Exploring and Optimizing Chipkill-correct for Persistent Memory Based on High-density NVRAMs

Da Zhang
Department of Computer Science
Virginia Tech
Blacksburg, USA
daz3@vt.edu

Vilas Sridharan
RAS Architecture
AMD
Boston, USA
vilas.sridharan@amd.com

Xun Jian
Department of Computer Science
Virginia Tech
Blacksburg, USA
xunj@vt.edu

Abstract—Emerging high-density non-volatile random access memories (NVRAMs) can significantly enhance server main memory by providing both higher memory density and fast persistent memory. An unique design requirement for server main memory is strong reliability because uncorrectable errors can cause a system crash or permanent data loss. Traditional dynamic random access memory (DRAM) subsystems have used chipkill-correct to provide this reliability, while storage systems provide similar protection using very long ECC words (VLEWs).

This paper presents an efficient chipkill-correct scheme for persistent memory based on high-density NVRAMs. For efficiency, the scheme decouples error correction at boot time from error correction at runtime. At boot time, when bit error rates are higher, the scheme uses VLEWs to efficiently ensure reliable data survival for a week to a year without refresh by correcting a large number of bit errors at low storage cost. At runtime, when bit error rates are lower, it reuses each memory block’s chip failure protection bits to opportunistically correct bit errors at high performance. The proposal incurs a total storage cost of 27%. Compared to a bit error correction scheme, the proposal adds chip failure protection at no additional storage cost and at 2% average performance overhead.

Index Terms—ECC, Microarchitecture, Persistent Memory Systems, Reliability

I. INTRODUCTION

Emerging high-density non-volatile random access memories (NVRAMs), such as multi-level phase change memory (PCM) and resistive random access memory (ReRAM), provide both higher density than DRAM and fast persistent memory [1]–[13]. High memory density is attractive for server systems, whose memory needs have increased due to big data, in-memory computing, and server virtualization. Persistent memory can accelerate I/O-intensive server applications by providing fast access to non-volatile storage at memory block granularity instead of the page granularity provided by storage systems. As such, dense NVRAM-based memory may find wide adoption among future server memory systems.

A key design requirement for server memory systems is reliability because an uncorrectable error can cause a system crash resulting in costly downtime; large-scale surveys report service downtime costs of millions of dollars per hour [14], [15]. In the context of persistent memory, uncorrectable errors can also cause permanent data corruption because data stored in persistent memory may not be backed up in storage.

To ensure reliability, DRAM-based server memory systems implement chipkill-correct as a standard feature to protect against both bit errors and memory chip failures. Improving the efficiency of chipkill-correct is an active area of research [16]–[31]. Providing chipkill-correct for NVRAM-based memory systems is challenging, however, because dense NVRAMs have higher random raw bit error rates (RBER) than DRAM [32]–[38]. RBER is especially high after a long time without refresh (i.e., a week to a year), and the ability to reliably tolerate long intervals without refresh is essential for NVRAMs to serve as persistent memory. As demonstrated in Section III-B, using DRAM chipkill-correct to tolerate the high NVRAM RBER incurs expensive (e.g., 69%) storage costs.

Storage systems also need to tolerate both chip failures and high RBER. Storage systems incur low storage overheads by using very long ECC words (VLEWs) to correct errors. At a given error rate, longer ECC words provide equivalent reliability to shorter ECC words with less storage overhead [39]. Applying VLEWs to main memory is challenging, however, because the access granularity of main memory is much smaller than the size of VLEWs, as VLEWs protect storage systems, whose access granularity is much bigger than main memory. Therefore, protecting main memory with VLEWs incurs high read bandwidth overhead as each read request must over-fetch many memory blocks to check the ECC. Protecting main memory with VLEWs also incurs high write bandwidth overhead because updating code bits for a write request requires a read-modify-write operation when the codeword is bigger than the written block.

This paper explores efficient chipkill-correct for NVRAM-based persistent memory. To provide efficient chipkill-correct for dense NVRAM-based persistent memory, we decouple correction of NVRAM bit errors at boot time from correction of NVRAM bit errors at runtime. We use storage-optimized VLEWs where each ECC word spans tens of blocks to ensure, at minimum storage cost, reliable data survival between the
last system outage and the next reboot; we use performance-
optimized short ECC words, where each word spans a single
block, to correct bit errors at runtime to minimize read band-
width cost. Instead of using a dedicated short ECC to correct
bit errors at runtime, our scheme reuses each block’s chip
failure protection bits to opportunistically correct bit errors
at runtime to minimize storage cost for runtime. To reduce
write bandwidth overhead for updating VLEW code bits, we
observe that dirty persistent memory blocks occupy only a
small fraction of on-chip cache capacity, on average, because
applications that use persistent memory frequently clean dirty
persistent memory blocks (i.e., memory blocks belonging to
persistent memory regions) [1]–[6]. As such, we propose pre-
serving old memory values of dirty persistent memory blocks
in the last-level cache to reduce write bandwidth overhead at
runtime; since the number of dirty persistent memory blocks
in the cache is small, this incurs a small dynamic reduction
in usable cache capacity. Compared to providing only bit
error correction for persistent memory, the proposal adds chip
failure protection while incurring no extra storage cost and
only 2% average performance cost across many persistent
memory applications.

II. BACKGROUND

A. Memory organization and technologies

Server memory systems access a group or rank of memory
chips in lockstep for each memory request. The amount of data
transferred per request is called a memory block. Each block
is typically 64B. Each memory chip typically contributes 8B1
to the accessed block.

NVRAMs capable of DRAM-like latency, such as PCM,
ReRAM, and STT-RAM, are emerging as viable memory
 technologies for future systems due to their higher storage
density and non-volatility [41]–[44]. NVRAMs can provide
higher density than DRAM for the same feature size because
the material can often store multiple logical bits per cell and
support a crossbar array architecture, which is ~50% density
than 1T1C DRAM arrays [45]–[44]. NVRAMs can provide
non-volatility because the bit cells have much longer retention
time than DRAM.

