Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories

Engin İpek[1] Jeremy Condit[2] Edmund B. Nightingale[2] Doug Burger[2] Thomas Moscibroda[2]

[1]University of Rochester, Rochester, NY 14627 USA

[2]Microsoft Research, Redmond, WA 98052 USA

Abstract

DRAM is facing severe scalability challenges in sub-45nm technology nodes due to precise charge placement and sensing hurdles in deep-submicron geometries. Resistive memories, such as phase-change memory (PCM), already scale well beyond DRAM and are a promising DRAM replacement. Unfortunately, PCM is write-limited, and current approaches to managing writes must decommission pages of PCM when the first bit fails.

This paper presents *dynamically replicated memory* (DRM), the first hardware and operating system interface designed for PCM that allows *continued operation through graceful degradation* when hard faults occur. DRM reuses memory pages that contain hard faults by dynamically forming pairs of complementary pages that act as a single page of storage. No changes are required to the processor cores, the cache hierarchy, or the operating system's page tables. By changing the memory controller, the TLBs, and the operating system to be DRM-aware, we can improve the lifetime of PCM by up to $40x$ over conventional error-detection techniques.

Categories and Subject Descriptors B.3 [*Hardware*]: Memory Structures; B.8 [*Hardware*]: Performance and Reliability

General Terms Design, Reliability, Performance

Keywords Phase-change memory, write endurance

1. INTRODUCTION

For the past forty years, DRAM has served as a fundamental building block of computer systems. Over this time frame, memory capacity has increased by six orders of magnitude, with corresponding reductions in feature size. Unfortunately, as semiconductor memories reach densities where individual atoms and electrons may affect correctness, atomic-level effects are threatening to bring a near-term end to DRAM scaling. For example, DRAM cells require mechanisms for precise placement and sensing of charge on an integrated trench capacitor—mechanisms that are becoming increasingly difficult to control reliably at deep-submicron geometries. The 2009 edition of the ITRS [8] points out growing concerns over DRAM scalability, indicating that no known manufacturable solutions exist beyond 40nm. In response, the industry is turning its attention to resistive-memory technologies such as phase-change Memory (PCM). PCM is one of the most promising technologies to replace DRAM because functional PCM prototypes have been demonstrated at 22nm, and the ITRS projects PCM to scale to 9nm [8, 22].

PCM's greatest limiting factor is its write endurance: at the 65nm technology node, a PCM cell is expected to sustain 10^8 writes before the cell's heating element breaks and induces a stuck-at fault, where writes are no longer able to change the value stored in the PCM cell. Moreover, as PCM scales to near-atomic dimensions, variability across device lifetimes increases, causing many cells to fail much sooner than in systems with lower variation. Unfortunately, existing solutions to managing hard errors in DRAM and Flash memory technologies do not map easily to PCM. Although recent research has investigated PCM wear leveling [20, 21, 33] and techniques to limit the need to write to PCM [11, 33], researchers have not yet focused on what happens *after* the first bit of PCM fails.

This paper presents *dynamically replicated memory* (DRM), a new technique that allows for graceful degradation of PCM capacity when hard failures occur. DRM is based on the idea of replicating a single physical page over two faulty, otherwise unusable PCM pages; so long as there is no byte position that is faulty in both pages, every byte of physical memory can be served by at least one of the two replicas. Redundant pairs are formed and dismantled at run time to accommodate failures gracefully as faults accrue. The key idea is that even when pages have many tens or even hundreds of bit failures, the probability of finding two compatible real pages—and thus reclaiming otherwise decommissioned memory space—remains high.

DRM's error detection mechanism leverages PCM's failure modes and timing constraints to detect faults with 100% accuracy regardless of the bit failure rate or the number of errors in each data block, for the same storage overhead as standard SECDED error-correcting codes. Recovery is accomplished through a simple hardware-software interface that guarantees no data loss upon failures, while requiring zero modifications to the cores, to the on-chip cache subsystem, or to virtual-to-physical page tables. The paper also describes a low overhead memory request replication and scheduling policy that delivers dramatically *better* performance than a baseline, unreplicated system when failures are rare, and that minimizes performance overhead at the end of the device lifetime.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

The time to failure of PCM cells can be represented by a normal distribution, using the coefficient of variation (CoV) as a parameter to determine the variability. If we measure the lifetime of a memory array by the point at which 50% of the capacity has been depleted, then dynamic replication increases the halfway lifetime of PCM devices by $1.2x$, $2.7x$, and over $40x$ for low (CoV = 0.1), medium (CoV = 0.2), and high (CoV = 0.3) variance cases, respectively. Low-overhead techniques to reuse blocks with hard faults can thus provide enormous lifetime increases as variance increases in denser non-volatile memories; indeed, this technique may even be necessary to permit scaling to higher densities.

2. BACKGROUND AND MOTIVATION

The continued scaling of semiconductor memory technologies is increasingly affected by fundamental limits. Recent work [11] differentiates *charge memories* (such as DRAM and Flash memory), which use electric carriers to store the state of a bit, from *resistive memories*, which use atomic arrangements to set the resistance of a memory cell to store information. Resistive memories, which include phase-change Memory (PCM), Ferroelectric RAM (FeRAM), Spin-Torque Transfer Magnetoresistive RAM (STT-MRAM), and Resistive RAM (RRAM), the latter of which includes memristors, are all candidates to succeed charge memories if and when charge memories reach fundamental limits. The ITRS projection predicts difficulties in scaling DRAM past 40nm, and the SIA Roadmap projects difficulties scaling Flash past the 20nm node; at 21 nanometers, small numbers of electrons (e.g., 10) tunneling from the floating gate will cause a Flash cell to lose its state.

Resistive memories have properties that are different from those of current high-volume charge memories. First, they are typically non-volatile, since the state of the cells is held by the arrangement of atoms as opposed to flightier electrons. Second, they are typically byte-addressable, since competitive densities are possible without the block-erase requirements of NAND and NOR Flash memory and without the block read requirement of NAND Flash. Third, they have lifetimes which are initially projected to be even higher than Flash in terms of the number of writes before failure. Fourth, writing to these technologies is typically more expensive, in terms of latency and/or energy, than a charge memory at a given technology node, since moving or aligning atoms is more expensive than moving electrons across an insulator (Flash), which in turn is more expensive than moving electrons onto a capacitor (DRAM).

Given the scaling trends, it is possible (and perhaps likely) that resistive memories eventually replace most of the semiconductor memories in systems, including those residing on the memory bus. Thus, future systems should provide the following capabilities: (1) sophisticated but low-level wear-leveling algorithms that do not require management by an I/O storage controller, (2) mechanisms to reduce the number of writes to memory when possible, and (3) mechanisms to gracefully handle cells that fail permanently while programs are running. The first two capabilities have been heavily studied, whereas this paper focuses on the third requirement for future memory systems. The implementation strategy used to handle hard faults depends on the specific memory technology, fault model, and error-correction codes. This paper develops a fault handling architecture assuming a phase-change memory system, a run-time hard fault model, and standard ECC support, all described below.

2.1 Phase-Change Memory

Phase-change memory, first proposed in 1967, is the resistive memory that is most likely closest to high-density commercial deployment. PCM is being developed commercially by several companies, including Samsung and Numonyx, and has been the subject of considerable recent interest in the architecture research community [2, 11, 20, 21, 29, 31, 33].

2.1.1 Memory Cells and Array Architecture

PCM cells are formed by sandwiching a phase-change material (often a chalcogenide such as $Ge_2Sb_2Te_5$) between two electrodes with a resistive element acting as a Joule heater between the chalcogenide and one of the electrodes. The chalcogenide can exist in a "melted" (amorphous) state with relatively unordered atoms, or a "frozen" (crystalline) state with ordered atoms. By running a low current through the device stack, the resistive drop across the chalcogenide can be measured; the amorphous state has a higher resistance than the crystalline state. The cell can be written by running a high current through the cell, heating it up to 1000°K, and then cooling it quickly to leave it in the amorphous state ("0") or slowly to allow the crystals to grow and place it in the set position ("1"). The memory array architectures are similar to conventional DRAM technologies, except that the writes tend to be narrower due to high current densities, and the sense amplifiers can be shared among more banks since PCM cells can be actively driven to be read, as opposed to reading an effervescent charge from a DRAM capacitor.

2.1.2 Endurance, Fault Models, and Wear-Leveling

Potential failure mechanisms for PCM have been extensively studied and characterized in the literature, including radiation induced transient faults [3, 17, 18, 26], proximity effects such as read or write disturbances [7, 10, 19, 26], data retention issues [7, 10, 19, 26], and hard errors due to finite write endurance [2, 3, 7, 9–11, 17–21, 26]. While the first three do not represent a cause for concern in practice, hard failures due to write cycling have historically been one of the most serious impediments to successful commercial deployment of PCM technology.

