of ID pairs $S_{stripe} - 1$ times and concatenate all S_{stripe} lists into a single list. Then, for each m^{th} group of $S_{stripe} \cdot p$ adjacent ID pairs within the single large list (where $m \in [0, S_{stripe} - 1]$), remove the m^{th} ID pair in each group of S_{stripe} adjacent ID pairs; these removed ID pairs point to parity sectors. Finally, insert the remaining ID pairs in the single list into the DMT in order, one ID pair per DMT entry.

4.3 Error Correction

Similar to RAID, when a line encounters an error that is uncorrectable by the line's dedicated ECC, Parity Helix uses the parity line and other data lines in the same stripe to reconstruct the erroneous line. While the parity line of the erroneous data line can be found through the PMT, the other data lines in the same stripe need to be identified using a separate table, which we call the correction table or *CT*. The CT contains a total of $p \cdot S_{stripe}$ entries, one for each of the p stripes in S_{stripe} memory slices. The i^{th} entry in the CT contains the partition coordinates of all the data stripes in a memory slice that belongs to the i^{th} stripe in the memory slice.

Reading out all lines in the stripe to correct an erroneous line can incur significant performance and energy overheads. The problem of having to read out multiple lines to perform error correction using a parity line is common across all RAID-based error resilience solutions, which include prior works such as [15] and [16]. This problem is especially of concern for permanent DRAM faults, which requires repeated error correction. Unfortunately, field studies show that the permanent fault mode is also the most common fault in DRAMs [8, 9]. Since error correction via the parity lines is used only when a data line encounters an error that is uncorrectable by its dedicated ECCs, error correction via the parity lines is only needed for large granularity faults such as the DRAM die fault or channel fault that can cause complete line failures. To reduce the overheads of Parity Helix for error correction via the parity lines, we propose gracefully degrading the available capacity of a faulty stack by retiring (or disabling) faulty channels and dies. This prevents future accesses to faulty channels and dies to prevent frequent error correction via parity lines; in addition, avoiding repeated accesses to faulty channels and dies also alleviates the burden on error detection since repeatedly accessing erroneous lines increases the chance of errors going undetected.

Retiring a DRAM die or channel reduces the size of the corresponding partition dimension size by one. Since the number of dies or channels in a stack is reduced, the data previously stored in the stack can no longer completely fit in the stack and thus some of the data must be migrated outside of the stack. This requires alerting the OS to migrate the content of the top $1/M$ OS pages, where M is the total number of dies or channel per stack, currently mapped to the faulty stack to other stacks or to disk and then permanently retire the corresponding OS pages. This frees up a channel or die worth of free space within the stack such that the memory controller of the faulty stack can reorder in hardware the remaining data in the stack to completely avoid using the disabled die or channel. This requires recalculating the contents of the DMT, PMT, and CT using the new dimensions via the procedures described in Section 4.2 and then making minor adjustments to the tables to completely avoid any access to the disabled DRAM die or channel; the latter can be accomplished by simply increasing all coordinate values in the table by one for all DRAM die IDs or channel IDs greater than or equal to the disabled DRAM die or channel.

4.4 Hardware Overheads

Parity Helix requires adding new hardware structures to the processor to implement the division and modulo operations in Figure 7. These are divide-by-constant and modulo-by-constant operations, with the constants being S_{ds} and $S_{stripe} \cdot S_{ds}$, respectively. The area overhead of a modulo-by-constant circuit is low [29, 14];

Qureshi *et al.* report that it requires only a few hundred logic gates and incurs a two-cycle latency [29]. For the divide-by-constant operation, Drane *et al.* report a $0.001mm^2$ area overhead and 1ns delay for a non-approximate always-round-to-zero constant divider synthesized using the 65nm processing technology [30].

In addition to adding the divide-by-constant and modulo-by-constant circuits to the processor, Parity Helix also requires adding the DMT, PMT, and the CT to the processor. Both the DMT and PMT require $S_{stripe} \cdot S_{ds}$ entries with $ceil(log_2(s_1))+ceil(log_2(s_2))+...+ceil(log_2(s_N))$ bits per entry. The CT requires p entries with $(S_{stripe}-1)\cdot(ceil(log_2(s_1))+ceil(log_2(s_2))+...+ceil(log_2(s_N)))$ bits per entry. Finally, two sets of DMT, PMT, CT are needed during the retirement process because both the mappings before and after the retirement are needed during the retirement process. For our evaluation in Section 6 with 8GB stacks, the total size of the two sets of all three tables is 2016 bytes per stack. Due to the small sizes of each individual tables, the latency overhead of accessing each table is only one cycle.

Finally, to identify faulty die or faulty channel for graceful capacity degradation in a faulty stack (see Section 4.3), we require a counter for each channel and each die in the stack to count the number of errors encountered in the channel or die. If the number of errors for a channel or die exceeds a threshold, we perform read-write-read on all the lines in the channel or die to detect whether errors exist in multiple[2] banks in the channel or die; if so, the channel or die is marked as faulty. Assuming an 8-bit threshold counter for each channel or die, and one bit per bank for each channel or die to indicate whether the bank contains error(s), a total of 64B is required for the evaluated system in Section 6.

