ECC Parity: A Technique for Efficient Memory Error Resilience for Multi-Channel Memory Systems

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

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

***Abstract*—**

Servers and HPC systems often use a strong memory error correction code, or ECC, to meet their reliability and availability requirements. However, these ECCs often require significant capacity and/or power overheads. We observe that since memory channels are independent from one another, error correction typically needs to be performed for one channel at a time. Based on this observation, we show that instead of always storing in memory the actual ECC correction bits as do existing systems, it is sufficient to store the bitwise parity of the ECC correction bits of different channels for fault-free memory regions, and store the actual ECC correction bits only for faulty memory regions. By trading off the resultant ECC capacity overhead reduction for improved memory energy efficiency, the proposed technique reduces memory energy per instruction by 54.4% and 20.6%, respectively, compared to a commercial chipkill correct ECC and a DIMM-kill correct ECC, while incurring similar or lower capacity overheads.

I. Introduction

Error resilience in memory is an indispensable feature for servers and HPC systems. Without adequate error resilience, an error in memory can lead to expensive downtime and corruption of application data. As the number of DRAMs and DIMMs per system increases, faults in memory systems are becoming increasingly common. As such, error resilience in memory has been widely adopted [11], [5], [17]. However, memory error resilience often incurs high power and/or capacity overheads [4], [24], [22].

In conventional memory error resilience designs, the memory system stores and manages an independent ECC for each memory channel[1]. For systems with a few memory channels, this simple design may be the best option. However, the number of channels per server has been increasing to keep up with the growing memory bandwidth requirement due to the increase in the number of cores per processor. Some high-performance processors, such as Intel Xeon 6500 and Xeon 7500, contain four and eight physical memory channels, respectively [3], [22]. In this paper, we argue that independently storing ECC resources for each channel may not be a capacity or energy efficient option for such systems.

We observe that because memory channels are independent from one another, faults typically occur in only one channel at a time. Therefore, error correction typically needs to be performed for only one channel at a time as well. As such, for systems with many channels, we propose storing in memory only the bitwise parity of the ECC correction bits (i.e., the bits in an ECC that are used for correcting errors, as opposed to the bits in an ECC that are used for detecting errors) of different channels. We call this bitwise parity of the ECC correction bits an *ECC parity*. When needed, the actual ECC correction bits of a line in a faulty channel can be obtained by XORing the line's ECC parity with the ECC correction bits of appropriate lines in the remaining healthy channels; the ECC correction bits of lines in the healthy channels can be directly computed from the lines. Only after a fault occurs in a channel do we store the actual ECC correction bits of its faulty region(s) in memory; this protects against the accumulation of faults across multiple channels over time. In comparison, current systems always store in memory the ECC correction bits of every channel.

Storing the ECC parity, instead of the actual ECC correction bits themselves, can significantly reduce the ECC capacity overhead for systems with multiple channels. The resultant ECC capacity overhead reduction can be traded off for improved memory energy efficiency. When applied to chipkill correct, the proposed optimization, ECC Parity (with uppercase 'P', as opposed to the lower case 'p' in ECC parity) reduces memory energy per instruction by 53% and 56%, respectively, compared to a quad-channel and a dual-channel memory system protected by a commercial chipkill correct ECC [5]. When applied to a quad-channel and a dual-channel memory system protected by a commercial DIMM-kill correct ECC [17], ECC Parity reduces memory energy per instruction by 21% and 18%, respectively.

The rest of the paper is organized as follows. Section II provides the relevant background and motivations. Section III describes ECC Parity. Section IV describes the evaluation methodology. Section V presents the experimental results. Section VI discusses system-level impacts. Section VII surveys related work. Section VIII concludes the paper.

II. Background and Motivation

Large-scale field studies report that DRAMs are susceptible to device-level faults that cause a significant fraction of an entire memory device to become faulty [20], [21]. Correspondingly, chipkill correct ECC, which is capable of tolerating device-level faults in memory, is provided by virtually all servers on the market [11] and is widely adopted by supercomputers and data centers [19], [20], [16], [12]. Chipkill

[1]Unless stated otherwise, we refer to a memory channel as a logical memory channel. A logical memory channel consists of one or more physical memory channels.

SC14, November 16-21, 2014, New Orleans, Louisiana, USA
978-1-4799-5500-8/14/$31.00

correct can tolerate up to a complete chip failure in a *rank* [9], which is a group of DRAM chips in a channel that are always accessed in parallel to serve each memory request. Stronger error correction techniques that guard against complete DIMM failure(s) per channel, or DIMM-kill correct, are also found in commercial systems [17].

Providing memory error resilience often incurs high overheads, however. For example, RAIM, a commercial DIMM-kill correct ECC, stripes each memory line and its ECC across five DIMMs; as such, each rank consists of a large number (i.e., 45) of DRAM chips [17]. Accessing such a large number of memory chips for every memory request incurs a high memory energy overhead. In addition, 13 out of the 45 chips per rank are ECC chips, which lead to a high $13/32 = 40.6\%$ capacity overhead. As another example, a common commercial chipkill correct implementation [5], which we refer to as the 36-device commercial chipkill correct in the rest of the paper, stripes each line and its ECC across 36 DRAM chips, and thus also suffer from a high memory power overhead [4], [24]. Although one can choose to stripe each line across fewer chips, doing so increases the fraction of data in a line that can be affected when a chip fails and thus increases the amount of ECC bits per line, or capacity overhead, needed to support chipkill correct. For example, LOT-ECC [22], a recent chipkill correct proposal, stripes each line and its ECC bits across as few as five chips to reduce the power overhead of chipkill correct; however, it increases the capacity overhead from the 12.5% overhead found in commercial chipkill correct to a 40.6%[2] overhead. The goal of our work is to explore memory architecture level optimizations to reduce the overheads of providing memory error resilience in the context of multi-channel memory systems.

It is well known that a memory ECC can often be decomposed into ECC detection bits and ECC correction bits. For example, 36-device commercial chipkill correct use four check symbols per memory word [4], [24]; only two check symbols are required for error detection [24], while the remaining two are needed for correcting detected errors. Figure 1 shows the breakdown of the capacity overheads of commercial chipkill correct, commercial DIMM-kill correct, and two different LOT-ECC implementations into ECC detection bits and ECC correction bits; LOT-ECC I and LOT-ECC II in the figure stand for the nine-chip per rank and five-chip per rank implementations of LOT-ECC, respectively. Typically 50% or more of the ECC capacity overhead comes from the ECC correction bits. This is a very high fraction considering that the ECC correction bits are only needed to correct errors after memory faults occur - typically a rare event. Our goal is to reduce the overhead of memory error resilience by reducing the capacity overhead of these rarely used ECC correction bits.

In the conventional memory system design, the memory system stores the actual ECC correction bits of every channel in memory. This approach can correct errors due to faults occurring simultaneously in all channels. However, the mean time between faults occurring in different channels is in the order of 100's of days, as shown in Figure 2. The mean time between faults in different channels is high because memory

[2]Under this implementation, every four 72B (64B data + 8B ECC bits) memory lines holding data are protected by one 72B memory line holding only ECC bits. Therefore, the capacity overhead is $(8 \cdot 4 + 72)/(64 \cdot 4) = 40.6\%$.

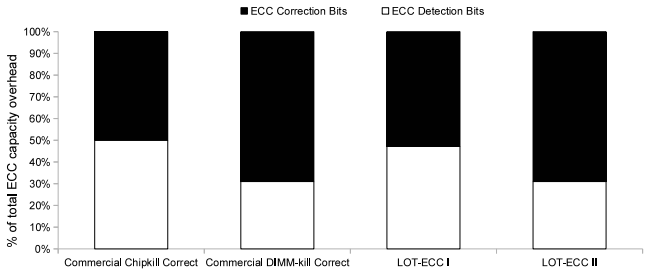

Fig. 1. The breakdown of the capacity overheads of different memory ECCs. A significant fraction of the capacity overhead is due to ECC correction bits.