NVRAM subsystems will likely have similar chip structure
and system organization as DRAM subsystems [41]–[44],
[46]–[51]. Therefore, maximizing the reuse of existing infra-
structure and standards facilitates NVRAMs2 adoption. Many
DRAM-like NVRAM chips and modules have been prototyped
and manufactured [52]–[55]. 3DXpoint [56], a commercially
available non-volatile memory for servers, also spreads each
access across a group of chips like accesses to a rank in server
memory [57]–[59].

B. Bit Errors in NVRAMs

Figure 1 shows the RBERs of 2-bit PCM, 3-bit PCM,
and ReRAM reported in recent studies [34], [60]–[64]. For
1Although current DDR4 X4 chips transmit only 4B data per access, DDR5
X4 chips will also transmit 8B data [40].

1. Fig. 1. RBERs of memory and storage.

A. Extending prior work on NVRAM bit error correction

Prior works have explored how to correct NVRAM random
bit errors [33], [34], [67]. They protect each 64B memory
comparison, Figure 1 also includes the RBER of com-
mercially available Flash devices and the cell fault rate of 28nm
DRAM [29], [65], [66]. The RBERs of NVRAMs resembles
Flash more than DRAM.

The RBER of high-density NVRAMs increases with the
amount of time since last write or refresh; this is the reason
for the wide range of RBER in Figure 1. Similar behavior
also exists in Flash; Cai et al. [66] report that the RBER of
Flash cells three months after last write can be 100X higher
than RBER one day after last write. In general, all memories
(e.g., HDD, Flash, NVRAM, DRAM) forget data over time;
the longer since last memory refresh, the more data memory
loses. To ensure data survive across system outages, which
can last a long time, persistent memory must tolerate the high
RBER after a long time without refresh. We target a RBER
of $10^{-3}$ for persistent memory; this corresponds to the RBER
of ReRAM one year since last refresh [63] and the RBER of
3-bit PCMs one week after last refresh [60].

The raw bit errors in dense NVRAMs are predominantly
stochastic, similar to DRAM soft errors [33], [34]. In par-
ticular, bit errors in multi-level PCMs are dominated by
resistance drift and bit errors in ReRAM and STT-RAM are
dominated by retention errors, both of which are random
processes [34], [60], [61], [63], [64]. Wear errors in ReRAMs
are also probabilistic; the probability that a given cell will be
read erroneously rises gradually with the number of writes to
that cell before eventually reaching 100% [64].

III. PROBLEM

Due to the high RBER of dense NVRAMs, simply extend-
ing prior works on memory error correction to implement
chipkill-correct for dense NVRAM-based persistent memory
incurs prohibitive storage overheads. For the remainder of this
paper, we assume a reliability target of less than one block with
an uncorrectable error (UE) per $10^{15}$ blocks [60] and less than
one block with silent data corruption (SDC) per $10^{17}$ blocks
at any instant (e.g., boot time or runtime) during the mem-
ory system’s lifetime. We use standard combinatorial error
probability analysis throughout the paper, similar to [34]. For
simplicity, our analytical model assumes that write requests
do not provide any free error scrubbing effect, similar to [34].

A. Extending prior work on NVRAM bit error correction

Prior works have explored how to correct NVRAM random
bit errors [33], [34], [67]. They protect each 64B memory
block with a multi-bit-correcting BCH code. For example, to tolerate PCM RBER up to 10 minutes after last refresh, Awashi et al. [33] propose protecting each block with a BCH code that can correct up to eight bits of errors (i.e., 8-bit-EC BCH); similarly, to tolerate the RBER of STT-RAMs five seconds after last refresh, Naeimi et al. [34] propose protecting each block with a 5-bit-EC BCH. One way to tolerate $10^{-3}$ RBER for NVRAM-based persistent memory is to protect each block with a stronger 14-bit-EC BCH. BCH requires $t([\log_2(k)] + 1)$ code bits to correct $r$ bad bits when protecting $k$ bits of data; protecting each 64B memory block with 14-bit-EC BCH incurs 28% storage overhead. 14-EC BCH provides adequate protection against $10^{-3}$ RBER.

Note that simply increasing the strength of BCH ECC to 14-bit ECC only provides bit error protection, but not chip failure protection. A single chip failure in the rank can cause up to 64 bits of errors in each block. Strengthening the short per-block BCH ECC to correct the 64 bits of errors due to chip failure requires increasing its strength to $64 + 14 = 78$-bit error correction, which incurs a prohibitive 152% storage overhead.

Some prior works on addressing NVRAM RBER have proposed limiting memory density; NVRAM RBER is lower when memory density is lower. For example, Seong et al. [68] propose reducing PCM RBER by limiting the number of logical bits per cell to 1.5; this comes at a high cost of 100% capacity overhead compared to allowing 3 bits/cell. This approach is also not scalable. For example, commercially available QLC Flash cells has recently scaled up to four bits per cell; ideally, we want NVRAMs to benefit from similar scaling.