Radiation Effects. PCM cells are not susceptible to radiation induced transient faults in the foreseeable future, due to the high energy required to change their state [17, 18, 26]. In addition, unlike the case of DRAM where bitlines are precharged and are left floating to sense an effervescent charge from an integrated trench capacitor, PCM cells are actively driven during reads; this makes the bitlines immune to radiation effects. Soft errors can still manifest on row buffers and on interface logic (e.g., I/O gating circuitry), but these can be protected in a straightforward way by applying parity or ECC prior to buffering the data [26], exchanging parity bits during bus transmissions [17], and reissuing requests in the case of a failure. No additional storage within the PCM data array is required for protection against transient faults.

Proximity Effects. Since PCM relies on resistive (ohmic) heating at high temperatures (1000°K) to program a cell, it is important to ensure that nearby cells do not undergo unintentional bit flips during a write operation. Fortunately, thermal isolation between PCM cells has been shown to be highly effective and scalable, reducing cross-cell heat contamination sufficiently that write disturbance of adjacent cells is not considered a problem in current or future technology generations [10, 19, 26]. Experimental results confirm that temperatures decrease sharply with distance, falling from 1000°K near a programmed heating element to less than 350°K within 80nm of the heater [10].

Data Retention. Although the expected data retention time of an isolated PCM cell at room temperature is in excess of 300 years, retention time can decrease appreciably due to environmental conditions in a large, dense, and heavily exercised memory array [7, 10, 19, 26]. In PCM, the amorphous phase of the GST stack represents a meta-stable material state with respect to the stable

crystalline phase, making it possible for a relatively modest amount of electrical or thermal energy to accelerate data loss. Nevertheless, even under a worst-case, sustained write cycling scenario at 430°K, retention times greater than 10 years have been demonstrated [10, 19]; thus, performing a highly infrequent (e.g., once a year) "refresh" operation would suffice to guarantee data retention.

Write Endurance. The major fault model for PCM cells is permanent failure of a cell, which occurs when the heating element detaches from the cell material due to the continued thermal expansion and contraction resulting from writes [9, 11]. Permanent faults occur in individual cells, where the bit in the cell becomes "stuck at" a fixed one or zero when the heater detaches. Beyond this point, it becomes impossible to alter the state of the cell, but the cell contents can still be reliably read [7, 10]. In current technologies, cells are expected to be able to sustain 10^7 to 10^8 writes on average before failing, and the number of writes before failure is expected to increase as PCM geometries are shrunk. However, the reliability of a large memory array is typically determined by the endurance of a relatively small number of weak cells, which can reduce the perceived lifetime—and thus, the product specification—of a 128Mb array to as low as 10^6 write cycles, even with aggressive process and cell programming optimizations [7].

This paper assumes a fault model where hard, permanent bit failures occur in individual cells with no spatial correlation (i.e., a bit failure in one position has no implications for the spatial position of the next bit failure on the chip). These assumptions are supported by private communications with Numonyx [6], one of the largest PCM developers, which confirm the following reliability characteristics: (1) PCM failure modes result in reliably stuck bits that are uniformly distributed across all bits cycled, and these errors can be detected at the time the cells are written by reading the cells after each write to ensure that the stored data are correct; (2) no large systematic effects are observed within a die, and in general, if such correlations were observed, they would likely be artifacts of such positional effects as wordline and bitline location within the array, which would be compensated and trimmed using additional circuitry as standard practice; and (3) expected cell endurance at 65nm is 10^8 writes [6, 18].

2.2 Error Correcting Codes

Error correcting codes (ECC) add correction bits to a group of data bits to detect and/or correct errors. For these codes to be effective, they must tolerate errors in the ECC bits as well. These codes are often distinguished by specifying the number of bit errors they can *detect* and the number of errors they can *correct*. Parity is the simplest ECC scheme, which adds one bit to a data block to ensure that the number of set bits is always even. That way, if any bit is flipped, the number of set bits will be odd and a single bit error will be detected; thus, parity detects one error and can correct none.

SECDED ECC stands for single-error correction, double-error detection, and is the most common ECC code that can correct errors. It is highly useful in technologies vulnerable to soft errors, since a single-bit soft error can be detected and restored in place. In semiconductor memories, the (72, 64) Hamming code is widely used for SECDED ECC, which adds eight ECC bits to each 64-bit word. Higher-order corrections involving Reed-Solomon codes are possible, but they require both higher bit overhead and higher computation overhead. Since this paper focuses on errors in memories that may include non-block storage such as physical memory, we assume only low-overhead ECC codes, such as Hamming codes, that are tractable for byte-addressable memories.

Unfortunately, existing error correction schemes for DRAM and NAND Flash are inappropriate for PCM. For example, when applied in the context of hard errors, SECDED ECC requires the operating system to decommission an entire physical page as soon as the first (correctable) bit error in that page manifests itself. This approach may be acceptable if failures are rare and start to accrue near the end of the device's life; however, if failures begin to accrue early due to process variations, this overly conservative strategy causes large portions of memory to quickly become unusable.

MLC NAND Flash, on the other hand, relies on stronger ECC mechanisms based on Reed-Solomon codes; these codes are able to detect and correct many tens of failures over a large, 16KB block before the device has to be decommissioned. Unfortunately, Reed-Solomon codes require expensive polynomial division calculations that are difficult to implement at memory bus speeds [13]. More importantly, these codes require calculating and updating a large memory block (e.g., 16KB) on each write; while this overhead is naturally amortized over the block-oriented NAND Flash interface, it is inappropriate for a byte-addressable device such as PCM.

In all, the impending end of DRAM scaling in the post-40nm era, coupled with the inherent variability of PCM cell lifetime at near-atomic feature sizes, requires us to reconsider memory-system reliability from the ground up, and demands error tolerance schemes that go far beyond the capabilities of existing techniques used in NAND Flash or DRAM.

3. DYNAMIC REPLICATION: RECYCLING FAULTY PAGES

We first provide an overview of a dynamically replicated PCM subsystem (Section 3.1). We then describe the necessary hardware (Section 3.2) and OS (Section 3.3) support to enable dynamic replication, and we provide the theoretical underpinnings of the proposed replication algorithm (Section 3.4). We present our discussion in the context of a DDRx-compatible PCM interface (similar to [11]); dynamic replication with other memory interfaces is possible and is left for future work.

3.1 Structure and Operation of Dynamically Replicated Memory

To facilitate dynamic replication, we introduce a new level of indirection between the system's physical address space and PCM in the form of a *real address space* (Figure 1). Each page in the physical address space is mapped to either one *pristine* real page with no faults, or to two faulty but *compatible* real pages—pages that have no failures in the same byte position, and can thus be paired up to permit reads and writes to every location in the corresponding physical page.

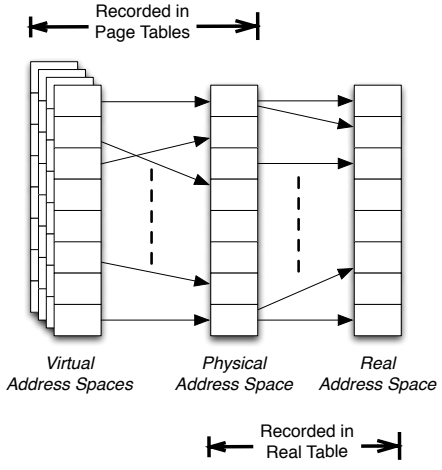

Figure 1. Example of a dynamically replicated physical address space.

Figure 2 illustrates the concept of compatibility by presenting two pairs of real pages, one compatible and the other incompatible. In each pair, the dark colored page (P_0 or P_2) represents the primary copy, and the light colored page (P_1 or P_3) is a backup page. In the figure, pages P_0 and P_1 are compatible since there is no byte that is faulty in both of them. If a single physical page were to be replicated on P_0 and P_1, P_0 could serve requests to bytes B_0 and B_7, P_1 could satisfy accesses to B_4, and accesses to all other bytes could be served by either page; in this way, a single physical page could be represented by and reconstructed from two faulty, unreliable real pages. Unlike P_0 and P_1, however, pages P_2 and P_3 both have a fault in byte position B_4, and are thus incompatible: if P_2 and P_3 were paired and a physical page mapped onto them, it would not be possible to read or write the data in B_4. The key idea is that, even when pages have many tens or even hundreds of bit failures, the probability of finding two compatible real pages—and thus reclaiming otherwise decommissioned memory space—remains high.

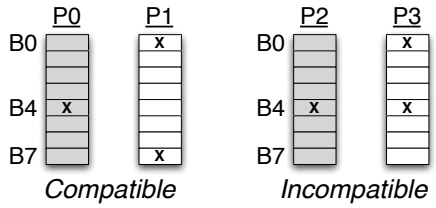

Figure 2. Example of compatible (left) and incompatible (right) pages.

