2017 IEEE 35th International Conference on Computer Design

Jenga: Efficient Fault Tolerance for Stacked DRAM

Georgios Mappouras, Alireza Vahid[+], Robert Calderbank, Derek R. Hower[*] and Daniel J. Sorin

Dept. of Electrical and Computer Eng.
Duke University
Durham, USA
{georgios.mappouras, robert.calderbank, sorin}@duke.edu

[+]Dept. Of Electrical Engineering
University of Colorado Denver
Denver, USA
alireza.vahid@ucdenver.edu

[*]Corporate Research and Development
Qualcomm
Raleigh, USA
dhower@qti.qualcomm.com

***Abstract*—In this paper, we introduce Jenga, a new scheme for protecting 3D DRAM, specifically high bandwidth memory (HBM), from failures in bits, rows, banks, channels, dies, and TSVs. By providing redundancy at the granularity of a cache block—rather than across blocks, as in the current state of the art—Jenga achieves greater error-free performance and lower error recovery latency. We show that Jenga's runtime is on average only 1.03× the runtime of our Baseline across a range of benchmarks. Additionally, for memory intensive benchmarks, Jenga is on average 1.11× faster than prior work.**

Keywords-3D Memory; HBM; fault tolerance; TSVs

I. Introduction

Current stacked DRAM designs include the Hybrid Memory Cube (HMC) and High Bandwidth Memory (HBM), and these designs are attractive for many systems [1]–[3]. The appeal of 3D stacked DRAM is its ability to deliver far greater memory bandwidth to processors that need it, and researchers have eagerly explored ways of leveraging it [4]–[7].

Stacked DRAM comes with new challenges. First, the stacking introduces new error models, including the potential failure of through silicon vias (TSVs) and failure of a chip in the stack (whose failure has a different impact than the failure of a DRAM chip in traditional 2D DRAM). Second, the traditional solution to DRAM errors—Hamming error correcting codes (ECC) with the error correcting bits on a dedicated DRAM chip—are a poor fit in 3D stacked DRAM.

Prior work [8]–[11] has also identified this fault tolerance challenge, and several clever schemes have been proposed to improve fault tolerance for 3D stacked DRAM. These schemes tolerate failures in rows, banks, and TSVs, but they do so at a considerable expense. These expenses include extra reads before every write and error correction processes that involve accessing and synchronizing multiple blocks.

We introduce Jenga, a new scheme for protecting 3D DRAM, specifically HBM, from failures in bits, rows, banks, channels, dies, and TSVs. Jenga's key innovation—which enables better performance and less complexity than previous work with similar fault tolerance goals—is to provide additional redundancy at the granularity of the size of a cache block instead of across multiple blocks. By not involving multiple blocks, Jenga's procedures for writes and error recovery are much simpler. Jenga requires no modifications to the existing HBM protocol.

II. HBM Background and Error Model

HBM is a 3D memory that allows the integration of multiple memory dies in the same chip (stack).

A. *HBM Organization*

Today HBM provides 4 or 8 dies per stack. Each die is divided into two channels, and all of the channels have their own memory controllers and can work simultaneously and independently from each other. Each channel is organized in bank groups (typically 4 per channel) that contain banks (typically 4 per bank group). The process of accessing a row in a bank is similar to traditional DRAM technologies. A row needs to first be opened in order to read or write it, and accessing an already open row is faster than a closed row.

A key feature of HBM is that it provides a wide interconnection interface for each channel by making use of through-silicon via (TSV) technology [12]. TSVs allow for point-to-point connections that run vertically through the silicon, creating a dense and compact interface.

B. *Current HBM Fault Tolerance*

DRAM has been shown to be vulnerable to transient and permanent bit errors [13],[14]. To tolerate these errors, memories are often protected with error correcting codes (ECC). Although there are many types of ECCs, memory is often protected with codes that provide single error correction and double error detection (SECDED). A commonly used SECDED code is Hamming (72,64), in which a 64-bit dataword is encoded as a 72-bit codeword. In a common implementation of this code with x8 DRAM chips, nine chips are used. The 64 bits of each dataword are striped across the first eight chips, and the 8 ECC bits are on the ninth chip. To tolerate the problem of "chipkill," in which an entire DRAM chip fails, the data and ECC bits can be striped across chips such that the loss of any one chip can be tolerated (i.e., no chip holds more bits of a given codeword than can be corrected by the code) [15].

Naively, one might expect to be able to straightforwardly adapt SECDED codes (without chipkill tolerance) to HBM. In an HBM stack with 8 channels, one could imagine adding a ninth channel for ECC and striping the data across the channels as is done with 2D DRAM. Unfortunately, this design, while good for fault tolerance, would greatly hurt performance. Each read or write would require accessing

1063-6404/17 $31.00 © 2017 IEEE
DOI 10.1109/ICCD.2017.62

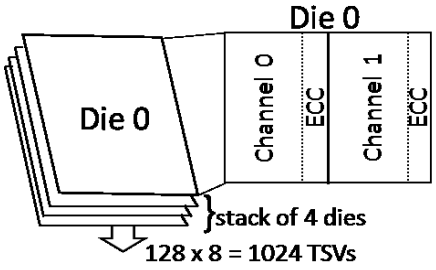

Figure 1: The organization of HBM2

every channel and would thus eliminate much of the bandwidth benefit of using HBM.