B. Extending prior works on DRAM chipkill-correct to persistent memory based on dense NVRAMs

The fault rate of DRAM cells may rise sharply to $10^{-4}$ in future generations of higher density DRAMs [28], [29], [31]. As such, several recent works have explored chipkill-correct for DRAMS with high cell fault rate [28], [29], [31]. Unlike bit errors in NVRAMs, which are random, bit errors in DRAMs are dominated by permanent cell faults, such as stuck-at-faults [28], [29], [31]. Weak ECCs can tolerate very high rates of permanent cell faults (e.g., even when 0.01% of all cells are permanently faulty) [28], [29], [31]; because permanently faulty cells can be identified at manufacturing time, memory chip or module manufactures can simply discard chips or even memory modules with patterns of permanent faulty cells that are uncorrectable by the weak ECCs at a small yield loss [28], [29], [31]. However, when the large set (e.g., millions) of erroneous bits keep changing over time, which is the case when bit errors are random, weak ECCs can no longer provide adequate protection. For example, DUO [31], the most recent work on chipkill-correct for DRAMS with high cell fault rates, only tolerates up to $10^{-10}$ random bit error rates. [2]

A simple approach to extend prior works on DRAM chipkill-correct to protect dense NVRAM-based persistent memory is to add more code bits to each codeword to correct more bit errors. XED [28] and a Samsung study [29] protect every group of 8B and 16B of data within each chip, respectively, with a BCH ECC to correct bit errors in individual data chips and then use a parity chip to correct a faulty chip in the rank. One simple way to adapt these works to NVRAM-based persistent memory is to increase the number of code bits in each BCH ECC word. DUO [31] uses a Reed-Solomon (RS) code to protect each 64B block against both bit errors and chip failures. RS ECC corrects errors at the byte granularity, at the storage cost of two check bytes to correct each erroneous data byte; when the locations of the bad bytes are known, however, as is the case for errors due to a chip failure, RS ECC can correct each bad byte using just one check byte via erasure correction. A bad byte whose location is known is called an erasure. DUO uses one RS check byte to correct each chip-failure-induced erasure and uses two RS check bytes to correct each random bit error. DUO can be extended to NVRAMs again by simply increasing the strength of RS ECC to tolerate the higher RBER. However, the above simple extensions incur high storage overheads, as shown in Figure 2; the lowest storage cost for $10^{-3}$ RBER is 69%.

IV. MOTIVATION AND CHALLENGES

We observe storage systems also need to tolerate both chip failures and high RBER. Storage systems commonly use strong ECC (e.g., 12 to 41 error correction [69], see Figure 3) to tolerate high RBER and use a parity disk/chip to correct chip/disk failures. Storage systems require very low redundancy, however; assuming eight data chips and one parity chip, the total storage overhead is only $13% + 1/8 * (1 + 13%) = 27%$ when protecting MLC Flash chips against bit errors using 41-bit-EC. Storage systems enjoy such low redundancy despite using strong (e.g., 41-bit-EC) codes by exploiting a well-known fact in coding theory that longer ECC words require less storage cost than shorter ECC words to ensure same reliability for the same RBER [39]. Storage systems have very large access granularity (i.e., 4KB) and, therefore, naturally benefit from VLEWs. Figure 3 shows BCH ECC words commonly used in commercial Flash chips; all ECC words in Figure 3 contain 512B of data.

---

2Random RBER of $10^{-10}$ is calculated from the worst case random error modeling assumption in [31] where $10^{-5}$ of DRAM cells are intermittently faulty cells and each such cell has a $10^{-5}$ error activation probability.
Figure 4 shows the total storage cost when extending storage-inspired protection to NVRAMs by using VLEWs to correct bit errors and by using one parity chip for every data chips to correct a chip failure; Figure 4 shows BCH words of different lengths for comparison. When using VLEWs with 256B of data, total storage cost to tolerate both NVRAM bit errors and chip failure reduces to 27%, which matches the 28% storage cost of bit error protection alone (see Section III-A).

A. Read Challenges when Using VLEWs

While protecting persistent memory with VLEWs enables storage-optimized chipkill-correct, it comes at high memory bandwidth overhead. Because the access granularity of main memory is much smaller than that of storage systems, each VLEW spans many blocks when used in main memory. This is worsened by the fact that each VLEW only protects data within a single chip; the 256B of data in a VLEW spans $256B/8B = 32$ memory blocks. For $10^{-3}$ RBER, each VLEW must correct up to 22 bad bits, which requires 33B of BCH code bits; as such, the code bits in each VLEW span $33B/8B \approx 4$ blocks. Using VLEWs to correct bit errors in one block requires fetching $32+4−1 = 35$ additional blocks; this translates to high bandwidth overheads, especially when RBER is high, which causes frequent error correction. While RBER is relatively lower at runtime when memory can be refreshed, it is still very high in absolute terms. For example, the RBER of ReRAM is $\sim 7 \cdot 10^{-5}$ [63] at runtime.

Under $7 \cdot 10^{-5}$ RBER, 4% of accesses still contain bit error(s); correcting bit errors for 4% of accesses incurs 4%·35 = 140% bandwidth overheads for read requests. For 3-bit PCM, RBER is also $7 \cdot 10^{-5}$ if refreshed once every second [60]. Unfortunately, refreshing NVRAMs requires correcting errors that have accumulated in NVRAMs; fetching all blocks to correct their errors once every second causes high (e.g., $\sim 1000\%$) memory bus bandwidth overhead even for small memory channels with small amount of NVRAMs (e.g., 160GB). For a more realistic refresh rate of once per hour, the RBER of 3-bit PCM increases to $2 \cdot 10^{-4}$ [60], which causes 10.3% of memory accesses to contain bit errors. Correcting bit errors for 10.3% of accesses incurs 10.3%·35 = 360% overall bandwidth overheads for reads.

One possible solution to mitigate the high bandwidth overheads is to perform VLEW error correction within NVRAM chips themselves. However, this is expensive because VLEWs are very long and strong (e.g., 22-bit-EC over 2048 bits of data). For example, Flash chips with embedded error correction suffer from either lower performance (e.g., 3X [70], [71]) or lower (e.g., 16X [72]) density compared to raw Flash chips, which rely on processor-side error correction. Flash chips with embedded correction logic also pay high (e.g., 66%) energy overheads [70] and increases cost per bit [73]. The high cost is because the memory manufacturing process is sub-optimal for implementing complex logic due to low transistor speed and low (e.g., three [74]) metal layer count while VLEW correction requires solving complex large systems of simultaneous equations [75].