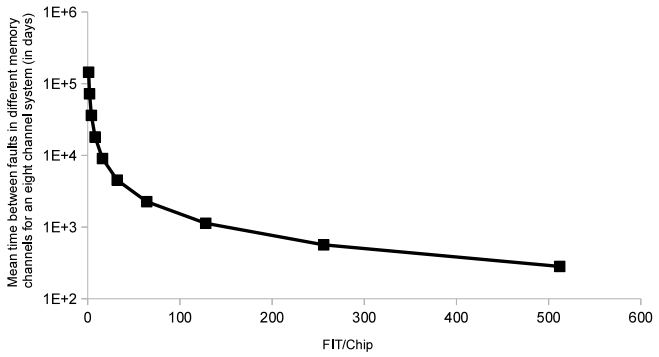

Fig. 2. Mean time between faults in different channels versus DRAM fault rates for an eight-channel system with four ranks per channel and nine chips per rank, assuming the exponential failure distribution. Sridharan et al. [21] report an average DDR3 fault rate of 44 FIT/chip across different vendors.

channels are independent from one another so they also fail independently from one another. For example, a DRAM failure or a DIMM failure in one channel does not affect the DRAMs or DIMMs in another channel because they do not share any common circuitry. Therefore, instead of always storing in memory the actual ECC correction bits of every channel, it may be sufficient to store a reduced form of the ECC correction bits that only provides full error correction coverage for one faulty channel at a time. Because often half or more of the capacity overhead of memory ECCs are due to their ECC correction bits (see Figure 1), storing only a reduced form of the ECC correction bits can result in significant capacity overhead reduction. The resultant savings can either simply remain as capacity overhead reduction or be traded off for improved memory energy efficiency.

III. ECC PARITY

We propose a memory architectural level optimization for multi-channel systems that can be applied on top of diverse memory ECCs (e.g., chipkill correct, double chipkill correct, DIMM-kill correct, etc.) to reduce the overheads of the underlying memory ECC. The optimization is to store in memory the bitwise parity of the ECC correction bits instead of always storing in memory the actual ECC correction bits as do existing memory systems. ECC detection bits continue to be stored in memory for every channel; this allows error detection to be performed on the fly with each read access, which lies on the critical path of execution. When errors are detected in a channel, the ECC correction bits of the data lines in the channel can be reconstructed from ECC parities and data lines

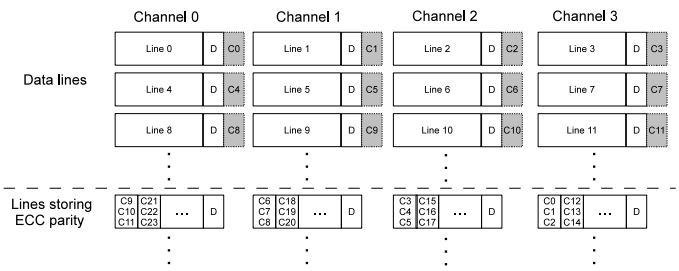

Fig. 3. Example ECC parities. 'D' stands for the eight detection check symbols and 'C' stands for the eight correction check symbols per line. 'C0C1C2' stands for $C0 \oplus C1 \oplus C2$. Shaded boxes represent values that are only calculated for the data lines but are not actually stored in memory.

in other channels as long as no two memory channels fail at the same relative location. To guard against the event that a second memory channel may later fail at the same location as where another channel had failed earlier, we calculate and store in memory the actual ECC correction bits of each faulty memory region.

The rest of the section is organized as follows. Section III-A describes the construction and layout of ECC parities. Section III-B describes the layout of the ECC correction bits after faults occur in memory. Section III-C describes added steps to memory requests. Section III-D describes optimizations to mitigate the performance impact of the steps added to regular memory accesses. Section III-E discusses the various overheads.

A. ECC Parity Construction and Layout

We use a two-stage encoding procedure to obtain the final redundancy bits for correcting detected errors. In the first stage we compute an ECC specified by the system designer (e.g., one that provides chipkill correct, double chipkill correct, DIMM-kill correct, etc.) for a data line. In the second stage, we compute a bitwise parity from the ECC correction bits of lines in different channels that are obtained in stage one. We refer to the end result as the ECC parity and store it in memory.

Specifically, to obtain an ECC parity in a memory system with N channels, one first computes the ECC correction bits of a memory line in each of $N-1$ channels and evaluates the bitwise XOR of these $N-1$ values. The result, which is the ECC parity, is then stored in the N^{th} channel. When a fault occurs in a channel, the ECC correction bits of an affected line in the faulty channel can be reconstructed by XORing the line's ECC parity with the ECC correction bits of the appropriate lines in the remaining channels. The ECC correction bits of these latter lines can be directly computed if they are error-free. The ECC parities guarantee the same error correction coverage as provided by the underlying ECC correction bits for faults within a single channel. However, when lines in the same relative location in two or more channels contain errors (e.g., due to the accumulation of faults across multiple channels over time), one cannot reconstruct the ECC correction bits of any of these erroneous lines from their ECC parity. Therefore, we only store ECC parities for fault-free memory regions; for memory regions with fault(s), we store their ECC correction bits in memory, as will be described in Section III-B.

As an example, we demonstrate the construction of ECC parities using the 36-device commercial chipkill correct as

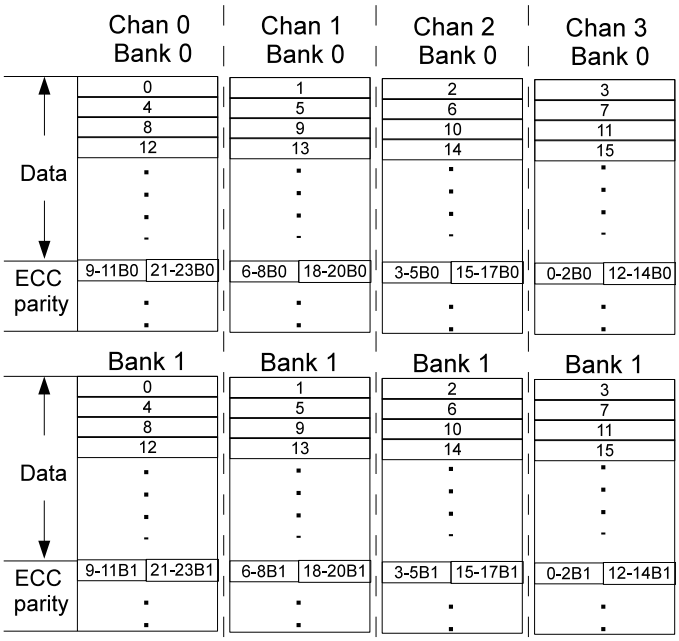

Fig. 4. ECC Parity layout example. Row 0 represents a 4KB page. '0-2B0' stands for all the ECC parities of lines in Rows 0, 1, and 2 of Bank 0.

the underlying ECC. 36-device commercial chipkill correct stores a four-check-symbol code per word in memory; two of these check symbols are needed for error detection [24], while the remaining two are used for correcting detected errors. These correction check symbols are used to construct ECC parities. Let us assume that there are four words per line, and therefore, eight detection check symbols and eight correction check symbols per line. Figure 3 shows the data lines of a quad-channel memory system, their detection check symbols, as well as their ECC parities that replace the correction check symbols, which are represented by the shaded boxes.

We store the ECC parities in memory regions separate from the data lines to avoid complicating the address translation for application read requests to memory, which often lie on the critical path of execution. Figure 4 shows the layout of the ECC parities for an example quad-channel memory system in which the size of the ECC correction bits of each data line is half the size of the data line. Each numbered row in the figure represents a typical 4KB row that stores a 4KB physical page. For simplicity, the figure only shows two banks per channel. We reserve the last rows in each memory bank to store the ECC parities. We evenly distribute the ECC parities corresponding to the same banks of different channels across these same banks. Each row of ECC parities protects $(N-1)/R$ rows of data, where R is the ratio of the size of the ECC correction bits per data line to the size of the data line. In the example in Figure 4, $R = 0.5$; therefore, a row of ECC parities corresponds to $(4-1)/0.5 = 6$ rows of data.

B. ECC Correction Bits Layout

While the ECC parities protect against faults in a single channel, they do not protect against faults that have developed in the same locations in different channels (e.g., due to the accumulation of faults in memory over time). Compared to storing the actual ECC correction bits, storing the ECC parities

also incurs significant performance overhead when errors need to be corrected because error correction using an ECC parity requires the additional step of reading out data lines from multiple channels to reconstruct the ECC correction bits. To overcome these problems, we store in memory the ECC correction bits of a faulty memory region after the fault occurs.

To store ECC parities for healthy memory regions and ECC correction bits for faulty memory regions, there must to be a way to record the type of error correction resources that is currently stored for different data lines. A fine-grained mechanism that tracks this information on a line-by-line basis will incur prohibitive overheads. Instead, we track the type of stored ECC resources at the much coarser granularity of pairs of memory banks in the same channel. When the combined number of errors encountered in a pair of banks in the same channel exceeds a certain threshold, the actual ECC correction bits for both banks are computed and then stored in memory. To store the ECC correction bits for this pair of banks, each bank in the pair stores the ECC correction bits corresponding to data in the other bank, as shown in Figure 5; this layout minimizes the latency of error correction by allowing the memory request to a data line and the request to its corresponding ECC correction bits to partially overlap. Meanwhile, all the ECC parity lines (i.e., memory lines that store ECC parities) that are used to protect these two banks have to be recalculated to remove the content of the two banks from their construction. While recalculating the ECC parity lines is expensive, it is performed rarely because the mean time between DRAM faults in different channels is high (e.g., once every hundreds of days, see Figure 2). Therefore, the impact on performance due to this operation is negligible (e.g., a few seconds, which is the time it takes to read out the entire content of a typical memory system, of degraded memory performance per hundreds of days).