Unlike virtual-to-physical mappings that need to be tracked on a per-process basis, the physical-to-real address mappings can be stored in a single, global *physical-to-real table* kept in PCM. A page is indexed by its physical page number, and each entry has two 64-bit fields that list the real page numbers of the two replicas associated with the physical page. Using 4 KB pages, this design makes it possible to address up to 16 TB of "real" PCM storage. While a physical-to-real table is not strictly necessary for correctness, this extra level of indirection is nevertheless desirable to achieve reasonable performance and to reduce storage overheads. In the absence of a physical-to-real table, standard virtual-to-physical page tables would need to be modified to store up to two physical addresses for each virtual address, and when a failure is detected at a physical address, the operating system would need to search every entry of every process's page table to find and update the relevant entries. Since we have as many page tables as we have processes, and since there can be an arbitrarily large number of virtual mappings to a single physical address, this update can be both complex and slow. The additional layer of indirection provided by the physical-to-real table eliminates this problem by giving us a table with two important properties: first, there is only one such table per PCM device (not per process), and second, there is at most one physical mapping for any given real page. Because the size of the physical address space is bounded by the size of the real address space, this table can be stored as a single contiguous array in PCM.

This additional layer of indirection is not entirely new; virtual machines use shadow page tables to manage the mapping between a virtualized operating system and the real page tables kept by the host OS or hypervisor. However, just as with virtual machines, the additional layer of indirection means that on a TLB miss the hardware page walker must first walk the page table, and then index into the physical-to-real table to put the real addresses into the TLB. Since the second lookup requires only a single access (indexing into the table) rather than an additional page table walk, the additional overhead is considerably lower than similar overheads found in shadow-page tables used for virtualization.

Because PCM can be used for non-volatile storage, the pairs described by the physical-to-real table must persist along with the data in the pages it points to; thus, the physical-to-real table must also be kept in PCM. However, because the table is used for the physical-to-real translation, we cannot use our page-pairing scheme for the table itself. Instead, we store three independent replicas of the table. In cases where the entry in the primary copy of the table is faulty, the system tries the second replica and then the third replica; if all three replicas of an entry are bad, the corresponding physical page is decommissioned. Since entries in the table are updated only when a page is remapped to a new pair, they take substantially less wear than the rest of PCM. Three copies of the physical-to-real table means 24 bytes for every 4 KB page, or about 0.59% of the available PCM; for 4 GB of PCM, three copies of the physical-to-real table occupy approximately 24MB of space.

To successfully retrieve data from PCM, hardware must track (and in some cases, access) both real addresses for each (cache-block sized) memory reference, and the OS needs to ensure that no incompatible pages are paired at each point in time. Section 3.2 discusses the necessary hardware and OS support to accomplish this.

3.2 Hardware Support

3.2.1 Tracking Real Addresses

Dynamic replication of system memory requires the memory controller to know the real addresses for both replicas associated with each physical location when accessing PCM. To determine these real addresses, the size of each entry in the data array of the d-TLB is adjusted to accommodate two 32-bit virtual-to-real mappings. On a d-TLB miss, the hardware page walker first accesses the conventional page table to locate the relevant page table entry (PTE), which contains a physical address, and then it obtains the corresponding real address(es) by indexing into the physical-to-real table. Although this approach doubles the area of the data array for the d-TLB, both the number of tags and the width of each tag in the TLB remains unchanged. The associative tag lookup typically dominates TLB lookup time, and consequently, doubling the size of the data array has a negligible impact on the overall TLB access time.[1]

Although the TLB records the real addresses of both the primary and the backup pages, each level of the cache hierarchy still stores only a single, fault-free copy of the data to avoid potential coherence problems and to prevent wasting precious cache space. If the caches are virtual, then a single TLB access is performed on a last-level cache miss to obtain the relevant virtual-to-real mappings. On the other hand, if the caches use real addresses for tagging, the data is always stored using the tag associated with the primary real page. A TLB lookup is performed prior to accessing the cache, and on a last-level cache read miss, the two real addresses are sent to the memory controller. However, handling writebacks with real tags requires additional support since the cache does not track the address of the backup page. To recover the address of the backup cache block, the TLB needs to be augmented with one additional associative port to search the backup address, using the primary real address as a tag. Our design assumes virtual L1 and L2 caches.

3.2.2 Detecting Faults

Historically, DRAMs with high reliability requirements have used single-error correction, double-error detection (SECDED) ECC to

[1] At 45nm, delay estimates from CACTI [28] indicate a 2.6% increase in access time when the size of the data array in a 64-entry, fully associative TLB is doubled.

combat failures. This choice is justified by the fact that the primary failure mechanism of concern in DRAMs is transient faults: the memory controller can periodically scrub DRAM and correct any errors flagged by SECDED ECC, and the scrubbing period can be adjusted to contain the probability of an unrecoverable error (double-bit or more errors accumulating in a codeword) within acceptable bounds. Being able to tolerate many tens or even hundreds of failures per page requires an error detection mechanism that goes far beyond the capabilities of conventional SECDED ECC.

In DRM, errors are detected by having the memory controller issue an additional read after each write to determine whether the write succeeded. The key observation that makes this approach practical is that writes are much more expensive (in terms of both latency and energy) compared to reads in PCM. If a "checker" read is issued after each array write to PCM, the read latency is largely amortized over the write latency, with only a small impact on overall system performance.

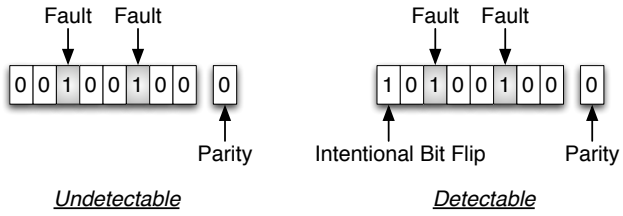

Figure 3. Example of an undetectable error (left) and the intentional bit flip performed to make it detectable (right).

If the checker read determines that the write did not succeed, the fault has to be recorded so that future reads from the same location do not return erroneous values. To do so, we associate a single parity bit with every byte in PCM, and on every write, the memory controller sets this bit so that the total number of ones in the codeword (parity plus data) is even at each point in time. (This results in a 12.5% storage overhead, which is equivalent to the storage cost of implementing conventional SECDED ECC.) If the checker read indicates that data was written to PCM incorrectly, then the controller checks the parity of the resulting codeword. If the parity bit has correctly flagged the error, then we are assured that future reads issued to this location will be able to determine that the location is faulty and use the backup page instead. If the parity bit did not correctly flag the error, then the memory controller intentionally flips one of the bits in the block, re-writes the data, and issues another checker read (Figure 3). In this way, the desired inconsistency between the parity bit and the parity of the block is generated. The controller attempts this bit flip as many times as needed until parity detects the fault, and if it cannot generate the desired inconsistency, then the page must be discarded, as we will discuss in the next section.

Once a location has been marked as faulty, the memory controller attempts to write the data to the backup location for that page, if it exists. If the backup page fails, then it too is marked as faulty using the procedure described above, and then we invoke our recovery mechanism, which is described next. If there is no such backup page, we proceed straight to recovery.

3.2.3 Recovering from Faults

When a write fails on both locations in a given pair, or on any location in a pristine page, the memory controller must detect the fault via the checker read and initiate recovery by copying the data to a new location in PCM.

To accomplish this task, the memory controller relies on an on-chip data structure called the *ready table*, which contains a list of real pages (or real page pairs) available to use when a write failure occurs. The ready table is a 128-entry SRAM array co-located with the controller's scheduling queue. Each table entry contains space for two 32-bit values indicating the real addresses for the pages supplied by the operating system. These values can indicate either a pair of pages or a single pristine page, in which case the second value is 0xFFFFFFFF. The ready table is kept in volatile memory, since it can be reconstructed at boot.

The ready table functions as a circular buffer that allows the memory controller and the operating system to exchange pages. The operating system places compatible pairs or pristine pages into the ready table, and the memory controller consumes these pages and replaces them with incompatible pages that must be re-paired. To indicate which part of the table is which, the table contains a memory controller pointer (`mc`) and an operating system pointer (`os`). Entries from the `mc` pointer up to the `os` pointer (exclusive) are ready for the memory controller to use, and entries from the `os` pointer to the `mc` pointer (exclusive) are dead pages for the operating system to handle. If both pointers point to the same location, then all pairs are ready for the memory controller.

On a fault, the memory controller first signals all of the cores to flush their pipelines and to stop fetching. It then gets the address of the next available page from the ready table (either pristine or paired), and starts copying the data from the old page to a new location. Once this is done, the controller overwrites the ready table entry used in the transfer with the addresses of the page(s) where the fault was observed (now obsolete) and increments the `mc` pointer. If a second fault is encountered in the process of copying the data to the new location, the controller increments the `mc` pointer without replacing the entries (so that the bad pair will be recycled) and retries the whole process with the next available entry in the ready table.