B. Write Challenges when Using VLEWs

Protecting persistent memory with VLEWs also incurs high bandwidth overheads for writes because writing to memory requires updating the code bits protecting the modified data block. Because the amount of code bits in each VLEW is $33B/8B \approx 4X$ the access granularity of each chip, writing the new VLEW code bits of a written data block to memory requires four overhead write requests; this translates to 400% overheads for writes. The common approach of mitigating write bandwidth overhead via caching code bits can significantly complicate persistent memory design. If the system crashes between the write of persistent memory data or log and the write of their cached VLEW code bits, the written data and their stale VLEW code bits in persistent memory will be inconsistent with each other during system recovery and cause uncorrectable errors; these uncorrectable errors can cause irrecoverable persistent memory data corruption if they affect committed data that are beyond rollback. While co-designing persistent memory programming and VLEW code bits caching to ensure reliable survival after system crash may

---

1 If each VLEW protects data across all chips in a rank, a chip failure can cause hundreds of bit errors in up to all VLEWs in the affected rank, rendering all VLEWs uncorrectable; correcting a faulty chip via the parity chip requires first correcting bit errors in working chips via the VLEWs.
be possible, it is complex because many memory blocks share the same VLEW code bits.

One possible solution to the write bandwidth overhead is to embed the encoder of VLEW code bits into NVRAM chips; when the new code bits are encoded in-memory, the processor no longer needs to perform any overhead write requests to VLEW code bits to update them after regular data write requests. Unlike error correction circuits, ECC encoding circuits incur very small overheads even for VLEWs. Because BCH ECC is a linear code, encoding simply calculates the right hand side values for a system of linear equations where all left hand side values are known [76]; this is far simpler than correction, which solves this very large system of equations with many unknown variables.

While the simple solution above eliminates overhead write requests to VLEW code bits, there still remains the high bandwidth overhead of accessing old data bits to compute the new VLEW code bits. Because a VLEW is bigger than a memory block, calculating the new VLEW code bits for a write request requires both old data bits and code bits as inputs. Specifically, \( ECC_{new} = ECC_{old} \oplus ECC_{Update} \); \( ECC_{Update} = f(x) \oplus f(x') \), where \( f \) is the ECC encoding function, \( x \) is the new data to be written, and \( x' \) is the old data to be overwritten, and \( \oplus \) is bitwise XOR [76]. While \( x' \) and \( ECC_{old} \) reside in memory and thus can be fetched in memory without incurring traffic over the memory bus, \( x' \) and \( ECC_{old} \) can contain bit errors due to the high NVRAM RBER. Fortunately, bit errors in \( ECC_{old} \) are tolerable because they simply propagate one-to-one (i.e., without spreading) to \( ECC_{new} \) during bitwise XOR with \( ECC_{Update} \); unfortunately, using a wrong \( x' \) to calculate \( f(x') \) to obtain \( ECC_{new} \) can directly cause SDC. As such, the processor must fetch \( x' \) from memory for error detection/correction and send the corrected \( x' \) back to NVRAM chips; this incurs an expensive 200% bandwidth overheads for write requests.

Figure 5 summarizes the error correction and write bandwidth overheads when protecting persistent memory with VLEWs. For clarity, Figure 5 shows only one NVRAM chip, since all chips in a rank are identical, and assumes the case where old data block happens to be error-free.

V. EFFICIENT CHIPKILL-CORRECT FOR DENSE NVRAM-BASED PERSISTENT MEMORY

NVRAM RBER is lower at runtime, when memory can be periodically refreshed, than at boot time when the NVRAM has potentially gone a long time without refresh. As such, a strong ECC is needed for boot time, but a weaker but faster per-block ECC suffices to correct bit errors that occur at runtime.

Therefore, to provide efficient chipkill-correct for dense NVRAM-based persistent memory, we decouple correction of NVRAM bit errors at boot time, when bit errors may have accumulated due to a long power outage, from correction of NVRAM bit errors at runtime. We propose using storage-optimized VLEWs to ensure reliable data survival at low storage cost and using performance-optimized short ECC words to correct bit errors at runtime. We also explore optimizations for both ECCs to address their respective drawbacks and achieve low storage and performance cost.

A. Data and ECC Layout

Figure 6 shows the layout of data bits and ECC bits under the proposed scheme. Within each chip, each group of 256B data in the same row is part of the same VLEW; because the access size of each chip is 8B, each VLEW spans 256/8 = 32 blocks. Each VLEW’s code bits are located in the same row as the VLEW’s data bits by increasing the number of bits per row in each chip, similar to Flash chips [78], [79]. Each VLEW contains sufficient code bits to provide 22-bit error correction to ensure data can reliably survive the high RBER after a long time without refresh.

Each rank also contains a parity chip to tolerate a complete chip failure. Each 8B from the parity chip protects 64B of data from eight data chips. Data in the parity chip is encoded using a Reed-Solomon code instead of a parity code [76]. The total storage cost due to the VLEW code bits in each chip and the parity chip is 33/256+1/8*(1+33/256) = 27%.
B. Error Correction at Boot Time

At boot, the memory controller fetches all VLEWs to scrub all bit errors that have accumulated in persistent memory. Assuming 3GHz memory bus frequency, scrubbing a large persistent memory system with a terabyte of memory per channel takes less than 1.5 minutes. If a VLEW in a data chip reports an uncorrectable error, a chip-level fault has occurred. The memory controller uses the parity chip to correct the faulty data chip. Through erasure correction, the eight Reed Solomon check bytes can correct up to eight bad bytes in the block to help correct a complete chip failure [76]. If the chip with an uncorrectable error is the parity chip, the memory controller recalculates the parity values in the parity chip using the contents of the data chips.