Finally, to maintain the same protection level as the ECC

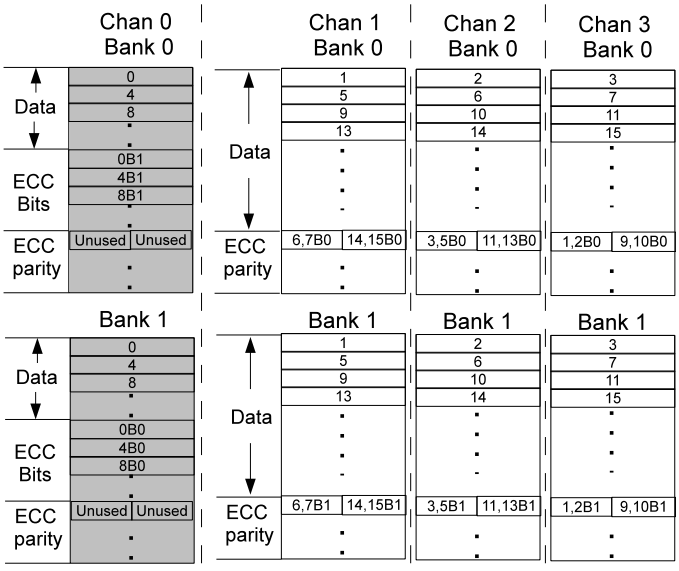

Fig. 5. Example illustrating the layout of both ECC parities and ECC correction bits in a memory system with fault. The shaded regions represent faulty banks. '0B1' represents the ECC correction bits for Row 0 of Bank 1. '1,2B0' represents the ECC parities for lines in Rows 1 and 2 of Bank 0.

parities, the ECC correction bits themselves require ECC protection because a fault in a channel can also affect lines storing the ECC correction bits; this increases the effective amount of the ECC correction bits. This is illustrated in Figure 5, where each row of data in a faulty bank (e.g., Row 0 of Bank 0) is protected by a full row of ECC correction bits (e.g., '0B0'), whereas each row of data in a healthy bank (e.g., Row 1 of Bank 0) is protected by only half a row of ECC parities (e.g., '1,2B0'). The exact extent that the amount of ECC correction bits has to be increased depends on the underlying ECC. In general, if the underlying ECC incurs a capacity overhead of x%, the effective amount of the ECC correction bits has to be increased by x% as well when one uses the same ECC to protect the ECC correction bits. For simplicity, ECC Parity allocates twice as many bits to store the ECC correction bits of a data line as the number of bits used to store the ECC parity of a data line.

C. Added Steps In Memory Operations

The memory system needs to support periodic scanning for memory errors (e.g., via a memory scrubber) to ensure a high probability that the ECC correction bits of a faulty memory region can be computed and stored before another fault occurs in the same relative location in a different channel. An detected error always increments the error counter corresponding to the memory bank with the error. Before an error reaches a threshold (e.g., set to a small number such as four to differentiate bit and row faults from larger faults like bank faults), the OS simply retires the physical page that contains the detected error, as well as all physical pages that share the same ECC parities as the faulty physical page; retiring the physical page prevents permanent bit faults and permanent row faults from repeatedly incrementing, and therefore, saturating the error counter. When the error counter saturates, implying a potential large device-level fault such as a bank fault, the corresponding bank pair is recorded as faulty. The actual ECC correction bits of this pair of banks are then to be calculated and stored in memory.

The use of ECC parities also requires some modifications to regular memory accesses. Figure 6 shows memory read and write operations starting from the last level cache (LLC) of the processor. In parallel to every application read request to memory, the LLC controller performs a bank health lookup (Step A1 in Figure 6) to check whether the bank containing the requested line is currently recorded as faulty. If the bank is recorded as faulty, the corresponding ECC line (i.e., a line that only holds ECC correction bits) in memory have to be read as well (Step B in Figure 6). If errors are detected in a line arriving from memory and the bank is not recorded as faulty, the ECC correction bits of the requested line have to be first reconstructed (via Step C in Figure 6) from the line's ECC parities before error correction can proceed. We expect the added bank health lookup step (Step A1) to have small impact on performance because this information is kept on chip in a small and, therefore, fast SRAM table. On the other hand, individual error correction operations using ECC parities are expensive because they require reading out all data lines that share the same ECC parity; this requires a total of N-1 additional memory accesses. However, we expect the overall impact of using ECC parity for error correction (Step C) on performance to be small because we record which banks are

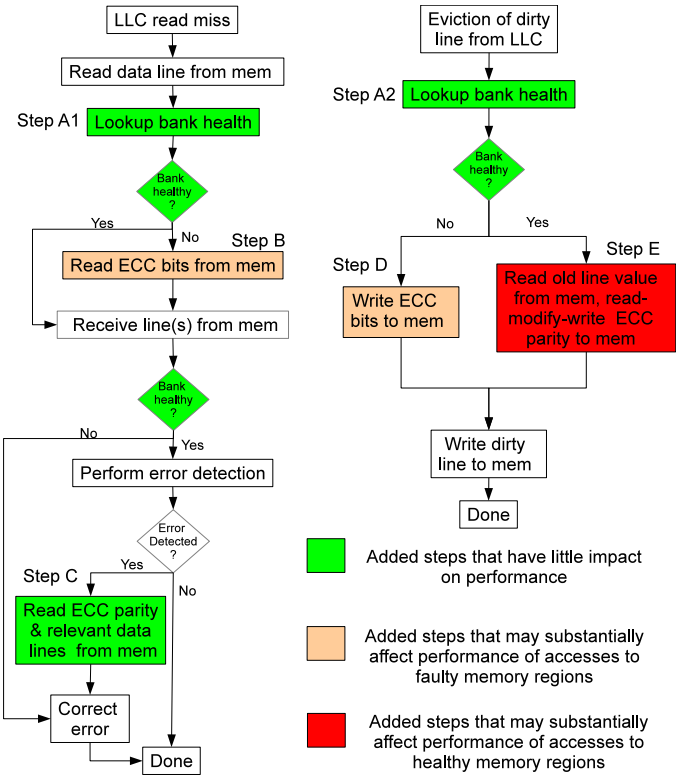

Fig. 6. Steps added to memory accesses to support ECC parities.

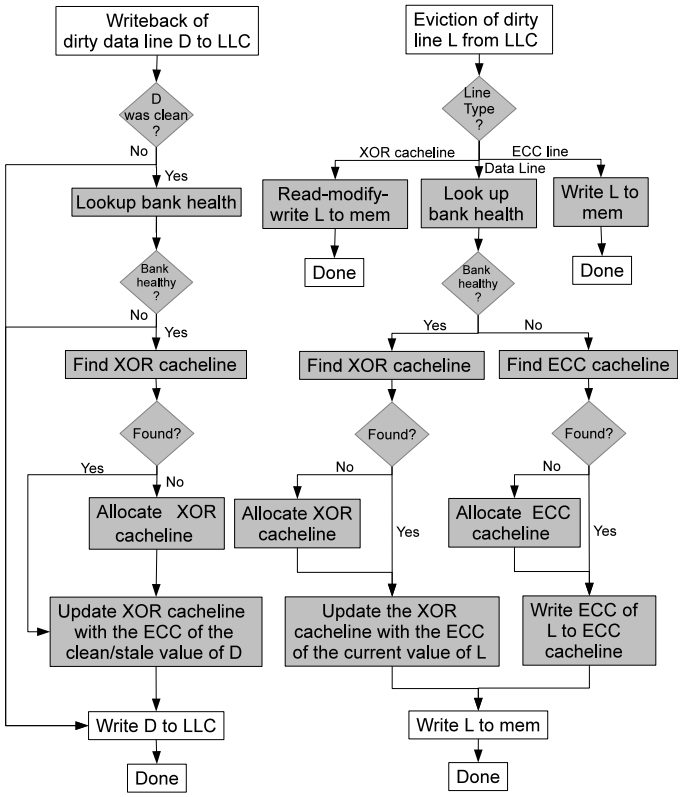

Fig. 7. Modifications to the LLC to mitigate the overheads of the added steps to memory accesses in Figure 6. Added steps are highlighted in gray.

faulty and then store in memory the actual ECC correction bits, not ECC parities, for these banks. Overall, we expect the steady state behavior of reading the ECC line for every application read request to a faulty bank (Step B) to be the most expensive step among the added steps above.