Once the data is copied successfully, the controller initiates L1 and L2 cache flushes (note that fetch is still stalled on all cores at this point). This cache flush is necessary to ensure that any data in the cache subsystem tagged with addresses to the old page are correctly steered to the new page. The controller monitors all writes in its transaction queue during the flushing period, and performs such writes to their new, up-to-date locations. Once this process is complete, the controller initiates a hardware page walk to determine the physical address that was just remapped, and it updates the contents of the physical-to-real table with the new mapping. If additional failures occur while flushing the cache, the same procedure is used to remap any additional failed pages.

If a real page is determined to be unrecoverable, either because it has too many bit failures or because there was a location that could not be correctly flagged as a failed location, then that page's index is not written back to the ready table for further processing by the operating system. Since the operating system never sees the failed page's index, that page will be effectively decommissioned.

Finally, there are two cases that generate hard, unrecoverable faults. First, if the `mc` pointer is equal to the `os` pointer after being incremented, then we have run out of ready table entries. Second, if the memory controller cannot update any of the three copies of the physical-to-real table, then that physical page can no longer be correctly mapped. In either of these cases, the memory controller generates a non-maskable interrupt for the operating system. These cases are negligibly rare; thus, the operating system is expected to simply halt the system if they occur.

The ready table is made visible to the OS by memory-mapping its contents and by redirecting any reads or writes referencing the corresponding addresses to it. The table is periodically refilled by the operating system using a timer interrupt, as we will explain in Section 3.3.

3.2.4 Scheduling Accesses to Primary and Backup Pages

One potential drawback to dynamic replication is the possibility of performance loss due to higher memory system contention, and due to the increased latency of retrieving both replicas on a PCM access. Figure 4(a) shows an example of how accesses to the two replicas are generated and scheduled with a *lazy replication policy*. Upon receiving a read to location A, the memory controller inserts a read request into its transaction queue. After the read gets scheduled (1) and returns from PCM (2), the controller checks the parity bits in the block to detect any faults. If a fault is detected, the controller inserts a read request for the replica into its transaction queue (3); when this read issues (4) and eventually completes (5), the controller returns the data back to the last-level cache. To maximize the chances of retrieving fault-free data on the first trial, the OS sets the page with fewer bit failures as the primary copy in its pairing algorithm. Although lazy replication mitigates unnecessary writes (and thus, traffic and wearout) to a backup page, the access latency is effectively doubled in cases where the primary page cannot produce the block by itself.

An alternative strategy that aims at eliminating this problem is an *eager replication policy* (Figure 4(b)). Under eager replication, the memory controller immediately inserts a request for the backup block in its transaction queue. In the pairing process, the OS chooses the primary and backup pages so that they reside on different DRAM banks. Consequently, bus transactions for the primary and backup pages proceed in parallel according to the DDR2 protocol (1 and 2), taking full advantage of PCM's bank-level parallelism. On writes, the memory controller waits for both checker reads to complete, and then it removes both the primary and the backup requests from its transaction queue. On a read request, however, the request is deemed complete as soon as one of the two accesses (either the primary or the replica) completes with fault-free data, and both requests are removed from the transaction queue immediately. As a result, load balance across PCM banks improves significantly since the replica in the least-loaded of the two banks can satisfy the entire request, which was previously impossible since each PCM bank serves a distinct set of addresses. As we will show in Section 5, this also has a dramatic impact on overall system performance (anticipated by earlier theoretical work on randomized load-balancing algorithms [1, 15]), and surprisingly, makes it possible for a dynamically replicated PCM subsystem to deliver *better* performance than a baseline, conventional system.

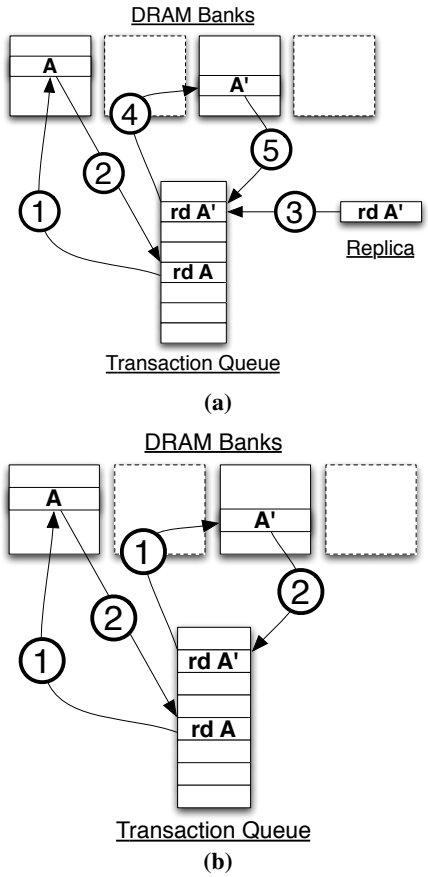

Figure 4. Example of memory access scheduling with lazy (top) and eager (bottom) replication policies.

3.3 OS Support

When failures occur, the memory controller places the failed pages back in the ready table for further processing by the operating system. The operating system's job is to refill the ready table with new compatible pairs so that the memory controller can handle further failures. The memory controller indicates that this processing is necessary by issuing an interrupt whenever it consumes pages from the ready table; this interrupt causes the operating system to signal a high-priority kernel thread whose sole job is to refill the ready table.

The operating system maintains two linked lists of real pages: the *incoming list* and the *unmatched list*. The incoming list contains all failed pages that were reported by the memory controller via the ready table. The unmatched list contains all pages that the operating system has failed to match with other pages in the unmatched list. As failed pages arrive from the memory controller, they are added to the incoming list; as pages are needed for the ready table, they are pulled off the incoming list and matched against the unmatched list.

Both of these lists are singly-linked lists maintained by storing pointers in the real pages themselves. Since any 64-bit entry in a real page might be faulty, the pointer to the next page is the first aligned 64-bit pointer with no faulty bytes (according to parity). Because we remove pages with more than 160 bit failures (see Section 5.1), there will always be such a pointer, often in the very first word of a page. These pointers can be designed to point directly to the location of the next pointer in the subsequent page (as opposed to the base of the page) so that the list can be traversed without searching for a valid 64-bit pointer at each step. The head of each list is a page stored at a well-known location in PCM, which allows the lists to be recovered on reboot. For the purpose of these reads and writes, the memory controller provides raw access to real pages; however, aside from these writes, the operating system accesses physical addresses only.

When the refill thread is signaled, its task is to refill any entries between the `os` pointer and the `mc` pointer. First, it scans all entries between `os` and `mc`, placing any valid page indices on the tail of the incoming list. Then, it refills each entry by creating a new pair. To create a new pair, it removes the head of the incoming list. If this page is pristine, it does not need to be paired; however, this case only occurs when the device is new. If the page is not pristine, then we traverse the unmatched list looking for a match. If a match is found, we remove the matched page from the unmatched list. If a match is not found, we place our unmatched page at the tail of the unmatched list and try again with a new page from the incoming list. Once a pair is found, we place it in the ready table at the `os` pointer and then increment `os`. When `os` catches up to `mc`, we are done. Note that because the memory controller only accesses elements of the ready table between `mc` and `os`, and because only

one core per memory controller runs this algorithm, there are no synchronization problems on the ready table entries. In particular, the memory controller is free to take additional faults and consume more entries from the ready table while the operating system is refilling other portions of the table.

Matches between two pages are determined by issuing reads to both pages, computing parity to determine which bytes are faulty, and then comparing the results from both pages to determine whether they have faults at the same location. These comparisons, which are the most expensive portion of the matching algorithm, are dominated by throughput-bound sequential reads from PCM that operate at near-perfect data bus efficiencies over a DDR2-800 bus; thus, with 8KB of data and at 800MB/s of throughput, we need approximately 10 μs for each comparison.

When the device is first initialized, we reserve a pool of pages by adding them directly to the incoming list. This reserve pool ensures that we will always have a large set of pages available for pairing and can thus form pairs on demand. Even under conservative assumptions about the probability of forming a new pair of compatible pages, a pool of 10,000 reserve pages (40 MB) per memory controller is sufficient for this purpose. Note, though, that forming pairs on demand is not strictly required for correct operation, so in practice, we could use far fewer than 10,000 reserve pages.

As mentioned above, the operating system can also receive non-maskable interrupts from the memory controller in the event that a hard fault occurs at the memory controller due to an empty ready table or a triple failure in the physical-to-real table. Both of these events occur with negligible frequency and will cause the system to halt.

3.4 Dynamic Pairing Algorithms

The algorithm used by the operating system to construct pairings of compatible pages in the unmatched list is a critical component of DRM. Given the set of non-pristine physical pages in the unmatched list, an efficient algorithm will be able to construct more real memory pages, and will recover more PCM capacity. In this section, we compare the pairing algorithm proposed above with a capacity-optimal algorithm and show that the proposed algorithm achieves a near-optimal solution, while being efficient.

