RePRAM: Re-cycling PRAM Faulty Blocks for Extended Lifetime

Jie Chen, Guru Venkataramani, H. Howie Huang
Department of Electrical and Computer Engineering,
The George Washington University, Washington DC, USA
{*jiec,guruv,howie*}@*gwu.edu*

***Abstract*—As main memory systems begin to face the scaling challenges from DRAM technology, future computer systems need to adapt to the emerging memory technologies like Phase-Change Memory (PCM or PRAM). While these newer technologies offer advantages such as storage density, non-volatility, and low energy consumption, they are constrained by limited write endurance that becomes more pronounced with process variation. In this paper, we propose a novel PRAM-based main memory system, RePRAM (Recycling PRAM), which leverages a group of faulty pages and recycles them in a managed way to significantly extend the PRAM lifetime while minimizing the performance impact. In particular, we explore two different dimensions of dynamic redundancy levels and group sizes, and design low-cost hardware and software support for RePRAM. Our proposed scheme involves minimal hardware modifications (that have less than 1% on-chip and off-chip area overheads). Also, our schemes can improve the PRAM lifetime by up to 43× (times) over a chip with no error correction capabilities, and outperform prior schemes such as DRM and ECP at a small fraction of the hardware cost. The performance overhead resulting from our scheme is less than 7% on average across 21 applications from SPEC2006, Splash-2, and PARSEC benchmark suites.**

***Keywords*-Phase Change Memory, Lifetime, Redundancy, Main memory, Performance**

I. Introduction

Modern processor trends toward multi-core systems exert increasing pressure on DRAM systems to retain the working sets of all threads executing on the individual cores. This has forced greater demands for main memory capacity and density in order for the computer systems to keep up with performance scalability while operating under limited power budgets. As a result, it becomes necessary to explore alternative emerging memory technologies such as Flash and resistive-memory types such as Phase Change Memory (PCM or PRAM) to reduce the overall system cost and increase the storage density.

PCM, in particular, has been shown to exhibit enormous potential as a viable alternative to DRAM systems because it can offer up to 4× more density at only small orders of magnitude (up to 4×) slowdown in performance [22]. PCM is made of chalcogenide glass, which has crystalline and amorphous states corresponding to low (binary 1) and high (binary 0) resistances to electric currents. There have been recent proposals that look at using PCM-based hybrid memory for future generation main memory [21].

A major challenge when using PCM as a DRAM replacement arises from its limited write endurance. PCM-based devices are expected to sustain an average of 10^8 writes per cell, when the cell's programming element breaks and the write operations can no longer change the values. Most of the existing solutions focus on wear-leveling [20] and reducing the number of writes to PCM [14], [34]. Some recent studies have looked at resuscitating the faulty pages that were normally discarded as unusable by the memory controller [5], [9], [24], [26], [32]. In this paper, we adopt a similar goal of extending lifetime of PCM beyond the initial bit failures.

Our objective behind rejuvenating faulty PCM blocks is to put faulty blocks (that would otherwise be discarded by the memory controller) back into use, i.e., not having to retire them prematurely. As PCM-based memory begins to find widespread acceptance in the market, memory manufacturers will need to take system engineering costs into account. Frequent replacement of failed memory modules can be cost prohibitive, especially for large scale data centers.

In this paper, we propose that instead of completely discarding a PCM page as soon as it becomes faulty, a group of faulty pages could be recycled and utilized in a managed way to significantly extend the PCM lifetime while minimizing the performance impact. To this end, we design RePRAM (Recycling PRAM), that explores advanced dynamic redundancy techniques, while minimizing the complexity of finding compatible faulty pages to store redundant information. We explore varying levels of redundancy as bits begin to fail. First, we add a redundant (parity) bit for a given number of data bits. We call it as Parity-based Dynamic Redundancy (PDR). As the number of faults per page increase, we add more redundancy where we have one redundant (mirror) bit for every data bit. We call this as Mirroring-based Dynamic Redundancy (MDR).