Because dedicating a channel in the stack to ECC has many drawbacks, the HBM2 standard [16] specifies memory channels that contain rows that are wide enough to store data and ECC bits together as shown in Figure 1. This collocation of data with ECC is good for performance, because each memory request needs to access only one channel (instead of one channel for data and one for the ECC bits). However, this collocation provides less fault tolerance. Specifically, with collocated data and ECC, the 3D DRAM cannot tolerate failures of rows, banks, channels, dies, or TSVs. For example, a single hard error in one of the 256 TSVs of a given channel would result in 2 (or 4) bit errors for every block access of size 32B (or 64B) from every bank/row of that channel. Then, the occurrence of a single transient error in any of the rows of that channel can result in an uncorrectable error.

III. JENGA

We propose the Jenga design for 3D DRAM in order to provide comprehensive fault tolerance while minimizing costs (due to extra storage) and performance degradation (due to extra accesses). As with any fault tolerance scheme, it requires redundancy, and redundancy inherently has costs.

We designed Jenga for HBM stacks that comply with the HBM2 standard, but there is nothing fundamental about that baseline system model other than that it provides ECC collocated with the data in each row, as shown in Figure 1. Moreover, Jenga requires no modifications to the HBM2 standard[1]. We assume the ECC is a Hamming(72,64) code that provides SECDED at a cost of 12.5%. That is, each 64B cache block requires an extra 8B for ECC, and we assume that block accesses involve all 72B. Like prior work [8]–[11], we assume the memory controllers implement the ECC logic (i.e., errors are detected and corrected by the memory controllers) and can diagnose failed rows, banks, channels, and chips.

A. *Adding Spatial Redundancy*

Jenga is a novel approach to fault tolerance for 3D DRAM, but it shares the same high-level structure as several pieces of prior work [8]–[11]. Although these papers differ in their details, which we describe in more depth in Section VII, they share a common structure.

All four schemes have two levels of error coding: a first level code (denoted L1C) that provides error detection and perhaps a small amount of error correction, and a second level code (L2C) that provides correction for large-scale errors. The L2C in all four schemes protects a given number of blocks, say N, by maintaining a redundant block that is the XOR (i.e., parity) of the N data blocks. These schemes use spatial redundancy to spread the N data blocks and one parity block such that no single fault is unrecoverable.

This structure tolerates many faults at relatively low cost, but it has four drawbacks—all due to the L2C being a parity across multiple blocks—that we seek to overcome with Jenga. Consider a data block A in a parity group with parity block P:

- Every write to A requires first reading the old values of A and P. The old value of A must be XORed out of the old value of P, and the new value of A must be XORed into P. Then A (with its L1C) and P can be written.[2]
- Writing to A and writing to P must be atomic with respect to other writes to other blocks in A's parity group. If P can be cached, this involves modifying the coherence protocol and being careful to avoid deadlock. If P cannot be cached, this involves memory controller support to "lock" P until the writes to A and P complete.
- Correcting an error in A with the L2C requires accessing all blocks in A's parity group. Moreover, until the correction completes, no blocks in A's parity group may be written. Once again, these requirements may affect the coherence protocol and/or memory controller.
- Most prior work [8], [9], [11] (but not [10]) assumes a unified memory controller that is aware of all memory. However, each HBM channel has its own unique controller that is aware of only that channel's memory. Without a unified controller, the LLC bank that submits a memory request would be burdened with calculating parity bits, issuing extra requests to load and store the parity bits, and identifying and correcting errors.

B. *Jenga's Finer-Grained Spatial Redundancy*

Jenga overcomes the undesirable aspects of prior work with one key innovation: adding redundancy at the granularity of a single block, instead of across multiple blocks. Every memory block A (72B) is divided into two sub-blocks, A_1 (36B) and A_2 (36B). We then perform a bit-wise XOR between A_1 and A_2 to produce a redundant sub-block A_3 (36B) that is the parity of sub-blocks A_1 and A_2. Thus, reading any two of the 3 sub-blocks can recreate our data. Concatenating A_1 and A_2 recreates block A or we can perform a bitwise XOR of A_1 (or A_2) with A_3 in order to generate A_2 (or A_1) and then concatenate the two to again recreate block A.

[1] Jenga does require the bit-width of the channel to be smaller than the cache block size, which is true of both HBM2 and HMC, which has a channel width of 16 bits.

[2] There are ways to reduce that penalty (e.g., with caching and proactive XORing of the old values of blocks), but we prefer to avoid it entirely.