C. Error Correction at Runtime

During normal operation, the memory controller fetches 64B of data and the associated eight RS check bytes from the parity chip. The memory controller uses the RS check bytes to opportunistically correct bit errors in the data without needing to fetch the VLEW code bits from the parity chip.

The eight RS check bytes can correct up to four random bytes of errors. When a memory access contains five or more errors, the number of errors exceeds the ECC’s correction capability and the ECC can miscorrect the errors, resulting in silent data corruption (SDC). 1.5 \cdot 10^{-7} of memory accesses contain five or more errors assuming an RBER of 2 \cdot 10^{-4} (see Figure 7); under 2 \cdot 10^{-4} RBER, using each block’s RS ECC to correct all bit errors yields an SDC rate of 3.2 \cdot 10^{-11} (see Appendix), which is 3,000,000X higher than the SDC target rate of 1 \cdot 10^{-17}. Assuming a lower RBER of 7 \cdot 10^{-5} still results in an SDC rate that is 18,000X higher than the SDC rate target.

To reliably correct random bit errors using per-block RS ECC, we observe that a misrecognition is more likely to appear as a large number of corrections (e.g., three or four bytes in error) than as a small number of corrections (e.g., one or two bytes in error). That is, a larger number of corrections indicates a higher probability of a misrecognition. As such, we set a threshold on the number of corrections. The threshold is less than the number of errors the per-block RS ECC can correct (i.e., four). We conservatively assume that accesses with a larger number of corrections than the threshold may be miscorrected, and fall back on VLEW correction in those cases, as shown in Figure 8. Based on Figure 7, we set the threshold to two errors because > 99.98% of accesses have two or fewer errors. When using RS ECC to correct up to two bits of errors, the SDC rate is 3.3 \cdot 10^{-22} (see Appendix), which is several orders of magnitude lower than the target rate.

Figure 9 shows the complete error correction procedure at runtime. After receiving a memory block from off-chip persistent memory, the memory controller uses the block’s RS ECC to correct errors; if RS ECC makes no more than two corrections, the memory controller accepts the RS correction results and sends the corrected memory block to the last level cache (LLC). If RS ECC either makes more than two corrections or recognizes the errors as uncorrectable, the memory controller fetches VLEWs to correct bit errors. Using VLEWs to correct bit errors frees up per-block RS ECC to correct errors from chip failures; as such, normal operation continues to benefit from chip failure protection.

On average, 0.018% of reads require fetching VLEWs to correct errors. This translates to a bandwidth overhead of 0.018% \cdot 36 = 0.6%, which is much lower compared to 140% -to 360% overhead when using VLEWs alone (Figure 5).

D. VLEW Updates at Runtime

VLEW code bits must be updated for every write request. Calculating new VLEW code bits for every write requires a read-modify-write operation, incurring a 200% bandwidth overhead (see bottom of Figure 5). However, persistent memory applications frequently write dirty persistent memory.
blocks to memory proactively through cacheline cleaning instructions such as `clwb` and `clflush` [1]–[6]. As such, dirty persistent memory blocks occupy only a small amount of on-chip cache capacity. Figure 10 shows that dirty persistent memory blocks occupy 4% of the total number of cachelines in the on-chip cache hierarchy, on average across all evaluated persistent memory applications (see Section VI). Based on the observation, we propose preserving the old memory values (OMVs) of dirty persistent memory blocks in the last level cache (LLC) at only a small dynamic cost to usable LLC capacity; preserving the old memory values (OMVs) of dirty persistent memory blocks helps to avoid the bandwidth overhead of fetching OMVs from off-chip before writing to persistent memory.

To preserve OMVs of dirty persistent memory blocks in LLC, we add two bits to each LLC cacheline’s tag. One is the “SameAsMem” (SAM) bit, which records whether the cacheline currently has same value as off-chip persistent memory. The other is the OMV bit, which records whether the cacheline holds the OMV of a dirty persistent memory block. While both bits record whether a cacheline has same value as off-chip persistent memory, the difference is that cachelines with SAM bits set are visible to accessible by memory instructions, but cachelines with their OMV bits set are not.

LLC cachelines update their SAM and OMV bits as follows. After being filled with data arriving from persistent memory or being cleaned by a cacheline cleaning instruction, an LLC cacheline sets its SAM bit; a cacheline resets its SAM bit after being filled with data from a dirty writeback from an upper-level cache. When receiving a dirty writeback to a cacheline with its SAM bit set, LLC preserves the cacheline’s OMV by setting the cacheline’s OMV bit and allocating a different way in the same cacheshet to handle the dirty writeback.

The LLC uses the SAM and OMV bits as follows. Before writing back or cleaning a dirty block, LLC searches within the dirty block’s set for a block with its OMV bit set and has the same address as the dirty block. When finding such a matching block, LLC computes the bitwise XOR of the block’s value (which is an OMV) and the dirty block’s value and sends the result to the memory controller to save the memory controller from explicitly fetching OMV from persistent memory for the memory write request; LLC also removes the matching block since its value will no longer equal off-chip persistent memory value after the memory write request. A cacheline cleaning instruction can also clean a dirty persistent memory block currently residing in an upper-level cache to off-chip memory; when such a dirty block passes through LLC, LLC looks for a matching LLC block with a set SAM or OMV bit, uses the LLC block’s value to compute the bitwise XOR to send to the memory controller, and removes the LLC block if its OMV bit is set.