5. METHODOLOGY

5.1 Baselines

We compare against three error resilience baselines. The first baseline is similar to [1]; it protects each line with only dedicated ECC bits but not with any parity line. We refer to this baseline as *Non-RAID ECC*. The second baseline is similar to [16]; it protects up to a complete channel in a stack by using RAID5 across channels. We refer to this baseline as *Channel Correct*. We evaluate these two baselines because the former minimizes capacity and power overheads while the latter maximizes protection strength among all prior works on error resilience for die-stacked DRAMs. The third baseline is the simple RAID adaptation to die-stacked DRAMs that we described in Section 3; it corrects up to a complete channel and a complete DRAM die fault in a stack by protecting each data line with two parity lines. We refer to this as *Two-Parity RAID (TPR)*.

5.2 Microarchitecture and Workloads

We use GEM5 [31], a cycle-accurate simulator, to simulate a 16-core x86 processor. Table 2 lists the processor parameters; they are based on Intel Xeon E7 processors. The cache access latencies in Table 2 are obtained using CACTI [32] assuming the 32nm process technology. For generality, our evaluation assumes that the processor implements memory error resilience; our techniques and analysis apply directly when a logic die on a die-stacked DRAM is used to implement error resilience [18]. For the RAID-based resilience schemes, the processor caches the parity lines in the last-level cache of the processor, similar to many prior works in memory error resilience [11, 33, 15]. We use the caching scheme in [33], which not only reduces the number of updates to the parity lines in memory, but also reduces the number of reads to the stale values of data lines needed to calculate the update to the shared parity lines.

We simulate 10 workloads in full system mode; the simulated OS is the Linux 2.6.28.4 OS provided with GEM5. The workloads consist of six NAS Parallel benchmarks [34], two SPLASH2X multi-threaded benchmarks [35], a floating-point mixed workload and an integer mixed workload. Each mixed workload consists of SPEC benchmarks and multi-threaded SPLASH2x benchmarks. We fast-forward each workload in GEM5 functional simulation mode until after the OS has started up and the parallel benchmarks have completed their initialization; we rely on the terminal output in the simulated Linux OS to determine when the OS startup and workload initialization have completed. After the fast-forwarding, we warm up the processor cache in functional mode for 200ms of simulated time; the 200ms cache warmup period is selected to fill up the 32MB last level cache

Table 2: Processor Microarchitecture

Core	16 cores, 3GHz, 2-issue OOO
L1 d-cache, i-cache	2-way, 64kB, 1 cycle
Private L2 cache	8-way, 512kB, 3 cycles
Shared L3 cache	32-way, 32MB, 20 cycles

Table 3: Workloads

Workload	Composition| F.F. Distance	Read BW(GB/s)| Write BW(GB/s)| RSS(GB)
NAS:bt.D	16T | 64.1s	3.4 | 1.2 | 10
NAS:lu.D	16T | 58.1s	24 | 12 | 8.9
NAS:sp.D	16T | 40.8s	36 | 6.0 | 11
NAS:ua.D	16T | 19.0s	5.4 | 1.4 | 7.3
NAS:ft.C	16T | 11.8s	7.0 | 6.0 | 5.4
NAS:mg.C	16T | 7.7s	13 | 11 | 3.3
Splash2X:fft	16T | 75.0s	7.6 | 3.8 | 12
Splach2X: ocean_cp	16T | 5.2	11 | 8.9 | 3.5
mix_INT	4T radix, 4 mcf 4 omnetpp, 4 astar | 8.0s	7.2 | 2.1 | 12
mix_FP	4T ocean_cp, 4 bwaves, 4 cactusADM, 4 wrf | 43.7s	21 | 8.5 | 12

[2]If errors exist in a single bank, we disable the whole die (instead of a single bank) for simplicity.

Table 4: Stack Configuration

Capacity/Dies/Channels	8GB/8/8
Channel Bandwidth	0.5GHz, 144-bit wide
data line size	64B
Banks/Row size	16 per channel/2KB
MSHR/refresh (ms)	32 per channel/64
Activate/Precharge (nj)	1.48/1.43
DRAM Read/Write (nj)	6.87/6.87
TSV/IO transfer (nj)	1.13/5.5
tCAS-tRCD-tRP-tRAS (ns)	18-18-18-26

and turn over the cache content many times. Finally, we simulate each workload in cycle-accurate simulation mode for 15ms of simulated time and report the measurements taken during this period. Table 3 lists the composition and fast-forward distance of the chosen workloads, as well as the corresponding memory characteristics such as memory read bandwidth, write bandwidth, and resident set size.