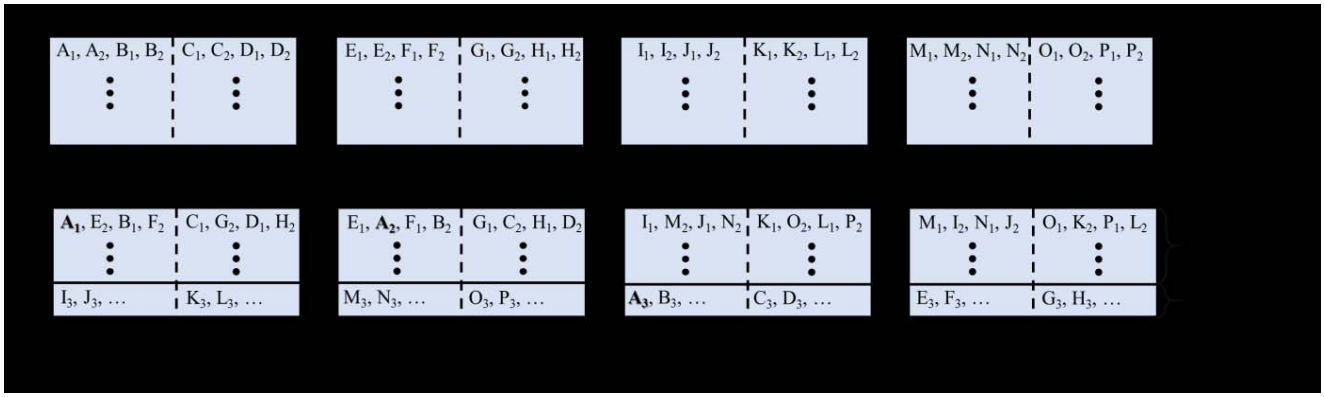

Figure 2: Comparing Jenga to baseline HBM2, in terms of mapping (sub-)blocks to channels

Like prior work, Jenga needs spatial redundancy to tolerate channel and die failures; we cannot have multiple sub-blocks vulnerable to a single failure. Jenga stripes the three sub-blocks in different channels and different dies, and it does this striping in a modulo-like fashion as shown in Figure 2. To explain the mapping, we use an example in which, for unmodified HBM (i.e., without Jenga), block *A* would have been stored in channel 0, row 0, and columns 0 and 1. Assume each DRAM column is a half-block wide.

We treat the parity sub-block A_3 differently from the original sub-blocks A_1 and A_2. To store A_3, we first need to find space in our HBM. Because we do not want to add dies (i.e., assume HBM stacks with an arbitrary number of dies), we choose to sacrifice the host-visible capacity of our memory. As illustrated in Figure 2, Jenga uses 2/3 of the total capacity to store data and reserves 1/3 of the total capacity to store the additional parity sub-blocks like A_3. In our example, Jenga stores sub-block A_3 in channel 4 of that reserved space. Observe that this space is enough to store all the extra sub-blocks (A_3, B_3, …) corresponding to blocks that have their first sub-block (A_1, B_1, …) in channel 0.

There are many functions that work for mapping the original sub-blocks A_1 and A_2, and we can choose from among those that best preserve DRAM row locality. In our example, sub-block A_1 is in channel 0, and it is at row 0, column 0. The mapping function must ensure that (a) a sub-block from some other block gets mapped to channel 0, row 0, column 1, and (b) sub-block A_2 gets mapped to some column 1 in row 0 of another channel (channel 2 in our example). The function we implement and present in Figure 2 interleaves the sub-blocks such that A_1 and B_1 are collocated in the same row, A_2 and B_2 are together in a different row, and A_3 and B_3 are together in a third row.

C. *Reads*

On a read miss in the last-level cache (LLC), the LLC bank acquires a block of data by accessing two different HBM channels on two different chips. Thus, a memory read generates two separate half-block read requests. Although the overall amount of data we read does not change, we do access two channels rather than one. Additionally, a read is only completed once both channels have replied.

Because Jenga needs only two of the three sub-blocks to obtain a block, we have the opportunity to choose which two we access, and there are several ways one could try to exploit this freedom of channel selection. For example, if one channel is malfunctioning (e.g., due to multiple TSV failures), then we always want to access the other two. Additionally, it would be possible to select the channels with the least occupied queues. To keep our implementation simple, Jenga selects the channels depending on their Manhattan distance from the LLC bank that issued the corresponding request.

Figure 3(a) presents a timing diagram of an LLC read miss in Jenga in the absence of errors. The LLC controller issues two read requests to channels 0 and 2 in order to retrieve sub-blocks A_1 and A_2. When both channels reply, data block *A* is reconstructed by concatenating the two sub-blocks.

D. *Writes*

To perform a memory write, Jenga generates three separate half-block accesses to different channels on different chips. In our running example, writing block *A* requires us to update all three sub-blocks A_1, A_2 and A_3 and thus access channels 0, 2 and 4. Jenga also requires the computation of A_3 as the logical XOR of sub-blocks A_1 and A_2, but this is simple and adds minimal overhead.