We explore the redundancy techniques along two dimensions, redundancy levels and group sizes (defined as the number of faulty PCM pages that form a group together for parity). In the first dimension, we investigate using different redundancy levels to get extra lifetime in PRAM. We begin with PDR, and as the number of faults per page begin to rise, we switch to MDR to minimize the complexity associated with finding compatible faulty pages. For the second dimension, we explore varying group sizes

within PDR. Specifically, we investigate the design trade-offs by adopting two and three pages for a PDR group, and the resulting effects on storage efficiency, performance, and hardware complexity in the system. While exploring redundancy, we follow two rules: (1) Any two PCM pages are deemed compatible with each other only when the corresponding pages do not have faulty bytes in the same byte position. (2) The PCM pages are discarded once they have at least 160 faults, because finding compatible pairs of pages becomes exponentially harder beyond this limit [9].

Our main motivation behind exploring dynamic redundancy-based techniques to improving the lifetime of PCM is driven by three factors:

- *Increase the space efficiency in the usage of faulty pages*: In MDR configuration, we mirror identical data on two compatible faulty PCM pages that effectively replicates data across both pages. As a result, both the PCM pages together store a single page worth of data, i.e., the storage density is 50%. In PDR configuration, a group G of n faulty pages have a dedicated block *P*, that stores the parity values for all of the n pages. Therefore, the storage density for $n+1$ (including the parity) pages is $\frac{n}{n+1}$. For example, with group size of 3, the storage efficiency is 75% (25% more efficient than MDR), and for group size of 2, the storage efficiency is 67% (17% higher than MDR). At higher values of n, we get better efficiency in terms of storage density, although finding compatible pages for larger groups become increasingly difficult.
- *Utilize off-the-shelf memory components without extensive hardware redesign*: Prior techniques such as ECP [24] have to custom design the PCM chip to integrate their lifetime-enhancing techniques. Such designs increase the hardware cost, and more importantly, reduce the flexibility of switching to other lifetime-enhancing techniques in the future. To counter such drawbacks, our goal is to maximize the use of off-the-shelf components and include techniques that would enhance lifetime with minimum changes to hardware design.
- *Explore design choices that will offer flexibility to the user*: The end-users can make an informed choice that is most suitable to their needs under a given cost budget.

To summarize, the main contributions of RePRAM are:

1) We explore *dynamic redundancy techniques to resuscitate faulty PCM pages* and investigate the merits of different design choices toward improving the lifetime of PCM-based Main Memory systems.
2) We propose low-cost hardware that can be combined with off-the-shelf PCM memory and show that only *minimal hardware modifications* are needed to implement RePRAM schemes. We also study how relatively small DRAM buffers can be used to store parity pages and reduce the performance impact.
3) We evaluate our design and show that we can improve the PCM lifetime by up to 43× over raw PCM without any Error Correction Capabilities at less than 1% area overhead both on-chip and off-chip. Also, we incur less than 7% performance overhead (average case), and less than 16% performance overhead (stress case) across SPEC2006 [27], PARSEC-1.0 [1] and Splash-2 [30] applications.

II. Background and Related Work

In this section, we present a brief overview of PCM main memory or PRAM. We discuss prior research that have studied extending PCM lifetime of PCM, and give a quick review of redundancy based techniques to tolerate faults.

A. PCM based Main Memory and Wear-out Problem

DRAM, which has served as computer system main memory for decades, is confronting scalability problems as technology limitations prevent scaling cell feature sizes beyond 32nm [18]. DRAM is also confronting power-related issues due to high leakage caused by shrinking transistor sizes. All of these have led to building main memory with alternative emerging memory technologies such as Phase-change Random Access Memory (PRAM).

An important consideration is PCM's high operating temperatures for set and reset operations that directly affect the lifetime of PRAM devices. In particular, repeated reset operations at very high temperatures cause to break the programming circuit of phase change material, and permanently reset the PCM cell into a state of high resistance. This introduces limited endurance to PCM that significantly restrains the use of PCM as a good replacement for DRAM-based memory. Additional complications arise in PCM due to process variation effects that could further decrease write endurance in these devices.