3.4.1 Optimal Pairing for Capacity

Interestingly, the problem of finding a capacity-optimal pairing given a set of non-pristine physical pages can be solved optimally in polynomial time. Essentially, the problem maps to a known *maximum matching* problem in the *compatibility graph* $G = (P, C)$, where P is the set of pages, and C is the edge set connecting any two pages that are compatible. A *maximum matching* is a subset $M \subseteq C$ of maximal cardinality, such that no two edges in M are adjacent. It was first shown by Edmonds that polynomial-time, optimal algorithms for this problem exist, regardless of the structure of the graphs [5]. Theoretically, we could periodically run such an optimal matching algorithm across *all* non-pristine physical pages (even those currently forming real pages). While this would guarantee that we always ran at maximum memory capacity, such an approach is impractical for several reasons.

First, optimal maximum matching algorithms are complex, and, while polynomial, rather slow. The fastest known algorithm [12] has an asymptotic running time of $O(|\sqrt{P}||C|)$, which can be as high as $O(|P|^{2.5})$. Also, this algorithm is so complex that it has taken nearly 10 years after its initial publication to prove its correctness. Second, the algorithm is neither incremental nor local. Whenever a new page is added to the pool of available pages, the algorithm might demand this page to be paired with an already paired page in order to preserve capacity-optimality. This can force existing real pages to be broken up, potentially incurring prohibitive copying overhead. Similarly, a single new bit-error in a currently paired page could trigger a chain-reaction in which many pairs are broken up and re-paired. In the worst-case, it is even possible for a single-bit failure to trigger a reaction that breaks up every single existing memory pair.

These observations motivate us to seek a simple, low overhead, and incremental pairing algorithm.

3.4.2 Low Overhead Approximate Pairing

The DRM implementation, described above, uses a much simpler algorithm that is both incremental and greedy. Whenever a page is under consideration for pairing, it is compared to the pages currently residing in the unmatched list one by one. As soon as a compatible page is found, the two pages are paired. As discussed, this algorithm can be implemented with low overhead, and it is also incremental and local by nature. On the other hand, it can be suboptimal with regard to capacity; possible pairings can be prevented. To see this, consider the following example: assume that a real page P is formed by two physical pages P_a and P_b. Now, assume that two new pages P_c and P_d are added the pool of available pages. These new pages P_c and P_d are mutually incompatible, but P_c is compatible with P_a, and P_b is compatible with P_d, respectively. The approximate algorithm will be unable to form a new real memory page (because P is already matched, and the two new pages are incompatible), whereas the optimal pairing would break the existing real page, and form two new real pages (P_a, P_b) and (P_c, P_d).

Fortunately, it can be shown that the expected difference in the number of pairs generated by the optimal and approximate algorithms is small in practice. First, observe that even in the absolute worst-case, the approximate algorithm finds at least half the optimal number of pairs. This is because a single pair in the approximate algorithm can prevent at most two pairings in the optimal solution. Moreover, in the real-world compatibility graphs we observe in our system, the difference between the approximate solution and the optimal one is much smaller. To see this, notice that if the lifetimes of bits are distributed randomly across PCM, then, assuming perfect wear-leveling, the compatibility graph forms a simple random graph $G_{n,p}$, where n is the number of physical pages with failures, and every compatibility edge exists with probability p, the probability of two pages being compatible. This probability can be computed as a function of the bit-failure rate r, $p = \left[1 - (1 - \bar{r}^9)^2\right]^{4000}$, where $\bar{r} = 1 - r$ is the probability that a random bit does not have a failure.

This random graph abstraction is useful because it was shown using methods from statistical physics that the size of the maximum matching in such graphs can be bounded from above by $|P|\left(1/2 - e^{-c}/2 + c^2 e^{-2c}/2\right)$, where $c = |C|/|P|$ is the average graph degree [32]. On the other hand, it is known that the simple greedy algorithm that picks edges randomly achieves a matching of $|P|\left(1/2 \cdot \frac{c}{c+1}\right)$ edges [4]. Comparing the two bounds, we observe two things. First, both bounds will approach the optimal value of $1/2$ very quickly as the average degree c in the graph increases. Second, there can only be a substantial difference between the two algorithms if the average degree is less than about 6 (i.e., every page is compatible with no more than a small constant number of other pages). Intuitively, this follows from the fact that for sufficiently many compatible pages, the "matching diversity" is higher, giving the algorithm more choice and opportunity to correct previously taken decisions. The algorithm used in our system does not strictly follow the above greedy-algorithm, because at any time, the pages in the unmatched list are not independent of each other (see also the discussion in Section 5.1.1). However, notice that the above bounds are robust: if we model the non-independence between unmatched pages as manifesting itself as a lower average degree in the random graph for the approximate algorithm, we see

that even with lower average degree, the optimal and approximate bounds match very closely. For example, if we assume that the average degree of unmatched pages in the approximate algorithm is only half the average degree for the optimal algorithm (a conservative assumption, see Section 5.1.1), the bounds imply that both the approximate algorithm and the optimal algorithm behave very similarly. Specifically, for 50k pages in the pool, and for 160 bit failures per page, the average degree in the compatibility graph is at least 50. But, even if we conservatively assume only half of that for our approximate algorithm due to dependencies, the expected capacity reduction is still only 2% compared to the optimum.

With regard to our work, the results are important: they show that the critical time-range in which the approximate solution could become noticeably suboptimal compared to the ideal capacity-optimal algorithm occurs only towards the very end of the life cycle—shortly before there are no compatible pages left to pair, even under capacity-optimal pairing. So long as enough compatible pages remain, the approximate solution achieves near-optimal performance. This analytical observation is supported by simulation results in Figure 5, which shows capacity under optimal and approximate pairing algorithms. The figure shows the respective matching sizes as a function of the number of failures per page. For almost the entire memory lifetime (expressed as the number of failures per page on the x-axis), both algorithms are equally good, and the approximate solution starts dropping off shortly before the optimal solution drops off as well—shortly before the lifetime of the device comes to an end even with the best possible algorithm.

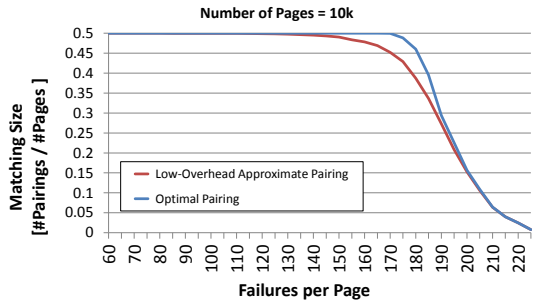

Figure 5. Size of the resulting matching when using optimal pairing algorithm compared to the low-overhead approximate algorithm.

4. EXPERIMENTAL SETUP

We conduct our experiments on a quad-core processor model with 4MB of L2 cache and an associated DDR2-800 PCM memory system. Table 1 shows the microarchitectural parameters of the processor cores we model, using a heavily modified version of the SESC simulator [23]. Table 2 shows the parameters of the shared L2 cache and the PCM memory subsystem modeled after Micron's DDR2-800 SDRAM [14], modified according to published PCM timing specifications in the literature [11]. We simulate ten memory intensive, scalable parallel applications with sixteen threads each; Table 3 lists the input sets we use. All applications are compiled using the gcc compiler at the O3 optimization level. Each application is simulated to completion.

Performing an accurate endurance comparison between conventional ECC and dynamic replication presents a methodological challenge. With a mean lifetime of 10^8 writes per cell, it takes over 24 billion writebacks from a last-level cache to exhaust half of the capacity of a 1GB DIMM, even in the absence of any wear-leveling mechanisms, and under a worst-case traffic pattern that systematically writes to a single hot spot on every page. Moreover, recently proposed, state-of-the-art PCM wear-leveling techniques come within 97% of the lifetime achieved under ideal wear-leveling [20], pushing this figure to many tens of trillions of writebacks—a figure that is clearly outside the realm of microarchitectural simulation.

In response, we adopt an alternative, conservative strategy to evaluate the endurance potential of DRM. While any imperfections in wear-leveling that generate hot spots magnify the potential endurance improvement of DRM (or any other technique attempting to mitigate wearout), we conservatively assume an idealized write pattern that systematically scans through the entire PCM address range, thereby inducing an even amount of wear on each cell in the device. We distribute cell lifetimes using a normal distribution across all cells, and we experiment with three different values of the coefficient of variation (CoV, or the ratio of the standard deviation to mean) of lifetimes, representing low (0.1), medium (0.2), and high (0.3) levels of process variability.

Processor Parameters	
Frequency	4.0 GHz
Number of cores	4
Number of HW contexts per core	4
Thread Selection Policy	Round robin
Pipeline Organization	Single-issue, in-order
iL1/dL1 size	32 kB
iL1/dL1 block size	32B/32B
iL1/dL1 round-trip latency	2/3 cycles (uncontended)
iL1/dL1 ports	1 / 1
iL1/dL1 MSHR entries	16/16
iL1/dL1 associativity	direct-mapped/4-way
Coherence protocol	MESI
Consistency model	Release consistency

Table 1. Core parameters.