E. *Error Detection and Correction*

The baseline HBM against which we compare Jenga has SECDED coding. (We discuss in Section IV.E the possibility of using different codes.) If SECDED cannot correct an error, Jenga uses the third sub-block to recreate the correct data. For example, assume that we are reading block *A* and thus issued read requests for A_1 and A_2. As shown in Figure 3(b), if SECDED cannot correct errors in sub-block A_2, then the LLC controller requests A_3. When the LLC has A_1 and A_3, it can bit-wise XOR them to recover A_2 and thus obtain *A*.

F. *Half-Block Accesses*

Typically, the majority of accesses to memory are at least one block. However, systems occasionally perform half-

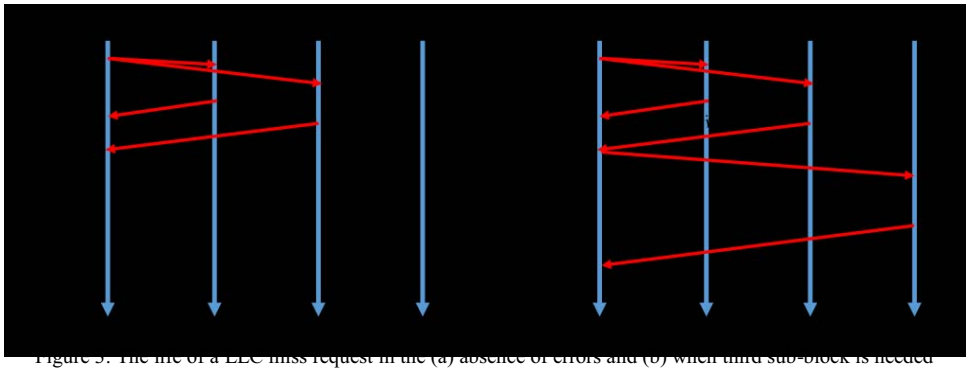

Figure 3: The life of a LLC miss request in the (a) absence of errors and (b) when third sub-block is needed

block accesses, for a variety of reasons, including segmented cache designs [17]–[20], I/O operations, or if the LLC maintains dirty bits at a sub-block granularity

For Jenga, half-block reads require less work. Instead of performing the two sub-block reads required to obtain a full block, Jenga would need to perform only one sub-block read. (An uncorrectable SECDED error in that sub-block could be reconstructed from the other two sub-blocks.) A half-block write requires reading two of the three sub-blocks before writing these two sub-blocks to correctly compute the parity sub-block. This need to read before writing is a drawback that prior work has for *all* writes (not just sub-block writes), and that Jenga has only for sub-block writes.

G. *Costs*

We now summarize the costs of Jenga, bearing in mind that all fault tolerance schemes require redundancy and more fault tolerance incurs more cost.

- Storage: For each block, Jenga adds another half-block of storage. Thus, Jenga memory has 2/3 of the host-visible memory as the baseline HBM2.
- Read latency: Each read requires waiting for the completion of two concurrent half-size accesses to different channels on different chips. Ideally these latencies overlap, but that overlap will not be perfect. Each half-block access takes somewhat less time than a full-block access, which slightly benefits Jenga.
- Write latency: Each write requires three half-size writes. Writes are rarely on the critical path of performance, and thus only bandwidth really matters.
- Bank conflicts: Jenga may introduce more HBM bank conflicts due to having more accesses (even if each access is smaller), and these bank conflicts may degrade performance. (On the flip side, Jenga may increase the number of row hits.)
- Interconnection network bandwidth: Reads require approximately the same bandwidth as the baseline (two half-block requests and responses compared to one full-block request and response), if we ignore packet header overheads. Writes require 1.5× the bandwidth as the baseline (three half-size requests and responses compared to one full-size request and response).
- Power/Energy: The power and energy overheads of Jenga are a function of how much extra work it does and thus track closely with its bandwidth overheads.

IV. Error Detection and Correction Analysis

In this section, we analyze Jenga's ability to tolerate errors. We consider one block, A, and its three sub-blocks. If at least two of its three sub-blocks are correct, Jenga will produce correct data when reading A. Thus, the analysis devolves to determining when at least two of the three sub-blocks are correct.

An additional subtlety arises because the SECDED code is at a finer granularity (72 bits) than a sub-block (36B per sub-block in DRAM). Consider the case where A_1 has an uncorrectable SECDED error in its first 72 bits, and A_2 has an uncorrectable SECDED error in its last 72 bits. Assume A_3 is error-free. Even though two of the three sub-blocks have uncorrectable SECDED errors, Jenga could still reconstruct A because the first 72-bit codeword can be reconstructed from A_2 and A_3 and the last 72-bit codeword could be reconstructed from A_1 and A_3. To achieve this benefit, we would need to modify the memory controllers such that, when they detect an uncorrectable SECDED error, they still send to the LLC the data and the locations of the codewords with the uncorrectable SECDED errors.