5.3 Memory Modeling

We evaluate memory systems with two 8GB HBM stacks to accommodate the memory footprint of all of our workloads. We evaluate ECC stacks, which contains 12.5% more bits per line than a non-ECC stack for storing dedicated per line ECC bits [17]. We select the die, channel, and bank count of each stack according to the top configuration available in the HBM specification [17]. For the memory I/O frequency, however, we evaluate a low 500Mhz frequency; since Parity Helix incurs a bandwidth overhead for updating the parity lines, evaluating a lower I/O frequency enables us to stress the stack bandwidth utilization and, therefore, stress the performance overhead due to the bandwidth overhead. To further stress the stack bandwidth utilization, we assign the first contiguous half of the physical address space to the first stack and the second half to the second stack instead of interleaving the physical address space evenly across the two stacks. Our evaluation shows that our workloads utilize up to 48% of the available bandwidth in the more frequently accessed stack out of the two, and 27% on average, under the Non-RAID ECC baseline; we also observed bursty periods within the evaluation interval that utilize nearly 100% of the memory bandwidth. As such, we believe our workloads sufficiently stress the memory bandwidth utilization. We use CACTI-3DD [36] to model the stack power and timing similar to [16]. We use the numbers reported in [37] to calculate the IO energy of a stack. We use DRAMsim2 [38], a cycle-accurate DRAM simulator, to model the timing of the stack. Table 4 summarizes the evaluated stack parameters.

We model memory controllers that prioritize reads over writes and use the first-come-first serve policy and bank-round-robin policy as the intra-bank and inter-bank memory scheduling policies, respectively. We evaluate the open-page row-buffer policy. For the RAID-based resilience schemes, we colocate eight adjacent physical lines in the same DRAM row (i.e., we set the sector size, described in Section 4.2, to eight) to strike a good balance between row buffer hit rates and good parity line hit rates for these resilience schemes. For Non-RAID ECC, which does not require updating the parity lines, we co-locate 32 adjacent lines in the same row since each 2KB row in the evaluated stacks can hold up to 32 64B lines; we interleave adjacent groups of 32 lines across different banks and channels to maximize bank-level parallelism for Non-RAID ECC.

When modeling Parity Helix, we apply Parity Helix to each stack individually. For each stack, both the channel and die coordinate values range from zero to seven (since there are eight channels and eight dies per stack in Table 4); since a channel fault or a die fault can affect up to eight partitions, we set p to eight. As such, the capacity overhead is 1/7 (see Equation 4).

Table 5: Evaluated stack fault rates [9, 16].

Fault Type	Affects up to	Fault rate
Single-bit	one bit per line	0.030 FIT/Mb
Single-column	four bits per line	3.8E-3 FIT/Mb
Single-TSV	four bits per line	41 FIT/TSV
Single-bank	the whole line	9.4E-3 FIT/Mb
Single-channel	the whole line	4.8E-4 FIT/Mb
Single-die	the whole line	4.4E-4 FIT/Mb

5.4 Reliability Modeling

We use combinatorial analysis to calculate reliability. Similar to prior works [14, 15, 16], we assume the fault rates remain constant over time and use 2D DRAM fault rates to approximate fault rates in die-stacked DRAMs since no reliability data is currently publically available for die-stacked DRAMs. We use the DDR3 DRAM fault rates reported in [9]. We pessimistically categorize all faults within a bank that can affect up to an entire line in a bank (e.g., the single-row fault or single-line fault) as the single-bank fault for simplicity. For Parity Helix, we directly map the 2D-DRAM multi-bank fault rate in [9] to our 3D-DRAM die fault rate because both affect multiple banks in a single DRAM die. Similarly, when modelling the channel fault rate, we directly map the 2D-DRAM multi-rank fault rate in [9] to our 3D-DRAM channel fault rate because both affect multiple banks in different DRAM dies with common I/O connection. We use the average fraction of lines affected by a bank, multi-bank, and lane fault reported in [8] to model the fractions of lines affected by a bank, die, and channel fault, respectively, in a stack. For the bank, channel, and die faults, which can affect up to an entire line, we assume that half of the bits in each affected line are erroneous, similar to [14]. Since [9] does not report any TSV fault rates, we use the TSV fault rate used in [16]. Table 5 lists the fault rates used in our evaluation.

Recall from Section 4.3 that Parity Helix can protect against the accumulation of faults by retiring faulty dies or channels at the cost of reduced available memory capacity. When modeling the reliability of Parity He-

lix, our reliability calculation assumes that Parity Helix is allowed to retire up to two channels or two dies or one of each since retiring more memory in a stack may render the stack unusable for the intended applications. We make the same assumption for TPR; similarly, we assume that Channel Correct can retire up to two faulty channels (but not any dies since it cannot correct against die faults). Our reliability calculation assumes that a patrol scrub is performed once every 24 hours to ensure that faulty channels or dies are detected and, therefore, retired in a timely manner; if multiple faults accumulates during the same scrub period - more than the particular resilience scheme can correct without performing retirement -, our calculation assumes that the faults cause an uncorrectable error.