Meanwhile, for each memory write request, ECC Parity looks up whether the bank written to is recorded as faulty (Step A2 in Figure 6) to properly update the dirty line's error correction resources (as ECC parities or ECC correction bits). If the bank is recorded as faulty, ECC Parity simply calculates the ECC correction bits of the written line and writes it to the ECC line in memory (Step D). However, if the bank is not recorded as faulty, the corresponding ECC parity needs to be updated (Step E) using the following equation:

$$ECCP_{new} = ECCP_{old} \oplus ECC_{old} \oplus ECC_{new} \quad (1)$$

In equation 1, ECC_{new} stands for the ECC correction bits computed using the current value of the dirty line; ECC_{old} stands for the ECC correction bits of the old value of the dirty line that is currently stored in memory before the writeback to memory. $ECCP_{new}$ is the new ECC parity to be written to memory; $ECCP_{old}$ is the old value of the ECC parity that is currently stored in memory. Again, we expect bank health lookups (Step A2) to have small impact on performance. Updating the ECC correction bits (Step D) requires an additional write access to memory for every application write request to a faulty bank, and thus may have substantial impact on memory bandwidth. Finally, updating the ECC parity (Step E) requires three additional memory accesses - reading the old value of the dirty line from memory, reading the ECC parity line, and then writing the updated ECC parity line back to memory. This may incur significant bandwidth overhead even for a healthy memory system.

D. Hardware Optimizations

This section describes hardware optimizations borrowed from prior works [24], [13] to mitigate the potential performance impact of steps B, D, and E in Figure 6. The optimizations require modifications to the LLC controller.

To reduce the overheads of Steps B and D, we propose using the LLC to cache ECC lines similar to VECC [24]; this allows future application read requests to faulty data lines protected by the same ECC line to access the ECC line from the LLC instead of from memory. Similarly, caching the ECC line also helps to reduce the overhead of updating the ECC lines for application write requests to faulty banks.

To mitigate the performance impact of Step E, we borrow a caching technique from [13]; the technique eliminates the read access to the old value of the dirty line in memory and reduces the number of read-modify-write operations to the ECC parity lines in memory. Functionally, the borrowed technique compacts into a single LLC cacheline the old and new ECC correction bits of all dirty data lines that are protected by the same ECC parity line. It does so by storing the bitwise XOR of these ECC correction bits, instead of the ECC correction bits themselves, as only the XOR values are needed to update the ECC parity (see Equation 1). We call a cacheline that currently stores such an XOR value a XOR cacheline. For managing the XOR cachelines in the LLC, each XOR cacheline takes on the same physical address as the corresponding ECC parity line in memory.

Figure 7 summarizes the added steps to the operations of the LLC to implement all of the above optimizations, assuming an inclusive cache hierarchy.

E. Overheads

ECC Parity incurs four kinds of memory capacity overheads: one for the ECC detection bits, one for the ECC parity lines, one for the retired pages due to errors encountered before the error counter of a bank pair reaches its threshold, and one for the ECC lines. We use the dedicated ECC chips per DIMM to store the ECC detection bits. Since the dedicated ECC chips per DIMM typically incur a 12.5% capacity overhead, the proposal also incurs a 12.5% capacity overhead for error detection. Meanwhile, the ECC parity lines take up a constant capacity overhead of $(1+12.5\%)\cdot R/(N-1)$, where N is the number of channels that share ECC parities and R is the ratio of the size of the ECC correction bits of a data line to the size of the data line; the 12.5% term in the formula accounts for the capacity overhead incurred by the error detection bits for the ECC parity lines. Unlike the first two capacity overheads that are static, the next two capacity overheads increase over time as faults accumulate in memory. By setting the error counter threshold for each bank pair to four, the maximum number of retired pages before the error counter saturates is $4\cdot(N-1)$ pages, which is a negligible fraction out of the 100,000's of pages in a pair of memory banks. Next, we use Monte Carlo simulation to estimate what fraction of memory per memory system end up having their ECC correction bits stored in memory after seven years of operation. The Monte Carlo simulation models memory systems with different numbers of channels, where each channel contains four ranks with nine chips per rank; the simulations assume the average of DDR3 DRAM fault distributions of different DRAM vendors reported in [21]. Figure 8 presents the results; it shows that only a small fraction (i.e., 0.4%, on average) of memory per system end up having their ECC correction bits stored in memory.

In addition to memory capacity overheads, ECC Parity also incurs some area overhead on the processor. Each pair of memory banks requires an error counter. By using 0.5B of on-chip storage per pair of banks, the error counters require 512B of on-chip storage to support a large (e.g., 512GB) memory system with 1024 memory banks.

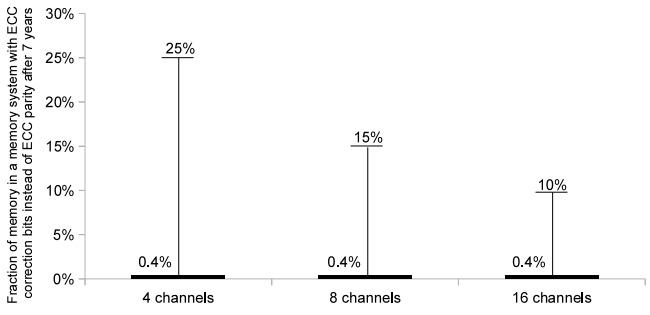

Fig. 8. The solid bars represent the average fraction of memory per system that ends up having their ECC correction bits stored in memory after seven years of operation. The horizontal lines indicate the upper limit of the fraction of memory protected by ECC correction bits for 99.9% of the simulated systems.

IV. Methodology

A. Baselines

To demonstrate that ECC Parity is a general optimization that can be applied to diverse memory ECCs, we evaluate it in the context of both DIMM-kill correct and chipkill correct. For DIMM-kill correct, we apply ECC Parity to RAIM [17] by storing in memory the parity of the ECC correction bits of RAIM instead of the ECC correction bits themselves. For chipkill correct, we apply ECC Parity to the most energy-efficient LOT-ECC implementation, which we call *LOT-ECC5*, that uses only five chips per rank; under this LOT-ECC implementation, four out of the five chips per rank are X16 DRAMs and the fifth is a X8 DRAM with half the capacity and I/O width as one of the X16 DRAMs. By itself, LOT-ECC5 incurs impractically high capacity overhead (i.e., 40.6%, see Section II). By applying ECC parity to LOT-ECC5, we enable the use of LOT-ECC5 by reducing its capacity overhead down to practical levels (e.g., from 40.6% down to 16.5% in a system with eight channels), while preserving its energy efficiency. The end result is that one achieves similar energy efficiency as LOT-ECC5, while incurring similar capacity overhead as the more power hungry, but low capacity overhead 36-device commercial chipkill correct. To quantify the above, we evaluate LOT-ECC5 and 36-device commercial chipkill correct and compare them against LOT-ECC5+ECC Parity. For additional comparisons, we evaluate a second LOT-ECC implementation, which we call *LOT-ECC9*, that uses nine chips per rank and another recently proposed chipkill correct ECC - Multi-ECC [13]. Finally, we also compare against a recent commercial chipkill correct implementation, which we refer to as 18-device commercial chipkill correct, that uses only two instead of the usual four check symbols per word [6]; compared to 36-device commercial chipkill correct, it requires accessing only 18 instead of 36 chips per memory request, but potentially slightly impacts error detection coverage.

B. Experimental Setup

We use GEM5 [7] to model a 2GHz processor with eight cores. Table I lists the core and cache parameters. We simulate 12 eight-core multiprogrammed SPEC workloads and four eight-core multi-threaded PARSEC workloads. All selected workloads consume at least 1% of the total bandwidth of the evaluated memory systems. Each multiprogrammed SPEC workload is generated using eight instances of the same benchmark. We fast-forward one of the instances in each workload one billion instructions before the SimPoint [1] instruction. We fast-forward the remaining seven instances such that all eight instances are separated from one another by 10 million instructions. After fast-forwarding, we also warm up the cache until every instance has executed for the next one billion instructions. Finally, we simulate each workload in timing mode for 10 million cycles and collect statistics over this period. For PARSEC workloads, we first fast-forward to the beginning of the region of interest, then warm up the cache until one of the threads has executed for the next one billion instructions, and finally simulate in timing mode for the next 10 million cycles. Figure 9 characterizes the bandwidth utilization of the evaluated workloads assuming a dual-channel commercial chipkill correct memory system.