A. *Individual Bit Errors*

Any number of 72-bit codewords with a single bit error will be corrected by the SECDED code. A 2-bit error in a 72-bit codeword in a given sub-block will be uncorrectable by SECDED, but it will be corrected by using the other two sub-blocks as long as neither of those sub-blocks has a multi-bit error in the same codeword position.

B. *TSV Failures*

Any single TSV failure will cause two bit errors in one of the three sub-blocks. These bit errors are separated by 127 bits and thus will be corrected by the SECDED code. However, once a TSV has failed, a subsequent bit error can cause an uncorrectable SECDED error in one 72-bit codeword in one sub-block. This uncorrectable SECDED error will force Jenga to fetch the third sub-block and thus produce correct data.

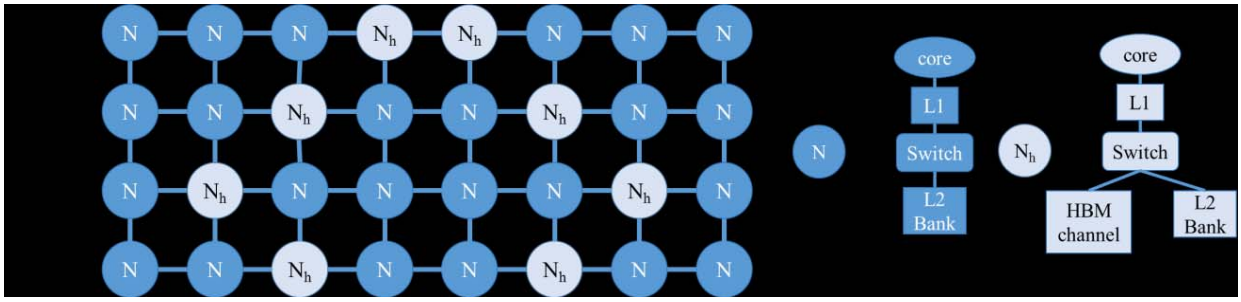

Figure 4: The nodes of the system and how they are interconnected. Only nodes marked as N_h connect to an HBM channel

Multiple TSV failures can be tolerated if the failed TSVs are separated by at least 64 TSVs, in which case the corresponding bits that are read through the faulty TSVs are protected by separate SECDED codes. Additionally, in the absence of individual bit errors, any two arbitrary TSV failures can be corrected.

C. *Row and Bank Failures*

Each of these large-scale failures can cause one of the three sub-blocks to be uncorrectable by SECDED. Jenga will still have two correct sub-blocks and thus produce correct data. Multiple row and bank failures can be tolerated as long as no two rows or banks that store sub-blocks of the same cache block fail. For example, in the case that both the rows that hold sub-block A_1 and sub-block A_2 fail then block A can no longer be recovered. However, considering the number of rows and banks in all channels this is an unlikely event.

D. *Chip and Channel Failures*

Chip failure is a well-known failure mode for DRAM chips [15], [21], and channel failure can occur for a variety of reasons including a permanent fault in a channel controller. Because any chip holds no more than one of the three sub-blocks, a chip failure can be tolerated like a channel failure. Jenga tolerates one channel or chip failure, and it can tolerate two channel failures if both channels are on the same chip.

E. *Using Different Collocated Error Codes*

In Jenga, we assume the use of a typical SECDED code, like Hamming(72,64). However, we could use other codes that offer different trade-offs between cost and fault tolerance. A straightforward change, that would improve fault tolerance but reduce the host-visible capacity, would be to strengthen the code from SECDED to DECTED (double error correcting and triple error detecting) or even stronger. A slightly different approach, with a subtler trade-off, would use a code like CRC that has very strong error detection capability but no error correction ability. With CRC, Jenga would detect some errors that are missed by SECDED (or DECTED) and enable them to be corrected using the third sub-block. However, with CRC, Jenga would not be able to correct *any* errors without having to resort to fetching the third sub-block.

V. Experimental Methodology

A. *Simulation Methodology*

We simulate Jenga with the gem5 simulator [22] in full-system mode. We use the Ruby memory system simulator for modeling and simulating our memory system (interconnection network, caches, HBM, coherence protocol etc.) in detail. By using Ruby, our simulation results will include any traffic overheads and delays that may occur because of the additional read and write memory requests that Jenga imposes.

B. *Benchmarks*

Our benchmarks are a set of multithreaded programs from the PARSEC benchmark suite [23]. We include benchmarks with different memory demands and we run them under large size inputs with 32 threads. The benchmarks with high memory demand, like canneal, are most relevant, but we run all of them for completeness.

C. *System Model*

System Organization. The system configuration is summarized in Table 1. We simulate a multicore chip with 32 cores that communicate through a mesh interconnection network. The 8 HBM channels are connected in a diamond-like fashion across the mesh [24] as shown in Figure 4.