Recall from Section 4, RAID uses ECC bits dedicated to each line to detect errors and/or correct errors and use parity lines to correct large errors that affect up to an entire line. We use the dedicated per line ECC proposed by prior works to evaluate all resilience schemes. The evaluated dedicated per line ECC uses 8B of redundancy for every 64B data line; this matches the $1/8 = 12.5\%$ ECC area overhead provisioned in an ECC stack in the HBM specification [17]. The 8B of dedicated ECC bits is a synthesis of different ECCs used in prior works [14, 15]. The 8B consists of a 32-bit CRC computed over the 64B data line, a 16-bit SSCDSD ECC computed over the 64B data line using 4-bit symbols, and 16 spare bits. The purpose of the 32-bit CRC is to reliably detect errors that cannot be corrected by the dedicated ECC, as proposed in [14]; also as proposed in [14], the address of the line is folded into the CRC to detect address decoder errors. The 16-bit SSCDSD ECC [16] (which is called the SBCDBD ECC in [1]) guarantees correction of errors due to the single-cell fault and single-TSV fault in a stack [16, 1]. The remaining 16 spare bits per line are used to protect against the accumulation of data TSV faults in a stack over time, similar to [15]. Since each data TSV supplies 4 bits per 64B line due to the 128-bit channel data width (see Table 4), the 16 spare bits per line can replace up to $16/4 = 4$ bad data TSVs per channel. Similarly, we also use the spare bits to protect against the accumulation of single bit and column faults. Since all resilience schemes we evaluate use the same codes for error detection, Parity Helix differs only in terms of error correction coverage, not error detection coverage; therefore, we evaluate the uncorrectable error probabilities, but not the undetectable error probabilities.

6. RESULTS

6.1 Reliability

Figure 8 presents the probability of experiencing uncorrectable error(s) in systems with different aggregate physical sizes of die-stacked DRAMs. Parity Helix reduces the probability of having uncorrectable error(s) by over 120X and 100X compared to Non-RAID ECC and Channel Correct, respectively, for all design points shown in Figure 8. This is comparable in magnitude

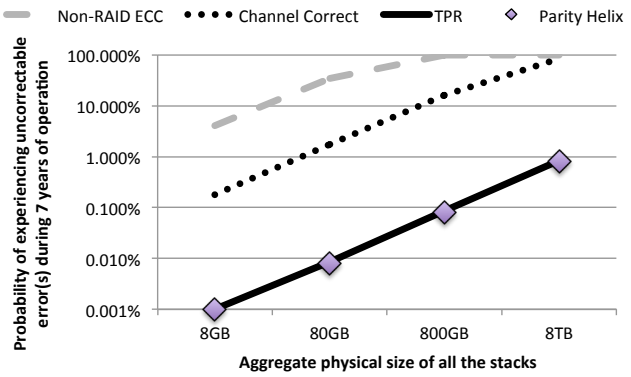

Figure 8: Probability of encountering uncorrectable memory error(s) in systems with different aggregate stack sizes during seven years of operation. Parity Helix reduces the probability of uncorrectable error(s) by over 120X and 100X compared to Non-RAID ECC and Channel Correct, respectively.

to the improvement achieved by Chipkill Correct over SECDED in conventional memory systems (40x [8]). Parity Helix offers remarkable reduction in uncorrectable error probability because it can correct all individual faults in a stack and only fail due to the accumulation of multiple faults; Non-RAID ECC and Channel Correct, on the other hand, can fail in the presence of a single fault in a stack (e.g., the DRAM die fault). Compared to TPR, Parity Helix incurs only 3.6% higher uncorrectable error probability than TPR, even though Parity Helix requires only half as much capacity overhead as TPR for sharing the same set of error correction resources across both dimensions. TPR incurs slightly lower probability of uncorrectable error because it can protect against the accumulation of more faults (e.g., both a die and a channel fault) occurring within each 24-hour scrub period due to its higher redundancy than Parity Helix. However, since the probability of multiple faults occurring within the same 24-hour scrub period within the same stack is small, the difference between the uncorrectable error probability of Parity Helix and TPR is also small (i.e., 3.6%). This small difference can be further narrowed by scrubbing more frequently than once per 24 hours.

Lower uncorrectable error probability reduces performance overhead from checkpoint-restart and improves availability. Consider an HPC system with 64PB of memory [39], and assume a checkpoint interval of 4 hours and a 2 hour checkpoint-restart performance overhead for each uncorrectable error. Also assume that the memory system consists entirely of die-stacked DRAMs. Using the single stack uncorrectable error probabilities (i.e., the 8GB data points in Figure 8), we calculate that Channel Correct incurs 11.6 hours of performance overhead per day, while Parity Helix and TPR incur only 0.053 and 0.051 hours per day, respectively.