Shared L2 Cache Subsystem	
Shared L2 Cache	4MB, 64B block, 8-way
L2 MSHR entries	64
L2 round-trip latency	32 cycles (uncontended)
Write buffer	64 entries
DDR2-800 PCM Subsystem [14]	
Transaction Queue	64 entries
Peak Data Rate	6.4GB/s
PCM bus frequency	400 MHz
Number of Channels	4
Number of Ranks	1 per channel
Number of Banks	4 per rank
Command Scheduler	FR-FCFS
Row Buffer Size	2KB
tRCD	22 PCM cycles
tCL	5 PCM cycles
tWL	4 PCM cycles
tCCD	4 PCM cycles
tWTR	3 PCM cycles
tWR	6 PCM cycles
tRTP	3 PCM cycles
tRP	60 PCM cycles
tRRDact	2 PCM cycles
tRRDpre	11 PCM cycles
tRAS	18 PCM cycles
tRC	22 PCM cycles
Burst Length	8
Array Read (pJ/bit)	2.47
Array Write (pJ/bit)	16.82
Buffer Read (pJ/bit)	0.93
Buffer Write (pJ/bit)	1.02

Table 2. Parameters of the shared L2 and PCM subsystem.

5. EVALUATION

We first present the endurance potential of dynamically replicated memory by comparing PCM capacity and lifetime under SECDED ECC and DRM. We then quantify the performance overhead of dynamic replication with lazy and eager replication policies at

Benchmark	Description	Problem size
Data Mining		
SCALPARC	Decision Tree	125k pts., 32 attributes
NAS OpenMP		
MG	Multigrid Solver	Class A
CG	Conjugate Gradient	Class A
SPEC OpenMP		
SWIM-OMP	Shallow water model	MinneSpec-Large
EQUAKE-OMP	Earthquake model	MinneSpec-Large
ART-OMP	Self-Organizing Map	MinneSpec-Large
Splash-2		
OCEAN	Ocean movements	514×514 ocean
FFT	Fast Fourier transform	1M points
RADIX	Integer radix sort	2M integers
CHOLESKY	Cholesky Factorization	tk29.O

Table 3. Simulated applications and their input sizes.

different points throughout the lifetime of the device. Next, we present an analysis of the performance overhead of PCM interrupt handling in the OS. Finally, we quantify the energy impact of dynamic replication.

5.1 Lifetime

Figure 6 compares the lifetime and capacity provided by SECDED ECC and dynamic replication, using a 4GB PCM device with three different CoV values. In the figure, both device capacity and lifetime are normalized to an idealized 4GB part with no process variation (CoV = 0). An eager replication policy (Section 3.2.4) is employed with dynamic replication in all three experiments.

With a CoV of 0.1, SECDED ECC starts decommissioning pages at roughly half of the lifetime of an ideal part. While the probability of any single bit having a lifetime sampled from the low-end of the tail of the lifetime distribution is very small, with a 4KB page size, a page is highly likely to have at least one bit sampled from the tail distribution. A page has to be decommissioned and its contents moved to a new location upon the first (correctable) bit failure under SECDED ECC; as a result, once pages start failing, capacity runs out quickly: half of the capacity is lost at approximately 55% of the lifetime of an ideal part, and the entire DIMM is rendered unusable at 58% of the ideal lifetime. In contrast, by dynamically recycling faulty pages, and by pairing them up on failures, dynamic replication keeps a page alive until 160 bit errors. Under dynamic replication, the device retains 50% of its capacity until 70% of the full lifetime of an ideal PCM part, and is not fully exhausted until 73% of the ideal lifetime—a 25% improvement over SECDED ECC.

As process variation increases, the benefits of using dynamic replication become more pronounced. At a CoV of 0.2, SECDED ECC decommissions half of the available capacity at 17% of the lifetime of an ideal device. Dynamic replicated memory pushes this figure to 47% of the ideal lifetime, and thus increases write endurance by a factor of 2.7 over SECDED. As CoV increases further, SECDED ECC takes a larger endurance hit, and at a CoV of 0.3, it delivers only 0.6% of the ideal lifetime. When the same device is operated under DRM, however, half of the capacity is retained until 23% of the ideal lifetime, for an endurance improvement of $40x$ over conventional SECDED ECC.

In summary, by leveraging compatibilities among faulty PCM pages, and by pairing up two compatible real pages to represent a single physical page, dynamic replication delivers dramatically better write endurance than SECDED ECC. The improvements provided by dynamic replication increase considerably as PCM scales to nanoscale feature sizes, and variations become more and more severe.

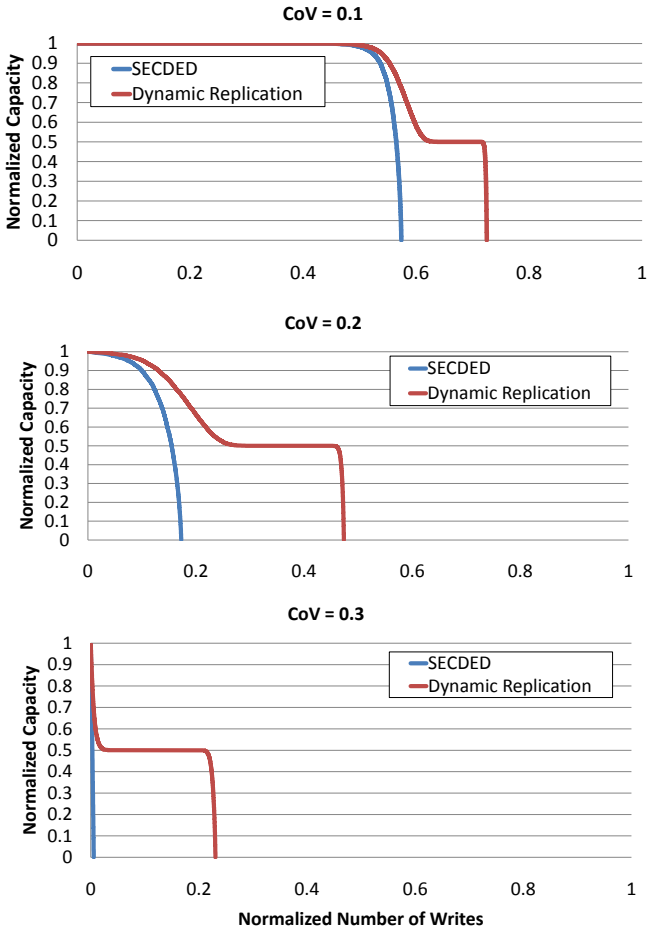

Figure 6. Endurance of SECDED ECC and dynamic replication. Results are normalized to an ideal PCM device with CoV = 0, operating under SECDED ECC.

Usage Models. These results have important ramifications on potential usage models for dynamic memory replication. Any fault detection or correction mechanism (e.g.; dual- or triple-modular redundancy, SECDED ECC) necessarily allocates a portion of the available storage space to implementing redundancy, and thus causes the protected part's product specification to be lower than the full capacity of the device; justifying these costs in a production environment requires commensurately larger gains in device lifetimes. In the case of dynamic replication, the right usage model—driven, to a large extent, by economical considerations—will likely depend on the magnitude of lifetime variability. For instance, one option would be to spec any parts built in a process with extreme lifetime variation (CoV = 0.3) at 50% of the maximum capacity in return for much longer lifetimes. Conversely, parts manufactured in a lower-variability process could be specced at a higher capacity, and dynamic replication could be selectively activated under OS control at the onset of failures beyond the spec. Flash-based SSDs employ a similar model for bad block management [27], where a set of blocks is reserved to replace blocks that fail prematurely to extend device lifetimes.

5.1.1 Ready Table Capacity

A secondary issue regarding lifetime is ready table capacity. Recall that the ready table acts a buffer between the memory controller and the operating system to ensure that the memory controller

has enough valid page pairs to handle incoming failures. Since the memory controller cannot handle errors when the ready table entries are exhausted, we must ensure that the capacity of the ready table is large enough to make such an event extremely unlikely.

To bound the probability of this event occurring, we consider intervals of n seconds in our experiments above. The ready table holds 128 entries, so we would like to show that the memory controller is unlikely to use more than 64 entries in an interval and that the operating system is likely to refill at least 64 entries in an interval. Thus, during each interval, the operating system will be capable to refilling the entries used by the memory controller in the previous interval, while having enough capacity to tolerate the memory controller's usage in the current interval.

First, to show that the memory controller is unlikely to use more than 64 entries in a second, we measured the rate at which failures occurred during our experiments. Although the failure rate for the first 75% of the device lifetime was less than 10^{-8} failures per write, it climbs to as much as 6.4×10^{-7} at the end of the device's lifetime. At 800 MB/s, we perform 1.25×10^7 writes per second. We can bound the probability of seeing 64 or more failures by computing the number of ways to choose 64 of these writes times the probability that all 64 writes fail, or $\binom{1.25\times10^7 n}{64}(6.4\times10^{-7})^{64}$. For an interval of 1.6 seconds, this result is approximately 1.6×10^{-17}. Thus, even with a device lifetime of 2.35×10^8 seconds, or 7.4 years, the chances of seeing even a single case where the memory controller consumes more than 64 entries in a second is extremely low.