While fetching OMV from LLC eliminates the bandwidth overhead of reading old block, the processor still needs to send OMV to NVRAM chips to compute the ECC update (see Figure 5); this still incurs 100% bandwidth overheads for writes to persistent memory. To address this final challenge, we propose piggybacking the old block in the new block by modifying each write request to persistent memory to send the bitwise sum of the two blocks to memory, instead of just the new block as do conventional write requests. Upon receiving a bitwise sum from a write request, each NVRAM chips can internally recover the new data by simply bitwise subtracting the old data stored in the NVRAM chip from the received bitwise sum of old and new data. Each NVRAM chip can also use the received bitwise sum to directly encode the update to the VLEW code bits because the BCH code is linear [76] like most ECCs used in memory and storage systems. Recall from Section IV-B that ECC update is \( f(x) \oplus f(x') \); because \( f \) is linear, \( f(x) \oplus f(x') = f(x \oplus x') \). \( x \oplus x' \) is the bitwise sum sent by a memory write.

Figure 11 shows the support needed within an NVRAM chip. On receiving a bitwise sum, each NVRAM chip fetches \( x' \) (old data) from the open row, bitwise subtracts \( x' \) from the received bitwise sum, and writes back the result (i.e., \( x \) or new data) to the open row; the ability to internally read-modify-write data is supported in current and emerging memory chips [29], [80]. To update the VLEW code bits, we...
is the number of memory banks and total surviving chips $R$ of all chip (see Figure 6). Because each reconfigured VLEW contains 256B of data, each striped across the rank, using it to correct bit errors only requires fetching four data blocks via four regular requests. Reconfiguration maintains the same VLEW length and strength and, therefore, incurs no additional capacity overheads.

Permanent bit faults can develop in individual memory blocks due to the limited write endurance of NVRAMs; NVRAM-based memory systems may need to disable individual worn-out blocks [86]. While VLEWs protect data at much coarser granularity than individual blocks, they are compatible with disabling individual blocks. When disabling a block, the memory controller can simply update the VLEW code bits assuming the physical bits corresponding to the disabled block in the VLEW hold only zeros. Similarly, when fetching a VLEW for error correction, the values of the physical bits in the VLEW corresponding to the disabled block can be replaced by zeros prior to performing error correction. Note that any underlying block disabling mechanism must already pay the overhead of tracking which blocks have been disabled; as such, applying VLEW protection to a memory system with block disabling does not incur additional storage overheads.

To identify worn-out blocks, prior works check whether errors remain in a block after error correction by re-reading the block right after writing it to memory [86]. This is also compatible with our proposal; after performing VLEW correction for a block, the memory controller may write the corrected block back to memory in the conventional manner (i.e., send raw data, instead of bitwise XOR, to directly overwrite data in memory) and re-read the block to identify worn-out blocks.

Prior works level wear between different data blocks by dynamically remapping blocks to different memory locations [87]. To support wear leveling, after remapping a block, the memory controller can update VLEW code bits assuming the physical bits that previously held the remapped block now contains only zeros, similar to handling block disabling above. Prior works level wear between ECC and data bits by periodically rotating the physical cells for storing ECC [88]; to support ECC leveling, while refreshing each row, the memory controller may instruct NVRAM chips to reserve a different group of bits in the row for storing VLEW code bits for the next refresh period.

Upgrading VLEW code bits for each write request requires increasing the number of code bits per write request and, therefore, may reduce NVRAM’s write lifetime. Prior work report that increasing write latency can effectively improve NVRAM write lifetime; as such, we make up for loss in write lifetime by increasing write latency [89]. Our evaluation accounts for the write latency overheads (see Section VI).

The proposal requires embedding BCH encoders in NVRAM chips. BCH code bits can be computed in parallel via one XOR tree per code bit; this allows simple memory-array-like layout using only two metal layers, as shown in Figure 13. Using CACTI [90], we calculate area to be $0.1\ mm^2$ when assuming only semi-global metal wires and that each logic gate

Figure 12 summarizes how to write to persistent memory. It eliminates the 200% write bandwidth overhead of protecting persistent memory with VLEWs (see Figure 5).

E. Discussions: Handling Permanent Faults, Compatibility with Write Leveling, Area/Latency Overheads

A permanently faulty chip in a rank may cause repeated errors in many blocks and, therefore, frequent VLEW correction, which in turn incurs high performance overheads. One solution to mitigate this problem is to retire memory affected by permanent chip failure after correcting its data and migrating them elsewhere. After correcting a faulty chip, many systems today retire the affected memory to avoid uncorrectable errors occurring due to another chip failing in the same rank later [81]–[85]; memory retirement will likely be common place for persistent memory where uncorrectable errors can cause permanent data corruption. Another solution is to remap the contents of the faulty chip to an ECC chip in the faulty rank, at the cost of replacing the per-block RS ECC bits. To efficiently correct bit errors without per-block RS ECC bits, the memory controller dynamically re-encodes each VLEW in the faulty rank from 256B of data across all surviving chips in the rank; recall in healthy ranks without chip failures, the memory controller encodes each VLEW from 256B of data in a single chip (see Figure 6). Because each reconfigured VLEW contains $256B/64B = 4$ 64B blocks, each striped across the rank, using it to correct bit errors only requires fetching four data blocks via four regular requests. Reconfiguration maintains the same VLEW length and strength and, therefore, incurs no additional capacity overheads.

Figure 12. Summary of how to efficiently update VLEWs at runtime when writing to persistent memory.
equals two SRAM cells in size (similar to [91]). We estimate latency to be 1.6ns using LSTP bulk transistor latency [92].

Under $2 \cdot 10^{-4}$ BER, 1/200 and 1.8/10000 of reads need multi-error RS correction and BCH correction, respectively. We estimate latency of correcting a multi-byte error using RS ECC to be 45ns and the area requirement to be $0.002mm^2$ based on latency and area reported for an 8-byte-EC RS decoder in [93], after adjusting for process technology and codeword length. We estimate latency of 22-EC BCH ECC to be 200ns and area to be $0.005mm^2$ based on a 32-EC BCH decoder reported in [94], after adjusting for process technology and codeword length.