6.2 Capacity Overhead

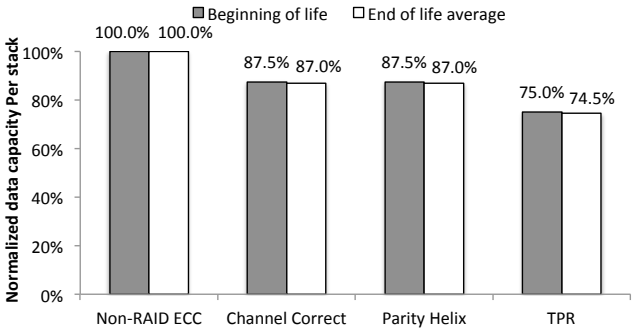

Figure 9: Available data capacity per stack. Parity Helix increases data capacity per stack by 16.7% compared to TPR and provides the same data capacity as Channel Correct.

Figure 9 shows the amount of data capacity per stack (i.e., the number of lines used for storing data lines instead of parity lines in a stack) of different resilience schemes normalized to that of Non-RAID ECC for the evaluated stacks. Figure 9 shows both the available memory capacity at the beginning of a stack's operation, when the stack is fault-free, and the average available memory capacity at the end of the seven years of operation, when some channel/dies have been retired due to developing faults. Parity Helix increases the amount of available data capacity per stack by 16.7% compared to TPR. Parity Helix incurs the same capacity overhead as Channel Correct, which only protects up to a complete channel fault in a stack.

Figure 9 shows, however, that Parity Helix provides 12.5% lower available data capacity than Non-RAID ECC. As will be shown in Sections 6.3 and 6.4, Parity Helix also consumes higher power and performs lower than Non-RAID ECC. On the other hand, Parity Helix protects against complete DRAM die fault - a fault commonly covered in memory systems of datacenters and supercomputers - while the Non-RAID ECC does not; Figure 8 shows that Parity Helix provides over 120X lower probability of experiencing uncorrectable errors than Non-RAID ECC due to the former's higher fault coverage. As such, we believe Parity Helix is a useful and distinctive design point from Non-RAID ECC.

6.3 Memory Energy

Figure 10 shows the memory energy per program read/write access for the different schemes. Parity Helix consumes 21% lower memory energy per program access, on average, and up to 45% lower memory energy per program access (for ft.C). This large difference in memory energy per program access is due to the larger number of memory accesses needed to update the parity lines for TPR. Figure 11 shows that compared to TPR, Parity Helix incurs 24% fewer stack access per program access, on average, and incurs up to 48% fewer stack accesses per program access for ft.C. TPR requires more stack accesses per program access than Parity Helix for three reasons.

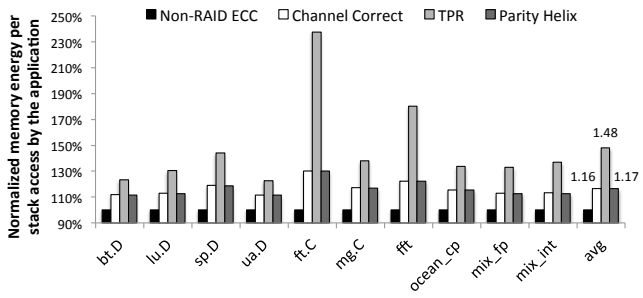

Figure 10: Stack energy per program access. Parity Helix reduces memory energy per program access by 21% compared to TPR.

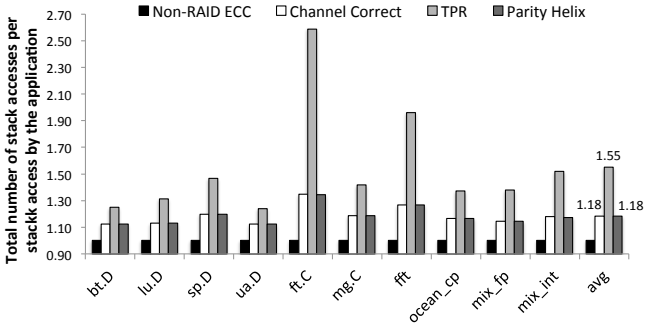

Figure 11: Number of stack accesses per program access. Parity Helix incurs 18.4% fewer access per program access than TPR and incurs the same number of accesses as Channel Correct.

First, TPR protects each line of data with two lines of parity, which effectively doubles the number of additional stack accesses for updating parity lines. Second, TPR incurs higher capacity overhead than Parity Helix (see Section 6.2); this reduces the amount of locality of the parity lines of TPR in the last-level cache since each parity line covers (one) fewer data line than Parity Helix. Third, the two parity lines are constructed from two orthogonal groups of data where one group consists of data lines from different channels and the other group consists of data lines from different DRAM dies; as such, the two parity lines have at most one data line in common. Therefore, while one parity line can protect adjacent physically lines (e.g., with physical addresses A, $A+1$, $A+2$,..., $A+6$), the other parity line can only protect more distant physical lines (e.g., with physical addresses A, $A+6$,..., $A+30$ etc.). This results in poorer cache locality of the latter parity line. Parity Helix, however, only protects each line of data with a single parity line and, therefore, enjoy high cache locality for the parity lines.