Second, to show that the operating system can refill at least 64 entries per second, we look at the operating system's refill algorithm. The bottleneck for this algorithm is the hardware primitive that checks two pages for compatibility, which takes about 10 μs per page pair as described earlier; thus, we can perform $10^5 n$ page comparisons per interval. If we assume that these comparisons are independent and that they succeed with probability p, then we can compute directly the probability that the operating system finds fewer than 64 valid pairs in an interval, which is $\sum_{k=0}^{63}\binom{10^5 n}{k}p^k(1-p)^{10^5 n-k}$. With a 1.6 second interval and a probability of 0.001, this bound comes to 1.9×10^{-18}, similar to the first bound computed above.

Unfortunately, our page comparisons are not strictly independent; because we perform many comparisons against the same pool of candidate pages, the choice of pages to compare is affected by the results of previous comparisons. To attempt to account for these dependencies, we simulated our page-pairing algorithm at 160 failures per page (the very end of the device's lifetime). If we compared two random pages, we succeeded with probability 0.0021, but when simulating the algorithm itself, we took 875 comparisons per match, which corresponds to a 0.0011 success probability. Fortunately, this latter probability, which takes into account many of the dependencies inherent in our matching algorithm, is above the threshold for the bound computed above. In fact, if we tighten our analysis to use precisely the $1/875$ success probability and a 1.45 second interval, we improve both bounds by about two orders of magnitude.

These bounds are conservative in many respects. First, they use the worst failure rate observed in our experiments, which is dramatically higher in the final 10–15% of the device's lifetime than it is for the beginning and middle of the device's life. Second, we assume the maximum number of bit failures allowed by our system, which also has a dramatic impact on the probability of forming pairs. And third, a failure to consume fewer than 64 entries or to refill more than 64 entries in an interval is not itself a fatal error; these events must occur frequently and in close proximity for a true failure to occur.

Overall, this analysis shows that the chance of either using more than half of the ready table or failing to refill more than half of the ready table in any given interval is extremely low; as a result, the chances of running out of ready table entries at some point during the device's lifetime is negligible.

5.2 Performance

Figure 7 compares the performance potential of lazy and eager replication policies at four different points along the lifetime of a device. In each case, the performance of both replication techniques are normalized to an ideal PCM device with no variability, operating under no failures.

Interestingly, early on in the lifetime of the device (1-50 failures), eager replication delivers a 22% performance improvement over the baseline, ideal PCM model. As we explained in Section 3.2.4, the primary cause of the speedup is a significantly better load balance across PCM banks. On a read, eager replication generates requests for both the primary and the backup blocks; as soon as one of the two requests provides a fault-free copy of the data, both requests are deemed complete, and are removed from the PCM scheduler's queue. As a result, the least loaded of the two PCM banks can satisfy the entire request, leading to a much better load balance (and thus, performance) than the baseline, unreplicated system.

As more and more failures accrue to each page, the probability of a single access satisfying a cache-block sized read request decreases appreciably. As a result, eager replication's performance falls below unreplicated PCM. However, even with 100 failures per page, eager replication delivers nearly 90% of the performance of an unreplicated system. Lazy replication is systematically inferior to eager replication as the serialization of accesses to primary and backup blocks effectively doubles read latency.

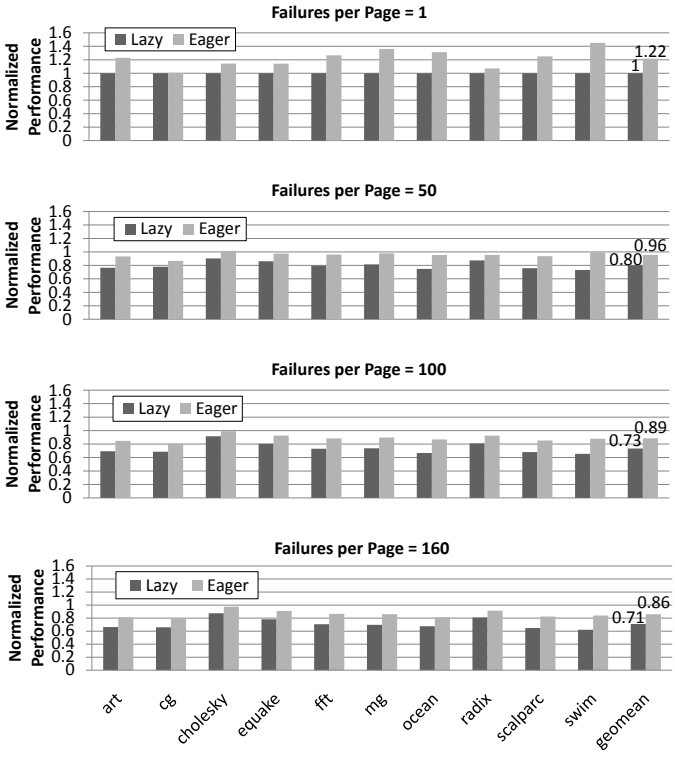

Figure 7. Performance of lazy and eager replication policies. Results are normalized to an ideal PCM device with CoV = 0, operating under no failures.

5.2.1 PCM Interrupt Handling

The main source of overhead for the operating system is the refill thread, which must respond to memory controller interrupts by generating new page pairs for the ready table. Adding and removing pages from the operating system's incoming and unmatched page lists has negligible cost per ready table entry, since only a few writes to memory are involved. However, a more significant source of overhead is the comparison operation which determines whether two pages are compatible. Near the end of the device's lifetime, when very few pages are compatible, this cost can become significant.

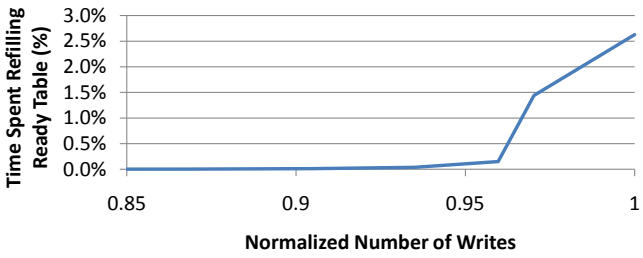

Figure 8. Expected percentage of total CPU time consumed by refilling the ready table as a function of device age.

To determine the overhead for ready table refilling, we simulated our algorithm at several points in the device's lifetime, up to the maximum of 160 failures per page allowed by our scheme. At each point, we measured the average number of comparisons needed to form a pair over 100,000 pairings, assuming that failures on incoming pages are distributed randomly. Since each comparison takes approximately 10 μs, we can use the failure rate and simulation results at each point in the device's lifetime to determine the fraction of each second that the operating system must devote to refilling the ready table.

Figure 8 shows the results from this analysis. Due to a combination of a low failure rate and a high compatibility probability, the time spent refilling the ready table is negligible (about one comparison, or 10 μs, per second) for the first 95% of the device's lifetime. For the last 5% of its lifetime, this cost jumps as high as 2.6% of overall CPU time; however, even in this state, the vast majority of CPU cycles can still be devoted to doing useful work.

5.3 Energy

Lee et al. [11] estimate a DDR2-800 compatible PCM part at the 65nm technology node to expend 16.82pJ/bit on an array write. In the worst case, assuming a sustained peak write bandwidth of 800MB/sec with 100% scheduling efficiency, this corresponds to less than 0.11W of power consumption for a 4GB PCM part. While this represents a negligible portion of the overall system power in most desktop, server, or mobile computing platforms, we would nevertheless like to understand the impact of dynamic replication on PCM energy, and we would like to confirm that the overall energy consumption under dynamic replication is still negligibly small.

Figure 9 shows the energy overhead of dynamic replication with eager and lazy replication schemes, normalized to a baseline unreplicated PCM subsystem. As expected, eager replication does result in higher energy overheads than lazy replication, but the overheads are never more than $2x$ over an unreplicated scheme. Given that the base power level of the device is well under 1W, we expect the additional energy overhead of dynamic replication to have a negligible impact on overall system power consumption.