TABLE I. PROCESSOR MICROARCHITECTURE

Issue width 2	Type OO	LSQ Size 32LQ/32SQ	ROB Size 64
L1 Line Size 64B	L1 D$, I$ 32 KB	L1 Assoc. 2 ways	L1 Latency 2 cycles
L2 size 8MB	L2 Assoc. 16 ways	L2 Latency 10 cycles	L2 miss/write buffer size 512/128

We use DRAMsim [2] to model memory power and performance. We model 2Gb DDR3 DRAM chips with 1GHz I/O frequency; their parameters are taken from die revision D in [18]. We use the Most_Pending memory access scheduling policy from DRAMsim. Similar to [22], we use the close-page row buffer policy, which allows a rank to be placed in sleep mode when idle to reduce background power consumption. For the device address mapping policy, we use the High_Performance_Map from DRAMsim as the intra-channel mapping policy and interleave adjacent physical pages across different memory channels to balance the bandwidth utilization across these channels. Similar to prior works [24], [22], we configure all memory systems protected by the same type of ECC (chipkill correct or DIMM-kill) with the same total physical memory capacity and I/O width to perform comparisons across different ECC implementations. In addition, we note that since ECC Parity is an optimization that exploits having multiple channels in a memory system, its benefits strongly depend on the number of channels in the memory system. Therefore, we evaluate two memory system sizes, one with a few channels where the expected benefits are small and one with more channels where the expected benefits are high. For the former, we evaluate memory systems that are equivalent in physical bandwidth and size to a dual-channel commercial ECC memory system (i.e., a memory system protected by 36-device commercial chipkill correct or by RAIM); for the latter, we evaluate memory systems that are equivalent in physical bandwidth and size to a quad-channel commercial ECC memory system. Table II summarizes the evaluated memory system configurations. Note from Table II that the line size of 36-device commercial chipkill correct and RAIM is 128B, which is twice as large as the 64B line size under other memory error resilience schemes (i.e., 64B). The former two schemes have a larger line size because they have a large number of (i.e., 32) data chips per rank, which in aggregate supply 128B of data per memory request. To maintain the same number of memory I/O pins in the different systems given the different line sizes, we let the number of logical (not physical) channels of 36-device commercial chipkill correct and RAIM be smaller than that of other techniques (see Table II).

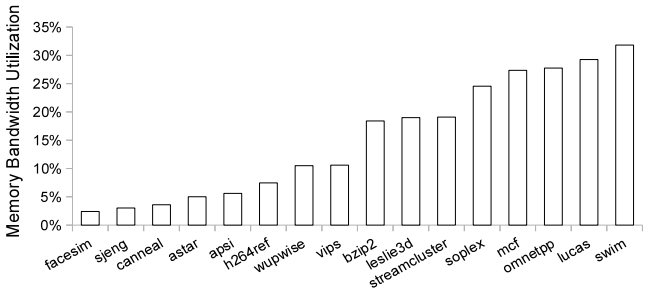

Fig. 9. Workload memory bandwidth utilization in a dual-channel commercial ECC memory system.

TABLE II. SUMMARY OF EVALUATED ECC IMPLEMENTATIONS.

	Rank Config.	Line Size	Ranks/ Chan	Logical Channels	Total I/O Pin Count
36-device Commercial Chipkill Correct	36 X4	128B	1	2, 4	288, 576
18-device Commercial Chipkill Correct	18 X4	64B	1	4, 8	288, 576
LOT-ECC5	4 X16, 1 X8	64B	4	4, 8	288, 576
LOT-ECC9	9 X8	64B	2	4, 8	288, 576
Multi-ECC	9 X8	64B	2	4, 8	288, 576
LOT-ECC5 + ECC Parity	4 X16, 1 X8	64B	4	4, 8	288, 576
RAIM	45 X4	128B	1	2, 4	360, 720
RAIM + ECC Parity	18 X4	64B	1	5, 10	360, 720

C. Modeling Details of ECC Caching

LOT-ECC, Multi-ECC, and resilience schemes with ECC Parity all require additional memory accesses to update the corresponding ECC-related lines in memory for application write requests. We model caching of the ECC parities in the LLC for the schemes with ECC parities. We also model caching the ECC correction bits of LOT-ECC and Multi-ECC in the LLC for fair comparison. In our evaluation, we treat the ECC-related cachelines the same way as data cachelines in terms of LLC insertion and replacement policies and begin caching them at the start of the warm-up period. Our modeling of ECC caching for Multi-ECC is identical to [13] with the exception that we cache the ECC correction bits in the 8MB LLC instead of a much smaller but dedicated 128KB ECC cache. To model ECC caching for LOT-ECC, we let each ECC cacheline cover four and eight logically adjacent data lines in LOT-ECC5 and LOT-ECC9, respectively; when the LLC evicts an ECC cacheline, the memory controller issues a memory write request. To model ECC caching for LOT-ECC5+ECC Parity, we let each XOR cacheline cover the same group of four logically adjacent data lines in N-1 logically adjacent physical pages, where N is the number of channels in the memory system. When the LLC evicts an XOR cacheline, the memory controller issues both a memory read request and then a memory write request. In general, the higher the number of data lines that each ECC/XOR cacheline covers, the lower the bandwidth overhead of the resilience scheme; the logically closer to each other the data lines that are protected by the same ECC line are, the lower the bandwidth overhead as well. The final memory bandwidth overhead also depends largely on the locality in the memory access pattern of the application and its ratio of memory write to memory read requests.

V. EXPERIMENTAL RESULTS

In this section, we compare the memory system energy, memory capacity overhead, memory traffic overhead, and the overall system performance for different resilience schemes.

A. Memory System Energy

Figure 10 shows the memory energy reduction over the baselines in memory systems that are equivalent in physical bandwidth and size to the quad-channel commercial ECC memory systems. The figure shows that for the eight workloads

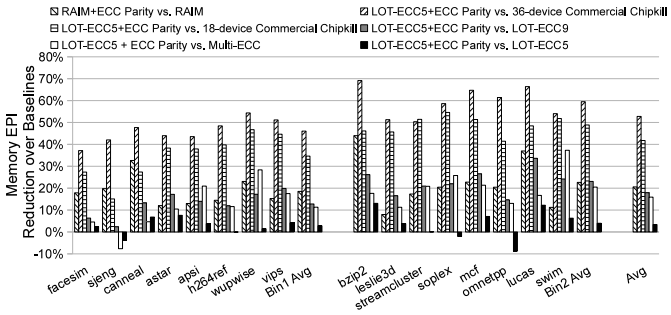

Fig. 10. Memory EPI reduction in systems equivalent in physical bandwidth and size to one of the quad-channel commercial ECC memory systems.

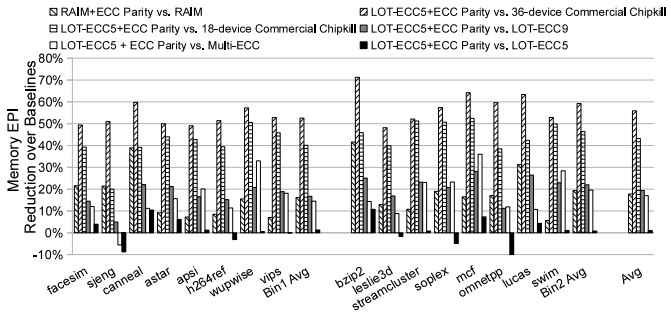

Fig. 11. Memory EPI reduction in systems equivalent in physical bandwidth and size to one of the dual-channel commercial ECC memory systems.

with higher memory access rates (i.e., Bin2 workloads), the average reduction in memory system energy per instruction (EPI) is higher than that of the eight workloads with lower memory access rates (i.e., Bin1 workloads): for Bin2 workloads, the EPI reductions are 59.5%, 48.9%, 23.1%, and 20.5% vs. 36-device commercial chipkill correct, 18-device commercial chipkill correct, LOT-ECC9, and Multi-ECC, respectively, while for Bin1 workloads they are 46.0%, 34.6%, 12.8%, and 11.3%, respectively. This is because LOT-ECC5+ECC Parity reduces memory energy by reducing the number of chips accessed per memory request; therefore, the smaller the memory access rate of the workload, the smaller the amount of memory energy savings. For the same reason, the energy reduction of RAIM+ECC Parity for Bin2 workloads (i.e., 22.6%) is higher than that for Bin1 workloads (i.e., 18.5%). The same trend exists in memory systems that are equivalent in physical bandwidth and size to one of the dual-channel commercial ECC memory systems, as shown in Figure 11. On the other hand, the memory EPI of LOT-ECC5+ECC Parity is similar to that of LOT-ECC5, as shown in Figures 10 and 11; the primary advantage of LOT-ECC5+ECC Parity over LOT-ECC5 is in terms of capacity overhead, as will be presented in Section V-B. Finally, both Figure 10 and 11 show that for a few workloads (e.g., *sjeng*, *omnetpp*, etc.), there is a slight increase in memory energy compared to one or more baselines. This is because the memory traffic overhead for updating ECC parities is higher than the memory traffic overhead of one or more baselines for these workloads. Detailed analysis of memory traffic overhead will be presented in Section V-D.