Figure 10 shows that Parity Helix consumes roughly the same memory energy per program read/write as Channel Correct. This is because Parity Helix requires roughly the same number of stack accesses per application access as Channel Correct, as shown in Figure 11. This is due to the fact that both of these resilience schemes protect each data line with one parity line and protect the same number of data lines per parity line.

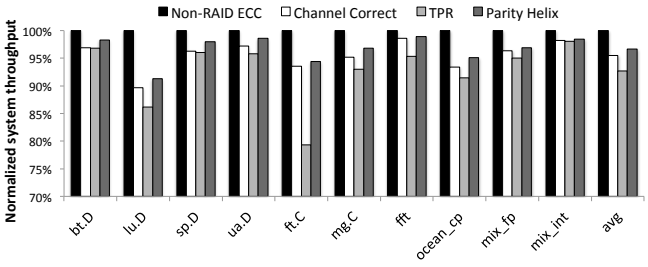

Figure 12: System throughput. The throughput of Parity Helix is 103% and 99% that of TPR and Channel Correct, respectively.

6.4 Performance

Figure 12 presents the system throughput for the different resilience schemes. For all except for mixed_int, we use FLOPS (floating point operations per second) as the metric of measurement. For mixed_int, we measure throughput using the number of committed store instructions per second[3]. Figure 12 shows that the average performance degradations of the RAID-based resilience schemes, TPR, Parity Helix, and Channel Correct, are 4.5%, 7.3%, and 3.3%, respectively, compared to the Non-RAID ECC. TPR incurs the highest performance degradation (62% higher degradation than Parity Helix) due to requiring a large number of overhead accesses to update the two parity lines protecting each data line. Parity Helix incurs 1.0% performance degradation compared to Channel Correct due to the added address decoding latency (see Section 4.4).

The above results assume a fault-free memory system. Recall from Section 4.3 that when a partition encounters an error that the dedicated per line ECC cannot correct, Parity Helix retires the channel or die that contains the partition. On average, the total amount of time per 8GB stack spent on reconstructing and copying data for retirement is only 7.4 milliseconds over the seven-year lifetime of the stack. This low overhead is due to the high bandwidth die-stacked DRAMs; all 8GB of data in the evaluated 64GB/s stack can be accessed in just $8/64 = 125$ milliseconds.

7. DISCUSSIONS

7.1 Applicability to Other Contexts

Parity Helix is applicable to other contexts where a multi-dimensional memory encounters single-dimensional faults. For example, applying Parity Helix to sub-ranked 2D DRAM memory system is straightforward. Recall from Section 3, 2D memory systems are vulnerable to multi-chip failures along different dimensions - both multi-chip failures among the same chip in different ranks and multi-chip failures within a single rank. A sub-ranked 2D DRAM memory system allows the option of accessing individual DRAM chips in a rank [40, 4, 2]; as such,

[3]We did not use the total number of instructions (including integer instructions) as the measurement metric to avoid falsely measuring program progress when threads spin on synchronization variables/barriers.

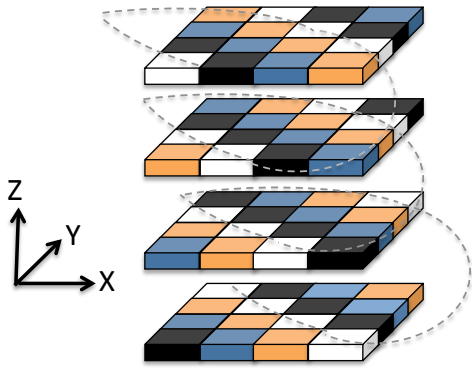

Figure 13: Parity Helix example when single-dimensional faults exist along three dimensions.

a DRAM chip in a sub-ranked memory system is logically equivalent to a single bank in a stack. The chips in the same rank in a sub-ranked memory system are logically equivalent to a DRAM die in a stack. The chips in the same lane (i.e., the same chip in different ranks) in a sub-ranked memory system are logically equivalent to all the banks in a channel in a stack. Due to the one-to-one correspondence of a chip, rank, and lane in sub-ranked memory system to a DRAM bank, DRAM die, and channel in a stack, respectively, Parity Helix can be applied directly to sub-ranked memory systems.

In the above discussions, single-dimensional faults develop along only two dimensions for both die-stacked DRAMs (i.e., across channels and across horizontal dies) and 2D DRAMs (i.e., across lanes and across ranks). However, as described in Section 4.1, Parity Helix is generalizable to even those memory systems that may experience single-dimensional faults along arbitrary number of dimension. Figure 13 shows the stripes within a memory slice for memories where single-dimensional faults can occur along three different dimensions - X, Y, and Z dimensions. Figure 13 shows that each stripe is still similar to a helix, with the difference that each ring in the helix is translated along the X and Y dimensions with respect to other rings.