VI. METHODOLOGY

We evaluated WHISPER persistent memory benchmarks [2] and SPLASH3 [95] benchmarks running in ATLAS [1], a persistent memory library. To stress the memory system, we increased the problem size of each benchmark to the maximum supported by our available software and hardware. For example, we increased the MemCached capacity setting from the default of 64MB in WHISPER to 1GB. The total memory footprint of the workloads range from 2GB to 20GB. We evaluate the workloads in Gem5 [96] by executing them in an OS running in a simulated X86 processor; the OS is Ubuntu Server 16, a recent Linux distribution. The WHISPER workloads take a long time to initialize because they begin with an empty in-memory database or data structure and gradually fill it up transaction by transaction; as such, we warmup the workloads in native execution speed via Gem5 KVM CPU until the workloads’ resident memory sizes, as reported by the simulated Ubuntu OS’ TOP utility, have increased to steady levels. The native execution warmup times range from 2-10 minutes. After native execution warmup, we use Gem5’s functional simulation to warmup the simulated processor’s cache for 500ms of simulated time. Finally, we use Gem5’s cycle-accurate simulation to measure the workload’s performance during 20ms of simulated time.

Table I shows the microarchitectural parameters of the simulated processor. We simulated four cores per processor, similar to [2]. For each Whisper benchmark, we simulate multiple processes of the same benchmark, with a single-thread per process. We use IPC as the performance metric for these single-thread Whisper workloads. For each SPLASH3 benchmark, we simulate one process with four threads; we use FLOPS as their performance metric because they are parallel floating-point-heavy scientific workloads. All Whisper and ATLAS workloads utilize both persistent memory and volatile DRAM; as such, we modeled a hybrid memory channel with one rank of DRAM and one rank of persistent memory. We map the persistent memory address ranges, as specified in each WHISPER workload, to the NVRAM ranks, and map the remainder to DRAM ranks. For SPLASH3 running under ATLAS, we kept ATLAS’ default setting of allocating all heap objects in persistent memory. Figure 14 shows the breakdown

| TABLE I  |
| Microarchitectural Parameters |
| Core | 4 cores, 3GHz, 4-issue OOO |
| L1 d-cache | 168 B entries, 64B cacheline |
| Shared LLC | 2-way, 4MB, 14 cycles |
| Memory Controller | 128 read buffer, 128 write buffer/channel |
| Memory System | One 2400MHz channel, with 1 DRAM rank and 1 persistent memory rank, 16 banks/rank |

Fig. 13. Circuit diagram of an encoder for 22-bit-EC BCH with 256B data.

Fig. 14. Workload characterization: off-chip memory access breakdown.

Fig. 15. Workload characterization: the ratio between the number of updates to VLEWs and the number of off-chip write requests to persistent memory.
of off-chip memory accesses; all benchmarks significantly exercise persistent memory.

We incorporate Ramulator [97] in Gem5 to simulate memory performance. We use Ramulator’s default 2400Mhz DDR4 parameters, default FR-FCFS command scheduling policy, and default row buffer policy, which closes a page after 50ns of inactivity. Similar to [42], we model NVRAMs by modifying DRAM timing parameters because datasheets for dense NVRAMs chips are not yet available. We set the NVRAM rank’s tRCD and tWR parameters to the NVRAM read and write latencies, respectively. For ReRAM, we model 120ns read and 300ns write latencies, similar to [89]. For PCM, we model 250ns read latency by taking the 250ns eM-metric reported in [60]. We model 600 NVRAM write latency, which is in the middle of the 100ns - 1000ns write latency range described in [60].

To model the proposal, we modify LLC to cache OMVs. We model the read bandwidth overhead due to fetching VLEWs to correct bit errors for 0.02% of read requests (see Section V-C) by randomly force-prefetching 37 blocks at 0.02% probability. Recall from Section V-E we increase write latency to make up for loss in write lifetime due to updating VLEW code bits, which increases the number of physical bits written per write request. The number of physical bits written per write request increases by $33B/8B \cdot C$, where $C$ is the ratio between the number of writes to VLEW code bits and the number of write requests to persistent memory; recall from Section V-D that a dirty row’s VLEW code bits are written only once when the memory controller closes the dirty row. When evaluating a workload, we pessimistically assume the worst-case (i.e., linear) relationship between write endurance and write latency [89] and, therefore, increase tWR by $33B/8B \cdot C$; Figure 15 shows the $C$ factor we measured for each workload and, therefore, used to calculate the increased tWR when evaluating the proposal. We note that $C$ depends on a workload’s spatial locality; to stress the proposed memory system design, we maximize $C$ by setting data item size to a small 64B for all Whisper benchmarks with adjustable data item sizes (i.e., echo, memcached, hashmap, btree, and rbtree). Finally, to account for the 1.6ns BCH encoder latency (see Section V-E) and internal read of old data (see Figure 11), we pessimistically increase tWR by yet another 20ns.

VII. EXPERIMENTAL RESULTS

Figure 16 and Figure 17 show the proposal’s performance normalized to the bit-error correction baseline under ReRAM latencies and under PCM latencies, respectively. The proposal incurs a slightly higher average performance overhead (i.e., 2.3% vs. 1.4%) under PCM latencies than for ReRAM latencies. This is because we modeled a much longer baseline write latency (i.e., 600ns) for PCM than for ReRAM (i.e., 300ns), which increases the impact of the proposal’s write latency overhead on overall performance. On average across both sets of evaluations, the proposal incurs a performance overhead of 2%. We believe this is a small cost for providing chip failure protection for persistent memory. In a large-scale field study, Vilas et al. [98] report that chipkill-correct provides 40X reliability improvement; we expect similar reliability improvement compared to protecting persistent memory with only bit error correction.