The overall memory system energy reduction described above can be attributed to reduction in both dynamic energy (i.e., energy consumed by read, write, and activate commands) and background energy (i.e., all other energy consumptions in the memory system). LOT-ECC5+ECC Parity and RAIM+ECC Parity consume lower dynamic energy than their respective baselines because they require reading/writing to fewer chips per memory request due to their smaller rank size. Figure 12 presents the reduction in dynamic energy per instruction of LOT-ECC5+ECC Parity and RAIM+ECC Parity over the dynamic energy per instruction of the baselines. LOT-ECC5+ECC Parity and RAIM+ECC Parity also consume lower background energy than their respective baselines because fewer chips have to be switched to active mode per memory request than the baseline systems; as such, when a DRAM chip is put into sleep mode it can often remain in sleep mode for longer before it needs to be put back into active mode to serve a memory request. Figure 13 presents the reduction in memory background energy per instruction of schemes with ECC Parity over the baselines.

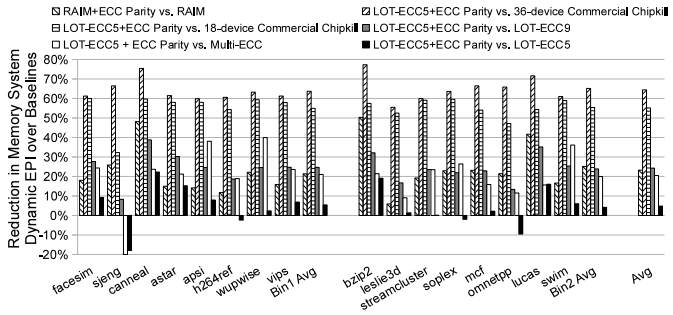

Fig. 12. Reduction in memory system dynamic EPI over baselines in systems that are equivalent in physical bandwidth and size to one of the quad-channel commercial ECC memory systems.

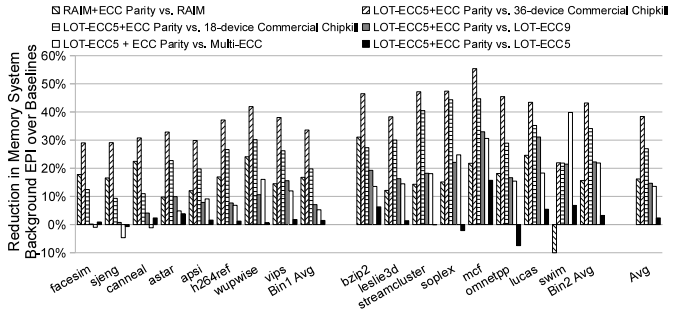

Fig. 13. Reduction in memory system background EPI over baselines in systems that are equivalent in physical bandwidth and size to one of the quad-channel commercial ECC memory systems.

B. Capacity Overhead

Table III presents the memory capacity overheads of various memory error resilience schemes. Notably, Table III shows that applying ECC Parity to LOT-ECC5 reduces the capacity overhead from an impractically high 40.6% down to 16.5% in an eight-channel system. Similarly, the capacity overhead of a 10-channel system with RAIM+ECC Parity (i.e., 18.8%) is significantly lower than that of an equally sized system with RAIM alone. Table III also shows that the capacity overhead savings from applying ECC Parity to a baseline is less pronounced in the smaller memory systems with fewer channels. This is as expected because the use of ECC parities reduces the capacity overhead of ECC correction bits by a factor of N-1, where N is the number of channels sharing

TABLE III. CAPACITY OVERHEADS. EOL STANDS FOR END OF LIFE.

36-device commercial chipkill correct	12.5%
18-device commercial chipkill correct	12.5%
LOT-ECC9	26.5%
Multi-ECC	12.9%
LOT-ECC5	40.6%
8 chan LOT-ECC5 + ECC Parity	16.5%, EOL avg: 16.7%
4 chan LOT-ECC5 + ECC Parity	21.9%, EOL avg: 22.1%
RAIM	40.6%
10 chan RAIM + ECC Parity	18.8%, EOL avg: 19.1%
5 chan RAIM + ECC Parity	26.6%, EOL avg: 26.9%

the same ECC parities; therefore, the higher the number of channels in the memory system, the more effective the optimization.

C. Performance

Figure 14 shows performance normalized to the various baselines in systems equivalent in physical size and bandwidth to one of the quad-channel commercial ECC systems. LOT-ECC5+ECC Parity has more ranks per channel than LOT-ECC9, Multi-ECC, 36-device commercial chipkill correct, and 18-device commercial chipkill correct (see Table II) because it requires fewer chips per rank. As demonstrated in other works [24], [23], having high rank-level parallelism in memory can improve performance. On average across all the workloads, the performance of LOT-ECC5+ECC Parity is slightly higher ($< 5\%$) than the aforementioned baselines. Similarly, RAIM+ECC Parity has a slight performance improvement (by 1.5%) over RAIM. On the other hand, compared to LOT-ECC5, which has the same number of ranks per channel as LOT-ECC5+ECC Parity, the performance difference is negligible ($< 1\%$). We do not observe any significant performance degradation for any individual workload except when comparing against 36-device commercial chipkill correct and RAIM. For example, Figure 14 shows that for some workloads such as *streamcluster*, LOT-ECC5+ECC Parity and RAIM+ECC Parity perform almost 20% slower than 36-device commercial chipkill correct and RAIM, respectively. This performance difference is due to the larger line size of 36-device commercial chipkill and RAIM. It is well known that having a larger line size can improve performance for applications with high spatial locality at the cost of a memory bandwidth overhead; for example, Figure 16 shows that LOT-ECC5+ECC Parity has 20% fewer memory accesses per instruction, on average, than 36-device commercial chipkill correct.

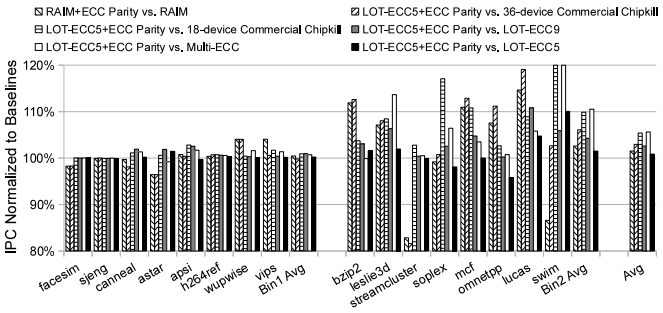

Fig. 14. Performance normalized to the baselines in systems that are equivalent in physical bandwidth and size to one of the quad-channel commercial ECC memory systems.

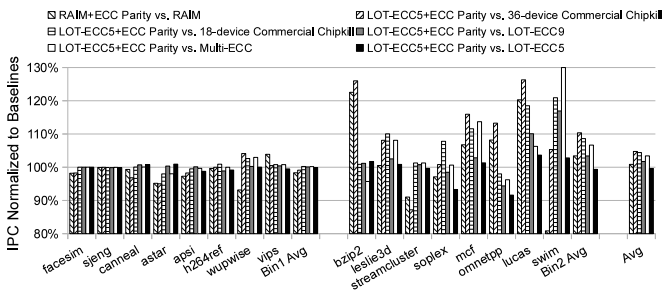

Fig. 15. Performance normalized to the baselines in systems that are equivalent in physical bandwidth and size to one of the dual-channel commercial ECC memory systems.

Figure 15 shows the performance normalized to the various baselines in systems equivalent in size to one of the dual-channel commercial ECC systems. The results exhibit similar behavior as those in Figure 14.

D. Memory Bandwidth Overhead

We measure the memory bandwidth overheads of the different schemes by measuring the number of memory accesses per instruction. Having a smaller number of memory accesses per instruction means that the bandwidth overhead is also smaller. Figure 16 shows the number of memory accesses per instruction normalized to the various baselines in systems equivalent in physical bandwidth and size to one of the quad-channel commercial ECC systems. The figure shows that there is significant variation in the normalized number of memory accesses per instruction across different workloads within the same set of comparisons (e.g., see LOT-ECC5+ECC parity vs. LOT-ECC9). This is due to the differences in the amount of locality in the memory access patterns of the workloads and in the caching details of the ECC-related cachelines of the different resilience schemes, as described in Section IV-C. Figure 16 shows that the number of memory accesses per instruction of LOT-ECC5+ECC Parity is 13.3% higher than that of 18-device commercial chipkill correct, which does not require any overhead memory requests for updating ECC bits. For scenarios where memory bandwidth is the bottleneck, this bandwidth overhead can lead to significant performance degradation (e.g., 13.3% slowdown). One way to avoid the potential performance degradation for this scenario is to use DRAM chips with a slightly higher frequency (e.g., 13.3% higher), if available. We estimate using [18] that DRAMs in a 16% faster speed bin consume roughly 5% higher memory EPI. However, this 5% increase in memory EPI due to using higher-speed DRAMs is small compared to the 48.9% reduction in memory EPI achieved by LOT-ECC5+ECC Parity over 18-device commercial chipkill correct for memory intenstive workloads (see Figure 10).