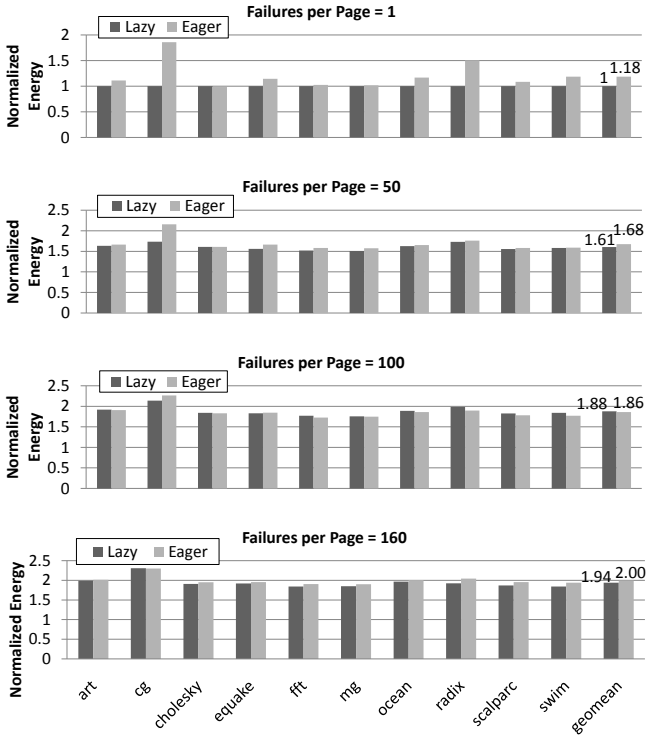

Figure 9. Energy consumption of lazy and eager replication policies. Results are normalized to an ideal PCM device with CoV = 0, operating under no failures.

6. RELATED WORK

Lee et al. [11] present area-neutral architectural techniques to make the performance and energy of DDR-compatible PCM competitive with a baseline DRAM subsystem, demonstrating that buffer reorganizations can bring PCM delay and energy within $1.2x$ and $1x$ of DRAM, respectively. Qureshi et al. [21] propose a hybrid main memory system comprising a PCM device and a tightly coupled DRAM buffer. The proposed system reduces page faults by $5x$ and improves performance by $3x$ compared to a baseline, DRAM-only system. Later, Qureshi et al. [20] propose start-gap wear-leveling, a low-overhead hardware wear-leveling technique that comes within 97% of the lifetime of an ideal wear-leveling scheme. Zhou et al. [33] propose a durable and energy-efficient main memory using PCM; the proposed wear-leveling and partial write techniques extend PCM endurance by 13–$22x$ on average. Wu et al. [29] present a hybrid cache architecture comprising disparate memory technologies, seeking to take advantage of the best characteristics that EDRAM, MRAM, and PCM offer. Zhang and Li [30] explore cross-layer techniques to mitigate the impact of process variations on PCM-based memory systems; the proposed mechanisms adjust PCM programming currents based on device variability characteristics, and apply data-comparison-write with memory compression to reduce the number of bit-flips on cells that suffer from process variations. These approaches are orthogonal to our proposal as none of this earlier work on PCM discusses the necessary support for error handling, detection, or recovery; and none of this earlier work explores the possibility of reclaiming faulty, decommissioned memory space.

Roberts et al. [24, 25] and Mudge et al. [16] propose to group faulty blocks in a last-level SRAM cache to construct a new, functional logical block; the proposed block grouping technique recovers more space and delivers lower miss rates than single- or double-error correction schemes. Unlike DRM, block grouping assumes

a fault model where the location of faulty cells can be identified at system start-up or can be hard-wired during manufacturing test, and where no new failures occur at runtime. Consequently, dynamic detection of new faults or recovery are not considered, and runtime mechanisms for breaking pairs and forming new ones are not discussed. Unlike DRM, the proposed scheme reads or writes both replicas on every access.

7. CONCLUSIONS

In this paper, we have presented a technique for dynamically pairing PCM pages in order to make one usable page from two faulty and otherwise unusable pages. We have shown that PCM devices require the memory controller to issue a read after each write in order to determine whether failures have occurred, and we have presented a design for hardware and software that can recover from any PCM faults that are detected. Our approach is complementary to wear leveling; whereas wear leveling attempts to distribute application writes evenly, our technique allows use to reuse pages that encounter a small number of early faults.

We believe that this technique is part of a larger trend toward building reliable systems from redundant, unreliable devices. As process technology scales, the variation in memory endurance is likely to become more pronounced, making such techniques essential.

8. ACKNOWLEDGMENTS

The authors would like to thank Sean Eilert for his help in developing a realistic PCM fault model, and Jim Goodman for many valuable discussions on fault tolerance in resistive memory systems.

References

[1] Y. Azar, A. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *Symposium on Theory of Computing*, May 1994.

[2] J. Condit, E. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *International Sympoisum on Operating System Principles*, October 2009.

[3] E. Doller. Phase change memory, September 2009. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[4] M. Dyer, A. Frieze, and B. Pittel. The average performance of the greedy matching algorithm. *Annals of Applied Probability*, 3(2), 1993.

[5] J. Edmonds. Path, trees and flowers. *Can. J. Math*, 17, 1965.

[6] S. Eilert. PCM fault models, November 2009. Private communication with Sean Eilert, Director of Architecture Pathfinding at Numonyx.

[7] B. Gleixner, F. Pellizzer, and R. Bez. Reliability characterization of phase change memory. In *European Phase Change and Ovonics Symposium*, September 2009.

[8] International technology roadmap for semiconductors. *Process integration, devices, and structures*, 2009.

[9] C. Kim, D. Kang, T.-Y. Lee, K. H. P. Kim, Y.-S. Kang, J. Lee, S.-W. Nam, K.-B. Kim, and Y. Khang. Direct evidence of phase separation in Ge2Sb2Te5 in phase change memory devices. *Applied Physics Letters*, 94(10):5–5, May 2009.

[10] K. Kim and S. J. Ahn. Reliability investigations for manufacturable high density pram. In *IEEE International Reliability Physics Symposium*, April 2005.

[11] B. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase-change memory as a scalable dram alternative. In *International Symposium on Computer Architecture*, June 2009.

[12] S. Micali and V. V. Vazirani. An $O(\sqrt{|V||E|})$ algorithm for finding maximum matching in general graphs. In *FOCS*, 1980.

[13] R. Micheloni, A. Marelli, and R. Ravasio. In *Error Correction Codes for Non-Volatile Memories*, 2008.

[14] Micron. *512Mb DDR2 SDRAM Component Data Sheet: MT47H128 M4B6-25*, March 2006. http://download.micron.com/pdf/datasheets/dram/ddr2/512MbDDR2.pdf.

[15] M. D. Mitzenmacher. The power of two choices in randomized load balancing. *Doctoral Dissertaion, Graduate Division of the University of California at Berkeley*, 1996.

[16] T. N. Mudge, G. S. Dasika, and D. A. Roberts. Storage of data in data stores having some faulty storage locations, March 2008. United States Patent Application 20080077824.

[17] Numonyx. The basics of PCM technology, September 2008. http://www.numonyx.com/Documents/WhitePapers.

[18] Numonyx. Phase change memory: A new memory technology to enable new memory usage models, September 2009. http://www.numonyx.com/Documents/WhitePapers.

[19] A. Pirovano, A. Radaelli, F. Pellizzer, F. Ottogalli, M. Tosi, D. Ielmini, A. L. Lacaita, and R. Bez. Reliability study of phase-change nonvolatile memories. *IEEE Transactions on Device and Materials Reliability*, 4(3):422–427, September 2004.

[20] M. K. Qureshi, M. Fraceschini, V. Srinivasan, L. Lastras, B. Abali, and J. Karidis. Enhancing lifetime and security of phase change memories via start-gap wear leveling. In *International Symposium on Microarchitecture*, November 2009.

[21] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *International Symposium on Computer Architecture*, June 2009.

[22] S. Raoux, D. M. Ritchiea, K. Thompsona, D. M. Ritchiea, K. Thompsona, D. M. Ritchiea, and K. Thompson. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(7):5–5, July 2008.

[23] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos˙ SESC simulator, January 2005. http://sesc.sourceforge.net.

[24] D. Roberts, N. S. Kim, and T. Mudge. On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology. In *Euromicro Conference on Digital System Design*, August 2007.

[25] D. Roberts, N. S. Kim, and T. Mudge. On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology. *Elsevier Microprocessors and Microsystems*, 2008.

[26] J. Rodgers, J. Maimon, T. Storey, D. Lee, M. Graziano, L. Rockett, and K. Hunt. A 4-mb non-volatile chalcogenide random access memory designed for space applications: Project status update. In *IEEE Non-Volatile Memory Technology Symposium*, November 2008.

[27] Samsung. Bad-block management, September 2009. http://www.samsung.com/global/business/semiconductor/products/flash/downloads/xsr_v15_badblockmgmt_application_note.pdf

[28] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. Cacti 4.0. Technical report, HP Labs, 2006.

[29] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Hybrid cache architecture with disparate memory technologies. In *International Symposium on Computer Architecture*, June 2009.

[30] W. Zhang and T. Li. Characterizing and mitigating the impact of process variations on phase change based memory systems. In *International Symposium on Microarchitecture*, September 2009.

[31] W. Zhang and T. Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2009.

[32] H. Zhou and Z.-C. Ou-Yang. Maximum Matching on Random Graphs. *Europhsics Letters – Preprint*, 2003.

[33] P. Zhouand, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *International Symposium on Computer Architecture*, June 2009.