HBM. We simulate unmodified HBM with a 4 die stack. Each die consists of two channels (i.e., 8 channels in total). Each channel connects to our system through 128 TSVs. Each channel contains 4 bank-groups that consist of 4 banks each. A bank contains 8192 rows that store 2KB of data and the appropriate ECC bits. In total, each channel can hold 256MB (i.e., 2GB per stack). More details about the timing specifications of the HBM are in Table 1.

D. *Comparison Schemes*

We compare Jenga against two schemes. First, we consider a baseline system (denoted "Baseline" in the figures) that uses unmodified HBM with just the collocated SECDED code. Second, we consider a scheme that, like prior work, uses an L2C that maintains parity across multiple blocks (denoted "PW", for "prior work"). To make the comparison fair—so that Jenga and PW have the same raw capacity—we assume that the L2C in PW uses two blocks from different

channels and logically XORes them to produce a parity block. The three blocks in each parity group are spread across different channels in a modulo fashion (similarly to Jenga) in order to ensure spatial redundancy.

VI. Evaluation

We now present the results of our simulations. We focus on measuring the impact of Jenga on performance in the absence and presence of memory errors.

A. *Error-Free Performance*

In Figure 6, we present the runtime for all the benchmarks, normalized to that of the baseline. The benchmarks are organized, from left to right, in order of decreasing memory demand [23], because memory demand is a key factor in these performance results.

The key result is that the performance of Jenga is very close to that of Baseline. For workloads with light memory demand, that is unsurprising. However, even for workloads with greater memory demand, like canneal, performance is almost identical (less than 1.03× runtime comparing to Baseline). Conversely, we observe that PW suffers a significant performance overhead for benchmarks that are memory intense. To highlight the impact of memory demand, we ran the same experiment with a smaller L2 cache; the results in Figure 5 show that performance degradation tends to increase. More specifically, Jenga achieves an average speedup of 1.11× over PW for the 2MB LLC across the 3 most memory intense benchmarks (canneal, facesim, dedup).

These results are primarily due to latency differences, although bandwidth differences can potentially have an impact. The key difference in latency is that Jenga, unlike PW, does not have to issue reads before each write. Jenga's reads can take a bit longer than PW's, because Jenga has to wait for both sub-block reads to complete, but those latencies often overlap considerably. Both Jenga and PW place more bandwidth demand on the memory and on the interconnection network, compared to Baseline, but modern interconnection networks tend to be highly overprovisioned [25], [26], and thus the increase in bandwidth has a relatively small impact on performance.

One anomalous result is that Jenga is sometimes marginally faster than the Baseline. This is a minor artifact and not a claimed benefit of Jenga, and there are two phenomena that contribute to it. First, Jenga's reads are half the size of Baseline's reads and thus each one can be completed slightly faster; if the two reads by the L2 overlap their latencies perfectly, a Jenga read will incur a slight benefit. Second, because the LLC issues two half-block read requests for each read miss, it opens two different DRAM rows at the same time. For certain memory access patterns, the second row opening is useful for future requests (i.e., like a prefetch) and the latency to open the row is overlapped with that of the first row.

Impact on Network Traffic. Jenga could impact performance indirectly if its multiple requests and responses cause significantly more congestion in the interconnection network. To study this phenomenon, we measure the average utilization of the on-chip mesh network. Although only writes to memory generate more data than the baseline (96B rather than 64B), the multiple requests (data and packet header) need to travel through the mesh in order to find the destination channel. Our results (not shown) reveal that network utilization remains low, regardless.

Conclusions. From these results, we conclude that although the error-free performance degradation due to Jenga is small, it is more evident for memory intensive applications. It is

Table 1: System Configuration

Multicore Processor Chip	
Cores	32 out-of-order cores
L1 Caches	private per-core, 32KB
L2 Caches	shared 4MB, 32 banks
Interconnection Network	2D mesh
Coherence Protocol	MESI with state at L2 cache
HBM Configuration	
Number of Dies	4 dies in one stack
Channels: number and size	2 @ 256MB each
Banks per Channel	16
Bank Groups per channel	4
Number of TSVs per Channel	128
HBM Timing Details	
Clock Cycle	t_{CK} = 2ns
Row Precharge	t_{RP} = 15ns
Row to Column Address Delay	t_{RCD} = 15ns
CAS Latency	t_{CL} = 15ns
Row Active Time	t_{RAS} = 33ns
Burst Delay	t_{BURST} = 2ns