Figure 17 shows the number of memory accesses per instruction normalized to the various baselines in systems equivalent in size to one of the dual-channel commercial ECC systems. As expected, the bandwidth overhead of LOT-ECC5+ ECC Parity and RAIM+ECC Parity are higher in these smaller memory systems because each ECC parity line is shared across fewer channels; this reduces the number of data lines covered by each XOR cacheline, and therefore, increases the miss rate of the XOR cachelines.

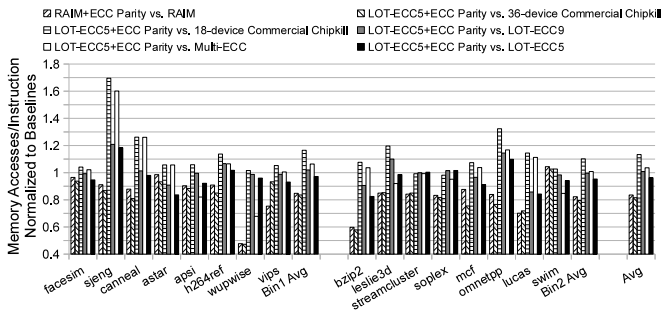

Fig. 16. Memory accesses (each 64B of data read from or written to memory is counted as an access) per instruction normalized to the baselines in systems that are equivalent in physical bandwidth and size to one of the quad-channel commercial ECC memory systems. The lower the better.

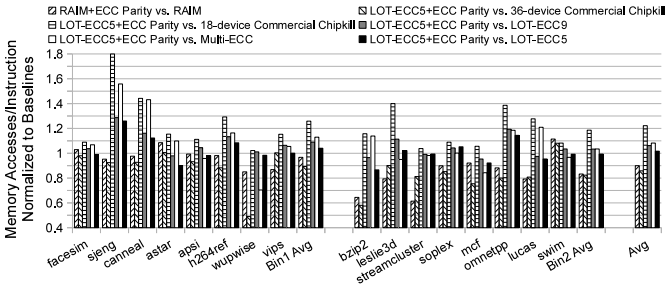

Fig. 17. Memory accesses per instruction normalized to the baselines in systems equivalent in physical bandwidth and size to one of the dual-channel commercial ECC memory systems. The lower the better.

VI. Analysis and Discussion

A. Impact on Maximum Memory Capacity

Commercial chipkill correct uses narrow X4 DRAMs, while energy-efficient chipkill correct implementations, such as LOT-ECC and LOT-ECC5+ECC Parity, use wider X8 and X16 DRAMs. Due to their higher DRAM I/O widths, these energy-efficient chipkill correct implementations require more ranks per channel than commercial chipkill correct to provide the same total physical memory capacity for the same total physical I/O width; see Table II for example. However, the total number of ranks that can reside in each channel is often limited by electrical constraints; therefore, requiring more ranks per channel to provide the same total physical capacity is problematic in scenarios where one is concerned with the maximum supportable memory system capacity.

One way to mitigate the issue above is to use a mixture of ranks with narrow DRAMs and ranks with wide DRAMs in the same channel. This may be practical because these ranks all share the same external interface (e.g., they all have a 72-bit data bus) and have the same DRAM electrical and timing parameters. By placing hot pages in ranks with wide DRAMs, one may achieve most of the energy savings of using wide DRAMs alone, while achieving high maximum supportable memory capacity through the ranks with narrow DRAMs. One drawback of the scheme is that ranks with narrow DRAMs must also use the same high strength, and, therefore, high capacity overhead ECC as the ranks with wide DRAMs, because a faulty wide DRAM can affect multiple narrow DRAMs by sharing the same I/O lanes with them [20]. Fortunately, the high capacity overheads of both types of ranks can be effectively reduced by storing ECC parities instead of the ECC correction bits.

B. Impact on a HPC System

For memory systems with ECC Parity, the effective memory capacity reduces when a device-level fault (such as bank fault, rank fault, etc., see Sections III-B, III-E) occurs. Reduced capacity may lead to thrashing to disk in the node with the reduced memory capacity. In an HPC system, thrashing in one node can lead to unacceptable performance degradation in the entire system. We note that many HPC systems contain spare nodes for taking over from faulty nodes after a checkpoint-restart [10], and, therefore, propose resolving the above issue for such HPC systems by migrating the threads in an affected node to a fault-free spare node. Migrating the threads from a faulty node to a healthy spare node also eliminates potential jittering effects due to error correction. The cost of this approach is that the entire HPC system may stall during the migration process. The duration of the stall will also include the time it takes to reconstruct the ECC correction bits (see Section III-B) before the faulty memory regions can be corrected. The overall performance degradation due to these stalls is small, however. We estimate that a large HPC system with 2PB of total memory, 128GB of memory per node, and a node NIC bandwidth of 1GB/s will be stalled only 0.35% of the time due to thread migration and the reconstruction of ECC correction bits. The calculation assumes that thread migration is performed each time a column, bank, multi-bank, or multi-rank fault occurs, and uses the average of the fault distributions of DDR3 DRAM of different DRAM vendors reported in [21].

C. Impact on Uncorrectable Error Rate

Storing ECC parities in memory, instead of always storing the actual ECC correction bits of every channel in memory, incurs a higher uncorrectable error rate because ECC parities lower error correction coverage for faults accumulated across different channels over time. To protect against the accumulation of faults, we rely on periodic memory scrubbing to detect faults and then reactively calculate and store in memory the actual ECC correction bits of faulty regions (see Section III-C). Memory scrubbing too frequently can lead to high memory power and performance overheads; on the other hand, memory scrubbing too infrequently decreases the probability of timely detecting and, therefore, reacting to a fault in a channel before a second fault occurs in a different channel in the same relative location, which cannot be corrected by ECC parities. To investigate the relationship between memory scrub rate and increase in uncorrectable error rate, we calculate the probability that two or more channels develop faults within any single detection window (i.e., the time interval between successive memory scrubs) during the lifespan of a server. Figure 18 presents the results, assuming an eight-channel system with four ranks per channel, and nine chips per rank. Figure 18 shows that for a detection window of eight hours, there is only 0.00020 chance that two or more channels develop faults in any single detection window during the seven years of operation even for a pessimistic DRAM fault rate of 100 FIT/chip. What does the probability of 0.0002 mean in terms of reliability? For simplicity, let us assume that ECC parities cannot guard against *any* combination of faults affecting two

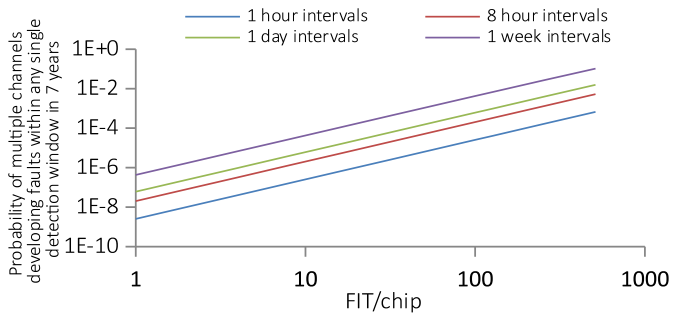

Fig. 18. Probability of faults occurring in more than one channel within any single time interval during the seven-year lifespan of an eight-channel system.

or more channels; note, however, that the ECC parities can still correct errors in different channels as long as they do not occur in the same relative locations in these channels. Under this simplistic but pessimistic assumption, the probability value means that if the memory is scrubbed once every eight hours, the memory system incurs only one additional uncorrectable error every $1/0.0002 \cdot 7 \approx 35,000$ years of operation compared to always storing the ECC correction bits for each channel. In comparison, a common uncorrectable error rate target is one uncorrectable error per 10 years per server [8]. A recent field study of failures in the Blue Waters supercomputer reports a similar failure rate of one failure per 12 years per node [16].

D. Impact on Undetectable Error Rate

ECC Parity does not modify the ECC detection bits of the underlying ECC, and therefore, does not affect its error detection coverage. For example, when applying ECC Parity to LOT-ECC5 in Section IV, the error detection coverage of LOT-ECC5+ECC Parity is the same as LOT-ECC5. However, since LOT-ECC relies on intra-chip checksums (i.e., checksums computed from data stored in a single chip) for error detection, it cannot detect address decoder errors [13]; on the other hand, commercial chipkill correct can because it uses inter-chip ECC bits (i.e., ECC bits computed from data from different chips) for error detection. Therefore, a straightforward LOT-ECC5+ECC Parity implementation may also not be able to detect address errors. However, one of the following two modifications can be used to enable LOT-ECC5+ECC Parity to reliably detect address errors depending on whether the read request is to a bank marked as faulty or not.