In the worst case (i.e., hashmap), the proposal incurs 14% performance overhead. According to Whisper description [2], hashmap performs only write queries, such as item deletion and modification; as such, hashmap represents the worst-case workload for the proposal, which requires increasing write latency to provide iso-write-endurance as the baseline. We note that ctree, btree, and rbtree also perform only write queries [2]; however, the proposal’s performance for these workloads is $>= 96.8\%$ normalized to the baseline. We believe these workloads’ lower sensitivity to write latency is due to the pointer-chasing memory access pattern of tree data structures, which reads from few banks (e.g., one) at a time and, therefore, reduces the probability of reading from a bank with on-going write.

While other Whisper workloads such as memcached, echo, redis, and vacation also have a high ratio of write to read memory requests (see Figure 14), they are also not performance-sensitive to the increased write latency (see Figures 16, 17). Unlike the previously discussed workloads, this group of workloads process a network request for each query, which takes up a significant portion of the query’s execution time and, therefore, reduces the overall performance impact of
increased write latency.

Figure 18 shows the fraction of writes to persistent memory whose OMV is served from LLC instead of off-chip memory. On average across all the workloads, the hit rate is 98.6%. As such, on average only 1.4% of writes to persistent memory incurs the overhead of fetching from OMV from off-chip memory. Surprisingly, barves has the highest OMV miss rate \( n = 29 \), despite only occupying 0.5% of cache capacity with dirty persistent memory blocks (see Figure 10). We believe this is because Gem5 does not enforce cache inclusivity and, therefore, reduces the probability of finding a matching LLC block when L1 cleans a dirty persistent memory block to memory or writes back to LLC.

VIII. CONCLUSION

This paper explores the problems and challenges of protecting dense NVRAM-based persistent memory with chipkill-correct (i.e., protection against both bit errors and chip failures). Chipkill-correct is a standard feature in today’s volatile server main memory. Because errors in persistent memory can cause permanent data loss, which is more severe than the loss of data in volatile memory, persistent memory requires at least the same level of protection as volatile server memory (i.e., chipkill-correct). Chipkill-correct for dense NVRAM-based persistent memory is challenging due to high RBER when stored in a long time (i.e., a week to a year) without refresh. The ability to tolerate high RBER enables the reliable survival of data in NVRAMS after system crashes/power outages, which is essential for NVRAMs to serve as persistent memory. Simply extending DRAM chipkill-correct techniques to dense NVRAM-based persistent memory requires prohibitive storage overhead of \( \geq 69\% \).

To efficiently protect dense NVRAM-based persistent memory with chipkill-correct, we decouple correction of errors at boot time from correction of errors at runtime to simultaneously achieve low storage cost and low performance cost. We use very long ECC words (VLEWs) to ensure reliable data survival for a week to a year without refresh by correcting the maximum number of bit errors at minimum storage cost; at runtime when RBER is lower, we use each memory block’s chip failure protection bits to opportunistically correct bit errors at high performance. The proposal incurs only a total storage cost of 27%. Compared to protecting persistent memory with only bit error correction, the proposed chipkill-correct adds chip failure protection at no additional storage cost and only 2% average performance overhead.

ACKNOWLEDGMENT

We thank Rakesh Kumar from the University of Illinois at Urbana-Champaign for his insightful comments. We thank Changhee Jung and Qing Rui from Virginia Tech for their help showing us how to run applications under ATLAS. We also thank Advanced Research Computing at Virginia Tech for providing computational resources and technical support for generating the experimental results in the paper.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

APPENDIX: MISCORRECTION PROBABILITY CALCULATION

We calculate miscorrection probability as the product of two terms. Term A is the probability of having a noncodeword containing at least the threshold number of errors required to cause miscorrection (i.e., \( n_{th} \)); Term B is the probability RS ECC will decode such a noncodeword (i.e., an invalid word under a given code) into a codeword (i.e., a valid word under a given code). Term A can be obtained using standard combinatorial probability analysis by taking as inputs \( n_{th} \), RBER, and the total number of data bytes (i.e., \( k \) and check bytes (i.e., \( r \)) each word contains. To obtain Term B, we denote as \( t \) the maximum number of errors one decides to correct in a codeword. Because RS ECC decodes any noncodeword within \( t \) Hamming distance\(^5\) from a codeword into the codeword [76], Term B is the probability that an uncorrectable noncodeword is \( \leq t \) distance from an unintended codeword; on average, this probability equals the total number of noncodewords that are \( \leq t \) distance away from each codeword (i.e., \( \binom{k+r}{r} C_t \cdot 2^{k-r} \)) multiplied by the total number of possible codewords (i.e., \( 2^{k \cdot r} \)), and divided by the total number of possible words (i.e., \( 2^{8 \cdot (k+r)} \)) [76].

For our per-block RS codewords, \( k = 64 \), \( r = 8 \), and every pair of codeword has a minimum distance of \( r+1 = 9 \); when using the RS ECC to correct \( t = 4 \) errors, miscorrection may occur for a noncodeword with \( n_{th} = 5 \) errors. Using these values and RBER= \( 2 \cdot 10^{-4} \), Term A and B are \( 1.3 \cdot 10^{-7} \) and \( 2.4 \cdot 10^{-4} \) respectively; this translates to an SDC rate of \( 3.2 \cdot 10^{-11} \). When \( t = 2 \), \( n_{th} = 9 - 2 = 7 \) because only noncodewords with 7 errors from their intended codeword can be within two Hamming distance from an unintended codeword and thus be mis-corrected into the unintended codeword; increasing \( n_{th} \) from 5 to 7 reduces Term A to \( 3.6 \cdot 10^{-11} \). A smaller \( t = 2 \) value also reduces Term B to \( 9.1 \cdot 10^{-12} \). This translates to an overall SDC rate of \( 3.3 \cdot 10^{-22} \) when \( t = 2 \).

\(^5\)For RS ECC, Hamming distance between two words is how many bytes the two words differ from one another.


724