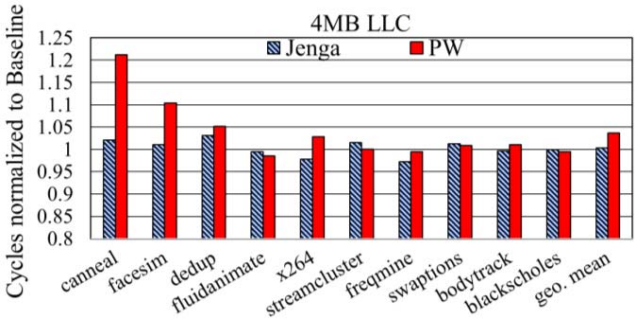

Figure 6: Runtime normalized to Baseline – 4MB LLC

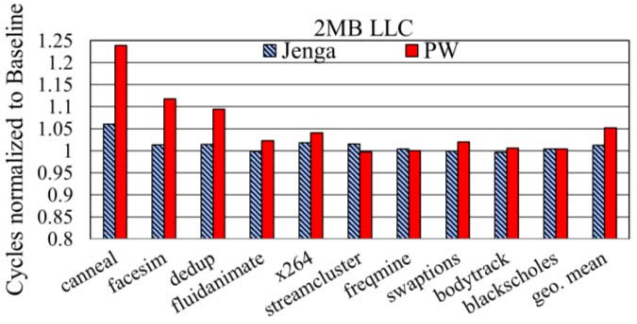

Figure 5: Runtime normalized to Baseline – 2MB LLC

possible that benchmarks with even more memory demand than parsec—or systems with many more threads—could reveal a larger performance penalty for Jenga. However Jenga achieves a significant performance gain comparing to PW and this performance gap only increases for higher memory demand benchmarks.

B. *Performance in Presence of Errors*

We consider two scenarios. First, as an extreme stress test, we consider the worst case in which every pair of read requests results in an uncorrectable error in the second response to arrive at the L2. Thus, after the second response arrives, the L2 issues an additional request to the third half-block in order to recreate the block. Second, we present a somewhat less extreme scenario in which 1% of every pair of read requests results in an uncorrectable error in the second response to arrive at the L2.

We observed (graph not shown) that even for the totally improbable worst-case error scenario on our most memory intensive benchmark, Jenga incurs an overhead that is less than 40% (~70% for PW). Jenga with 1% error rate has performance only slightly worse than that of error-free Jenga. We can safely conclude that Jenga's performance impact in the presence of occasional errors will be minimal.

C. *Capacity Cost.*

The host-visible capacity of Jenga is 2/3 of that of Baseline. Although it is technically possible to increase Jenga's host-visible capacity by using even finer granularity redundancy (e.g., quarter-blocks), doing so would waste HBM's bandwidth. HBM uses 128 TSVs twice in one DRAM cycle to transfer 32B. Thus, transferring anything less than a half-block (32B) during that cycle is wasteful. Although we assume PW to have the same capacity cost as Jenga, we could easily increase PW's host-visible capacity by creating parity across more than two blocks. However, that would result in even higher performance overheads for writes and error recoveries, as blocks that share the same parity need to synchronize during writes and recoveries. Additionally, increasing the size of a parity group decreases fault tolerance.

VII. Related Work

There are four prior papers that stand out in fault tolerance for 3D DRAM [8]–[11]. They have the same L1C/L2C structure that inspired Jenga, and they all perform parity across multiple blocks. We summarize the key characteristics of this related work and Jenga in Table 2, and we now discuss some relevant differences in their details.

Efficient RAS (E-RAS) [9] distributes the parity of the L1C and L2C so that data and parity are not collocated. Additionally, the L2C is spread across all the different channels to increase reliability.

Citadel [11] and RATT-ECC [8] use an additional dedicated ECC channel to store the L1C and L2C parity. That ECC channel can become a bottleneck as it needs to be accessed for every memory read and write.

Although Citadel, RATT-ECC, and E-RAS achieve high fault tolerance with relatively high host-visible capacity, they do not provide tolerance for die or channel failures. Additionally, all three of these works assume the existence of a unified memory controller that is aware of all the HBM memory capacity and thus can easily compute, store and load the necessary parity bits. However, this assumption is unrealistic as each HBM channel has its own unique controller in order to maximize the overall bandwidth.

Like Jenga, Parity Helix [10] uses the ECC that is collocated with the data as the L1C, and the L2C is distributed

Table 2. Comparing Jenga to Prior Work

Issue	**E-RAS** [9]	**Citadel** [11]	**RATT-ECC** [8]	**Parity Helix** [10]	**Jenga**
To read block	2 reads on the same channel (sequential)	2 reads to different channels (1 of which is always ECC channel)	2 reads to different channels (1 of which is always ECC channel)	1 read to a single channel	2 half-block reads to different channels
To write block	2 reads,3 writes (2 channels)	4 reads, 5 writes (2 channels, 1 of which is ECC channel)	2 reads, 3 writes (2 channels, 1 of which is ECC channel)	2 reads, 3 writes (2 channels)	0 reads, 3 half-block writes (3 channels)
To correct error	1 read from every channel	multiple reads from all channels	multiple reads from all channels	1 read from every channel	1 half-block read
Level-1 code	detection with CRC-8	detection with CRC-32	detection and some correction with Reed-Solomon	detection and some correction with Hamming(72,64)	detection and some correction with Hamming(72,64)
Level-2 code	1-dimensional XOR across channels	3-dimensional XOR across rows, banks and channels	2-dimensional XOR across banks and channels	1-dimensional XOR across channels	1-demensional intra-block XOR.
Hardware modifications	none	assumes added ECC die	assumes added ECC die	none	none
Normalized host-visible capacity	0.84	0.86*	0.875*	0.875**	0.66
Needs centralized HBM controller?	yes	yes	yes	no	no

***0.8 assuming ECC-die instead of ECC-channel** ****0.75 assuming 2 Channels per Die instead of 1.**

in a modulo fashion across different channels. Parity Helix's use of spatial redundancy allows it to tolerate die and channel failures.