B. Extending PCM Lifetime

To mitigate the effects of PCM's limited endurance, prior works have looked at better wear-leveling algorithms [20], reducing write traffic to PCM memory through partial writes [14], writing select bits [6], [33], randomizing data placement [25], exploring intelligent writes [4], and using DRAM buffers [21]. All these techniques focus on applying their optimizations prior to the first bit failure. We note that these techniques are complementary to our RePRAM, and can contribute to even higher PCM lifetime when used in conjunction with our techniques.

To the best of our knowledge, the first scheme to look at reusing faulty pages was Dynamically Replicated Memory (DRM) [9]. DRM forms pairs of faulty PCM pages that do not have faults in the same byte position so that paired pages can serve as replicas of one another. Redundant data

is stored in both pages to make sure that the system can read a non-faulty version of the byte from at least one of the pages. The idea behind pairing is that there is a high probability of finding two compatible faulty pages and hence, one could eventually reclaim what would otherwise be a decommissioned memory space. While this is useful, simply replicating the data in both pages can rapidly degrade the effective capacity of the memory system. We note that, by using replication scheme, DRM merely explores MDR or mirroring. First of all, by using MDR for PCM pages that have too few errors initially, DRM can waste a lot of non-faulty bytes in the paired pages and unnecessarily replicate the entire block. Secondly, writes need to update both the mirror copies, which means that both pages will endure increased wear, and subsequently result in expedited aging of the pages contributing to their failures. In RePRAM, we overcome the above disadvantages by exploring more dynamic redundancy techniques like PDR, that reduce unnecessary additional writes to PCM blocks. Further, we combine several PDR configurations to explore design points that will be most suitable to user's needs.

Error Correcting Pointers (ECP) [24] is another technique that handles the errors by encoding the locations of failed cells in a table and by assigning new cells to replace them. The main disadvantage with this approach is the complexity of changing PCM chip to accommodate the dedicated pointers, as well as, the costs involved in hardware redesign. ECP incurs a static storage overhead of 12% to store these pointers. SAFER [26] dynamically partitions the blocks into multiple groups, exploiting the fact that failed cells can still be read and reuses the cell to store data. LLS [10] is a line-level mapping technique that dynamically partitions the overall PCM space into a main space and a backup space. By mapping failed lines to the backup, LLS manages to maintain a contiguous memory space that provides easy integration with wear leveling. When accessing a failed line, the request will be redirected to access the mapped backup line through a special address translation logic, which requires PCM chip redesign efforts and also incurs extra latency and energy. LLS also relies on intra-line salvaging, such as ECP, to correct initial cell failures. RDIS [15] incorporates a recursive error correction scheme in hardware to track memory cell locations that have faults during write. In RePRAM, we use off-the-shelf PCM storage and perform minimal changes to existing hardware, which lowers the cost of redesign and helps us to minimize the performance overheads.

FREE-p [32] performs fine-grained remapping of worn-out PCM blocks without requiring dedicated storage for storing these mappings. Pointers to remapped blocks are stored within the memory block and the memory controller adds extra requests for memory to read these blocks, which results in additional bandwidth and latency overheads. As the number of faults per block increase, this approach incurs higher performance overheads due to sequential memory reads and increased memory bandwidth demands. Whereas, RePRAM incurs significantly lower performance overheads as the memory requests to group blocks can be overlapped and performed simultaneously (as shown in Section IV).

C. Redundancy-based Techniques

Storing redundant data to achieve high availability was explored by Patterson, Gibson, and Katz for Redundant Arrays of Inexpensive Disks (RAID) [17] in 1987, where the original five RAID levels were presented as a low-cost storage system. Since then, RAID has become multi-billion dollar industry [12], and inspired many similar concepts such as Redundant Arrays of Independent Memory (RAIM) [8], Redundant Arrays of Independent Filesystems (RAIF) [11], and so on. While our work leverages a number of redundancy techniques, we actually aim to reuse faulty PCM pages, whereas RAID was designed to recover the data on a failed hard disk, which statistically has a higher availability compared to average 10^8 writes per PCM cell. In the context of PCM-based memory, we face a set of new problems, such as mapping management, asymmetric read/write performance, and limited write cycles. On the very high level, our idea has similar spirit as HP AutoRAID hierarchical storage system [29], where RAID 1 and 5 are deployed at two different levels, with the former used for high performance data access, and the latter for high storage efficiency. In AutoRAID, the decision is made upon active monitoring of workload access patterns and data blocks are migrated automatically in the background. In contrast, our RePRAM employs a unique redundancy level at different stages of the PCM lifetime, which is selected to maximize its usability and performance.