7.2 Comparison against DRAM Manufacturer Side Techniques

Parity Helix is an architectural technique to enhance memory reliability. Many known techniques at the manufacturer side can also improve memory reliability. However, Parity Helix complements these techniques. For example, while defect testing and burn-in reduces early-life DRAM failures, Parity Helix reduces DRAM lifetime failures. While DRAM circuit-level improvements also enhances DRAM reliability, architectural-level error resilience techniques, such as Chipkill Correct, are still needed for conventional 2D DRAMs to achieve the additional reliability to satisfy the needs of data centers and supercomputers; this is despite the fact that 2D DRAMs have undergone many decades of circuit-level improvements. Similarly, we expect Parity Helix to provide the additional reliability needed by mission-critical systems with die-stacked DRAMs.

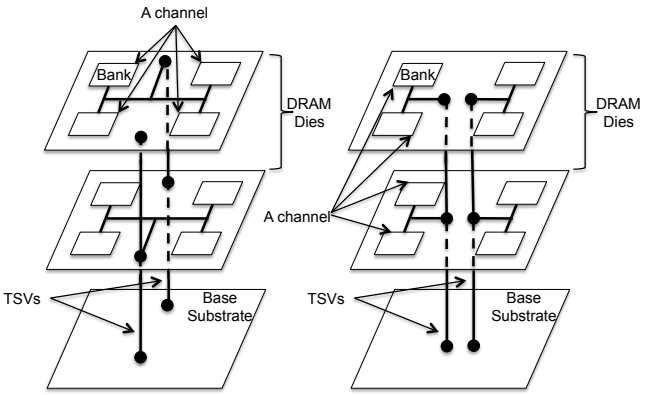

Figure 14: A horizontally partitioned stack (left) requires a different internal wire routing per die. In comparison, all DRAM dies are identical in a vertically partitioned stack (right).

It may also be possible to redesign memory systems such that faults only occur along one dimension. For example, one can design stacks using the alternative organization where a channel consists of a horizontal group of banks that lie within the same die; as such, both the channel fault and die fault affect only the same dimension. However, redesigning memories comes at the cost of losing many of the advantages of the original design. For example, the horizontal channel organization requires all banks in each die in the stack to connect to a set of data/address TSVs dedicated to the die; as shown in the left half of Figure 14, this requires a different wire routing within different dies in the stack and, therefore, a different die mask per die in the stack, which is expensive. Instead of requiring multiple DRAM die masks, the horizontal channel organization can alternatively pay the cost of connecting every bank in a die to every data/address TSVs in the stack and then later disconnecting the unwanted connections (e.g., by blowing fuses). However, the DRAM process allows only a few metal layers [41]; this all-to-all connection complicates intra-die routing for stacks with a large number of dies and channels. The more widely supported vertical channel organization, in comparison, allows every bank to connect to only a single set of data/address TSVs while requiring a single die mask (see right half of Figure 14). Parity Helix gives the option of improving reliability without modifying DRAM design.

8. CONCLUSION

Memory is fast becoming the power and performance bottleneck in current and emerging computer systems. The die-stacked DRAM technology is a promising solution to address the memory bottleneck. However, die-stacked DRAMs pose new challenges for memory error resilience because they are vulnerable to large-granularity faults along different physical dimensions. We propose Parity Helix, a general low-overhead technique to protect against single-dimensional faults in multi-dimensional memory systems that shares the same error correction resources across all dimensions. Parity Helix is most effective for memory systems where every fault can be mapped to a single dimension such that each fault affects only one dimension at a time. When applying Parity Helix to die-stacked DRAMs, our evaluation shows that Parity Helix reduces memory energy per data access by 21%, on average, and by up to 45% compared to the baseline scheme of maintaining dedicated error correction resources for each dimension; Parity Helix also increases available data capacity by 16.7% compared to this baseline. Compared to protecting against only channel fault but not DRAM fault, the strongest level of protection in prior works, Parity Helix incurs only 1.0% performance overhead.

9. ACKNOWLEDGMENT

This work was supported in part by NSF and Cisco. The authors would like to thank the anonymous reviewers for their helpful feedback.

10. REFERENCES

[1] B. Giridhar, M. Cieslak, D. Duggal, R. Dreslinski, H. M. Chen, R. Patti, B. Hold, C. Chakrabarti, T. Mudge, and D. Blaauw, "Exploring dram organizations for energy-efficient and resilient exascale memories," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 23:1–23:12, ACM, 2013.

[2] 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 *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 175–186, ACM, 2010.

[3] D. H. Yoon, J. Chang, N. Muralimanohar, and P. Ranganathan, "BOOM: Enabling mobile memory based low-power server DIMMs," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pp. 25–36, June 2012.

[4] D. H. Yoon, M. K. Jeong, and M. Erez, "Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 295–306, ACM, 2011.

[5] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," *Hot Chips 23*, 2011.

[6] M. O'Connor, "Highlights of the High Bandwidth Memory (HBM) Standard," 2014. http://www.cs.utah.edu/thememoryforum/mike.pdf.

[7] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," *SIGARCH Comput. Archit. News*, pp. 111–122, 2012.

[8] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), IEEE Computer Society Press, 2012.

[9] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory: Positional effects in dram and sram faults," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 22:1–22:11, ACM, 2013.