All of these prior schemes share a common challenge. When data blocks in the same parity group try to update the parity at the same time, they must be careful to avoid consistency violations (especially if the parity is cached). This problem also can occur during recovery, as synchronization is necessary to guarantee consistency when different cores are trying to read and write the same parity block. Modifications to the coherence protocol may be able to resolve this problem by locking accesses to parity blocks. Jenga avoids this problem because its L2C redundancy is at the size of a single block: the same size granularity as the coherence protocol.

VIII. Conclusion

Our goal in this work was to develop a comprehensive fault tolerance scheme for 3D DRAM that would minimize the performance costs. Jenga achieves this goal by using a combination of a collocated SECDED code and a sub-block parity scheme to tolerate faults in bits, rows, banks, channels, and chips.

Acknowledgments

This material is based on work supported by the National Science Foundation under grant CCF-142-1177.

References

[1] "Radeon™ R9 Series Graphics Cards | AMD." http://www.amd.com/en-us/products/graphics/desktop/r9#. [18-May-2017].

[2] "Multi-Channel DRAM (MCDRAM) and High-Bandwidth Memory (HBM) | Intel® Software." https://software.intel.com/en-us/articles/multi-channel-dram-mcdram-and-high-bandwidth-memory-hbm. [18-May-2017].

[3] "Tesla P100 Most Advanced Data Center Accelerator | NVIDIA." http://www.nvidia.com/object/tesla-p100.html. [18-May-2017].

[4] B. Black *et al.*, "Die Stacking (3D) Microarchitecture," in *Proc. of the 39th Annual International Symposium on Microarchitecture*, Washington, DC, USA, 2006.

[5] G. H. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," in *2008 International Symposium on Computer Architecture*, 2008, pp. 453–464.

[6] G. H. Loh, Y. Xie, and B. Black, "Processor Design in 3D Die-Stacking Technologies," *IEEE Micro*, vol. 27, no. 3, pp. 31–48, May 2007.

[7] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 239–249.

[8] H.-M. Chen, C.-J. Wu, T. Mudge, and C. Chakrabarti, "RATT-ECC: Rate Adaptive Two-Tiered Error Correction Codes for Reliable 3D Die-Stacked Memory," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, p. 24:1–24:24, Sep. 2016.

[9] H. Jeon, G. H. Loh, and M. Annavaram, "Efficient RAS support for die-stacked DRAM," in *2014 International Test Conference*, 2014, pp. 1–10.

[10] X. Jian, V. Sridharan, and R. Kumar, "Parity Helix: Efficient protection for single-dimensional faults in multi-dimensional memory systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 555–567.

[11] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "Citadel: Efficiently Protecting Stacked Memory from Large Granularity Failures," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[12] C. L. Rumer and E. A. Zarbock, "Through silicon via, folded flex microelectronic package," US6924551 B2, 02-Aug-2005.

[13] B. Schroeder and G. A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2006, pp. 249–258.

[14] J. Ziegler and others, "IBM Experiments in Soft Fails in Computer Electronics," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 3–18, Jan. 1996.

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

[16] "JEDEC Updates Groundbreaking High Bandwidth Memory (HBM) Standard | JEDEC." https://www.jedec.org/news/pressreleases/jedec-updates-groundbreaking-high-bandwidth-memory-hbm-standard. [Accessed: 19-May-2017].

[17] C. C. Huang and V. Nagarajan, "Increasing cache capacity via critical-words-only cache," in *IEEE 32nd International Conference on Computer Design (ICCD)*, 2014.

[18] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy," in *Proc. 45th Annual Int'l Symp. on Microarchitecture*, 2012.

[19] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines," in *IEEE 13th Int'l Symposium on High Performance Computer Architecture*, 2007..

[20] 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*, Washington, DC, USA, 2012, pp. 548–559.

[21] D. Locklear, *Chipkill Correct Memory Architecture*. 2000.

[22] N. Binkert *et al.*, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[24] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, "Achieving Predictable Performance Through Better Memory Controller Placement in Many-core CMPs," in *Proc. 36th Annual Int'l Symposium on Computer Architecture*, 2009.

[25] R. Hesse, J. Nicholls, and N. E. Jerger, "Fine-Grained Bandwidth Adaptivity in Networks-on-Chip Using Bidirectional Channels," in *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, 2012.

[26] P. V. Gratz and S. W. Keckler, "Realistic Workload Characterization and Analysis for Networks-on-Chip Design," in *4th Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2010.