For banks marked as faulty, we observe that LOT-ECC contains some inter-chip ECC check bits (i.e., its ECC correction bits); these bits can be reused to guarantee detection of address errors in any single DRAM chip in the rank. This requires changing the way LOT-ECC's check bits are used. Originally, LOT-ECC uses the intra-chip checksums both to localize errors to support erasure correction and to detect errors, and uses inter-chip ECC correction bits to perform erasure correction on detected errors. Instead, the inter-chip ECC correction bits can now be used to both detect errors and perform erasure correction, while the intra-chip ECC checksums are used only to localize detected errors. Note, however, that LOT-ECC stores the inter-chip ECC check bits in a separate line from the data it protects; therefore, reusing the inter-chip ECC correction bits for error detection can increase memory access latency. ECC Parity reduces this latency overhead by reading the ECC correction bits of a line in faulty banks in parallel with the line (see Figure 6).

For banks not yet recorded as faulty, the approach described above incurs high overheads because the ECC correction bits of these banks have to be reconstructed from ECC parities. A better way to detect address errors in these banks that does not require changing the rank size or capacity overhead of LOT-ECC5+ECC Parity is to use a Reed-Solomon code as the inter-device ECC, instead of the parity originally used by LOT-ECC. The new inter-device ECC computes two 16-bit check symbols from each word of eight 16-bit data symbols that are interleaved evenly across the four X16 chips in the rank. Using one of the two check symbols per word to detect errors enables the new encoding to detect address errors; meanwhile, this modified encoding corrects detected errors by using the intra-chip checksums to localize detected errors and using both check symbols together to perform erasure correction. The check bits of the new encoding are laid out as follows: the second check symbol and the intra-chip checksums are stored via ECC parities, while the first check symbol is stored in the X8 ECC chip in the rank so that error detection can be performed on-the-fly. Note that a single check symbol per word (e.g., the check symbol in the X8 ECC chip) cannot guarantee detection of errors affecting two symbols (e.g., the two data symbols per word in a faulty X16 device). However, since a bank is recorded as faulty after encountering a small number (e.g., four, see Section III-B) of errors, the probability of an error escaping undetected in a bank before it is recorded as faulty is low. Using the fault distribution in Section III-E and pessimistically assuming that all faults are address decoder faults which manifest as random bit flips, we estimate that the combined undetectable error rate of all banks not recorded as faulty in an eight-channel system is once per 300,000 years for the new encoding. In comparison, a common undetectable error rate target is once per 1000 years per server [8].

VII. Related Work

Several works in the literature, such as LOT-ECC [22] and Multi-ECC [13], have sought to reduce the overheads of chipkill correct. Unlike these prior works, which are specific chipkill correct implementations, our proposal of storing the parity of ECC correction bits instead of the ECC correction bits themselves is a general optimization for multi-channel systems that can be applied to diverse memory ECCs (e.g., chipkill correct, double chipkill correct, DIMM-kill correct, etc.).

Our work is closely related to RAID5, an ECC technique commonly used for hard drives that relies on a parity block stored in one disk to correct errors in data blocks stored in other disks. Although originally proposed for hard drives, RAID5 has also been proposed to protect custom DIMMs that allow accesses to a single DRAM device [23], [25]. The parity line in both proposals is also a direct bitwise XOR of data lines. Compared to these prior works, we target a different problem - error resilience for multi-channel memory systems. A straightforward application of RAID5 in this context results in high capacity overhead (e.g., 50% for a quad-channel system). We extend RAID5 by first computing an ECC specified by the system designer (e.g., one that provides chipkill correct, double chipkill correct, DIMM-kill correct, etc.) from the data and then computing the bitwise parity from the correction bits of

the ECC, instead of directly from the data as does RAID5. The bitwise parity of ECC correction bits of data lines incurs lower capacity overhead than the parity of the data lines because the number of correction bits for a line is often significantly smaller than the number of data bits in the line for many ECC schemes.

Similar to our proposal, ARCC [14] and Mage [15] also share ECC resources across multiple memory channels. They combine the ECC resources of different memory channels to form a stronger ECC when needed to increase fault coverage within a single channel. For example, when faults occur in a channel, ARCC combines the ECC bits of two channels, initially designed to cover a single chip failure per rank, to store a stronger ECC that covers two chip failures per rank. Similarly, Mage combines the ECC resources of multiple channels, initially designed to only cover bit faults in a single channel, into a stronger code that covers one or two chip failures in any of the combined channels when a higher reliability target is needed. In contrast, ECC Parity neither increases nor decreases the coverage of faults within a single channel, and is, therefore, orthogonal to these works.

VIII. Conclusion

Memory error resilience is a common feature in servers and HPC systems. However, it often incurs high capacity and/or energy overheads. Conventionally, the ECC correction bits of every channel are stored in memory. We observe that because faults in different channels are often independent from one another, error correction is typically needed only for one channel at a time. Therefore, we propose a memory architectural optimization that stores in memory the bitwise parity of the ECC correction bits of different channels; the ECC parity only provides full correction coverage for faults within a single channel. We store the ECC parity for healthy memory regions and store the ECC correction bits only for fault memory regions. By trading off the resultant reduction in capacity overhead for energy efficiency, our evaluation of ECC Parity reduces memory EPI by 54.4% and 20.6%, on average across two memory system configurations, compared to 36-device commercial chipkill correct and commercial DIMM-kill correct, at similar or lower capacity overheads.

IX. Acknowledgements

This work was partly supported by NSF and AMD. We thank Vilas Sridharan and the anonymous reviewers for valuable feedback and Mattan Erez for shepherding our work.

References

[1] "Simpoints." [Online]. Available: http://cseweb.ucsd.edu/~calder/simpoint/

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

[3] A. Acosta and J. Pledge, "Dell PowerEdge 11th Generation Servers: R810, R910, and M910 Memory Guidance." [Online]. Available: http://www.dell.com/downloads/global/products/pedge/en/poweredge-server-11gen-whitepaper-en.pdf

[4] J. Ahn, N. Jouppi, C. Kozyrakis, J. Leverich, and R. Schreiber, "Future scaling of processor-memory interfaces," in *SC '09*. New York, NY, USA: ACM, 2009, pp. 1–12.

[5] AMD, "AMD, BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors," 2009. [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/325591.pdf

[6] ——, "BIOS and Kernel Developers Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors," 2013. [Online]. Available: http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf

[7] 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*, vol. 26, no. 4, pp. 52–60, 2006.

[8] D. C. Bossen, "Cmos soft errors and server design," *IEEE Reliability Physics Tutorial Notes, Reliability Fundamentals*, April 2002.

[9] T. J. Dell, "A white paper on the benefits of chipkill correct ECC for PC server main memory," 1997, IBM Microelectronics Division.

[10] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *J. Supercomput.*, vol. 65, no. 3, pp. 1302–1326, Sep. 2013.

[11] Hewlett-Parkard Development Company, L.P., "RAS features of the mission-critical converged infrastructure," June 2010.

[12] 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.

[13] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar, "Low-power, low-storage-overhead chipkill correct via multi-line error correction," in *SC '13*. New York, NY, USA: ACM, 2013, pp. 24:1–24:12.

[14] X. Jian and R. Kumar, "Adaptive reliability chipkill correct (ARCC)," in *HPCA*, 2013, pp. 270–281.

[15] 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 *SC '12*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 33:1–33:11.

[16] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *DSN '14*, 2014.

[17] P. Meaney, L. Lastras-Montano, V. K. Papazova, E. Stephens, J. S. Johnson, L. Alves, J. O'Connor, and W. Clarke, "IBM zEnterprise redundant array of independent memory subsystem," *IBM Journal of Research and Development*, vol. 56, no. 1.2, pp. 4:1–4:11, Jan 2012.

[18] MICRON, "2Gb: x4, x8, x16 DDR3 SDRAM." [Online]. Available: https://www.micron.com/~/media/Documents/Products/Data%20Sheet/DRAM/DDR3/2Gb_DDR3_SDRAM.pdf

[19] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," in *SIGMETRICS '09*. New York, NY, USA: ACM, 2009, pp. 193–204.

[20] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *SC '12*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 76:1–76:11.

[21] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng Shui of supercomputer memory: positional effects in DRAM and SRAM faults," in *SC '13*. New York, NY, USA: ACM, 2013, pp. 22:1–22:11.

[22] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, "LOT-ECC: LOcalized and Tiered Reliability Mechanisms for commodity memory systems," *ISCA*, pp. 285 – 296, 2012.

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

[24] D. H. Yoon and M. Erez, "Virtualized ECC: Flexible reliability in main memory," in *ASPLOS XV*. New York, NY, USA: ACM, 2010, pp. 397–408.

[25] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez, "The dynamic granularity memory system," in *ISCA*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 548–559.