[10] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory

errors in modern systems: The good, the bad, and the ugly," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 297–310, ACM, 2015.

[11] D. H. Yoon and M. Erez, "Virtualized and flexible ecc for main memory," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 397–408, ACM, 2010.

[12] S. Li, D. H. Yoon, K. Chen, J. Zhao, J. H. Ahn, J. B. Brockman, Y. Xie, and N. P. Jouppi, "Mage: Adaptive granularity and ecc for resilient and power efficient memory systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 33:1–33:11, IEEE Computer Society Press, 2012.

[13] S. Li, K. Chen, M.-Y. Hsieh, N. Muralimanohar, C. D. Kersey, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi, "System implications of memory reliability in exascale computing," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 46:1–46:12, ACM, 2011.

[14] J. Sim, G. H. Loh, V. Sridharan, and M. O'Connor, "Resilient die-stacked dram caches," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 416–427, ACM, 2013.

[15] P. Nair, D. Roberts, and M. Qureshi, "Citadel: Efficiently protecting stacked memory from large granularity failures," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 51–62, Dec 2014.

[16] H. Jeon, G. Loh, and M. Annavaram, "Efficient ras support for die-stacked dram," in *Test Conference (ITC), 2014 IEEE International*, pp. 1–10, Oct 2014.

[17] "JEDEC STANDARD: High Bandwidth Memory (HBM) DRAM," 2013. http://www.jedec.org/standards-documents/results/jesd235.

[18] "Hybrid Memory Cube Specification 2.0," 2014. http://www.hybridmemorycube.org/specification-v2-download-form/.

[19] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, (New York, NY, USA), pp. 109–116, ACM, 1988.

[20] J. Kim and Y. Kim, "HBM: Memory Solution for Bandwidth-Hungry Processors," *Hot Chips 26*, 2014.

[21] Tezzaron, "Our Technology 101." http://www.tezzaron.com/about-us/our-technology-101/.

[22] W. Sun, W. Zhu, F. Che, C. Wang, A. Sun, and H. Tan, "Ultra-thin die characterization for stack-die packaging," in *Electronic Components and Technology Conference, 2007. ECTC '07. Proceedings. 57th*, pp. 1390–1396, May 2007.

[23] H. Guojun, L. Jing-en, and X. Baraton, "Characterization of silicon die strength with application to die crack analysis," in *Electronic Manufacturing Technology Symposium (IEMT), 2008 33rd IEEE/CPMT International*, pp. 1–7, Nov 2008.

[24] "EMC CLARiiON RAID 6 Technology: A Detailed Review," 2007. http://www.emc.com/collateral/hardware/white-papers/h2891-clariion-raid-6.pdf.

[25] M. Blaum, J. Brady, J. Bruck, and J. Menon, "Evenodd: an efficient scheme for tolerating double disk failures in raid architectures," *Computers, IEEE Transactions on*, vol. 44, pp. 192–202, Feb 1995.

[26] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *In Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04*, pp. 1–14, 2004.

[27] C. Wu, X. He, G. Wu, S. Wan, X. Liu, Q. Cao, and C. Xie, "Hdp code: A horizontal-diagonal parity code to optimize i/o load balancing in raid-6," in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pp. 209–220, June 2011.

[28] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 361–372, IEEE Press, 2014.

[29] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 235–246, IEEE Computer Society, 2012.

[30] T. Drane, W.-c. Cheung, and G. Constantinides, "Correctly rounded constant integer division via multiply-add," in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pp. 1243–1246, May 2012.

[31] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *MICRO*, 2006.

[32] "CACTI 5.3." http://quid.hpl.hp.com:9081/cacti.

[33] X. Jian and R. Kumar, "ECC Parity: A technique for efficient memory error resilience for multi-channel memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 1035–1046, IEEE Press, 2014.

[34] "NAS Parallel Benchmarks." http://www.nas.nasa.gov/publications/npb.html.

[35] "SPLASH-2x." http://parsec.cs.princeton.edu/parsec3-doc.htm#splash2x.

[36] K. Chen, S. Li, N. Muralimanohar, J.-H. Ahn, J. Brockman, and N. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pp. 33–38, 2012.

[37] N. Jouppi, A. Kahng, N. Muralimanohar, and V. Srinivas, "Cacti-io: Cacti with off-chip power-area-timing models," in *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pp. 294–301, Nov 2012.

[38] "University of Maryland Memory System Simulator Manual." http://www.eng.umd.edu/~blj/dramsim/v1/download/DRAMsimManual.pdf.

[39] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, (Berlin, Heidelberg), pp. 1–25, Springer-Verlag, 2011.

[40] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez, "The dynamic granularity memory system," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 548–559, IEEE Computer Society, 2012.

[41] Y.-B. Kim and T. Chen, "Assessing merged dram/logic technology," in *Circuits and Systems, 1996. ISCAS '96., Connecting the World., 1996 IEEE International Symposium on*, vol. 4, pp. 133–136 vol.4, May 1996.