III. Design Overview

In this section, we first describe the overview of our hardware design and later show how our proposed modifications can be incorporated into the multi-core processor hardware. We also show the software support needed for RePRAM.

A. Incorporating Dynamic Redundancy Techniques into PCM

In this paper, we assume that the PCM-based main memory starts without any faults and has wear-leveling algorithms in place to uniformly distribute the writes throughout the pages. For each write to the page, a read-after-write is performed to determine if the write succeeded. For cases where Error Correcting Code (ECC) exists onchip, the first few bit faults can be tolerated using these pre-built schemes. In particular, SECDED (single-error correction, double-error detection) ECC can correct one bit error, while Hi-ECC [28] can correct up to four bit errors, both of which can be utilized to tolerate errors before beginning to use our RePRAM schemes. Note that this does not require

RePRAM intervention, and thus for the first few faults, avoids performance impact on the applications. After the point when the error correction capabilities can not tolerate any more faults, the PCM-based main memory that did not use any additional lifetime extension strategies, would be forced to discard the faulty pages. Subsequently, this would lead to a quickly diminishing capacity of the memory. To maximize PCM lifetime with minimal overheads and low cost, we propose RePRAM that recycles and leverages advanced dynamic redundancy techniques in the context of the PCM-based main memory. We devise a number of hardware (and supporting software) techniques to realize the full benefit of this new memory architecture.

In RePRAM, when the first fault in the PCM page k occurs (beyond the tolerance limit of already incorporated error correction schemes), we temporarily decommission this page and place it in a separate pool of PCM pages $POOL_{PDR}$, where the faulty pages are waiting to be matched with other compatible faulty pages. At this point, we disable the ECC computation, an operation supported by most ECC-based memory controllers [19]. We reuse the parity bits to store the faulty byte vector within the already built-in parity support, i.e., for each byte we have a bit to indicate whether it is faulty or not. We start with the redundancy level of PDR, in which a dedicated parity block is associated with a group of data pages. Basically, we group multiple faulty PCM pages that are compatible (i.e., do not have faults in the same byte position) and use a separate high-performance DRAM buffer to store the corresponding parity to minimize the performance impact. The number of data pages per group (n) is determined based on the matching complexity. Note that for higher values of n, it becomes exceedingly difficult to find compatible faulty pages.

When we decommission the PCM page k, we copy its contents into one of the reserve PCM pages, which serve to store the data in the faulty pages until the matching is done. By default, we assume that there are 10,000 such reserve PCM pages. To start, we randomly pick a group G of compatible faulty pages from $POOL_{PDR}$. Alternatively, a low-cost approximate pairing algorithm similar to one in DRM [9] can be used to assemble the group G of compatible faulty pages from $POOL_{PDR}$. The mapping information is stored as tuples $MAP_{PDR} = \{k_1, k_2, ..., k_n, P\}$, where n is the group size and P is the parity page, which are managed by the Operating System (OS) (described in III-C). For example, in a PDR group size of three, a tuple of the form, $\{k_1, k_2, k_3, P\}$ is stored, where k_1, k_2, k_3 are the compatible faulty pages and P is the parity page. In the remainder of this paper, we use a default group size of three for PDR configuration as it is relatively less complex than higher order matching, while offering much better data to parity storage ratio than PDR with the group size of two.

We note that matching process (finding compatibility) is much easier with pages that have fewer number of faults. This is especially true when using PDR, as the cost associated with finding the three-way compatible pages beyond a certain number of bit faults increases sharply. To empirically evaluate the mapping process, we simulate the trials needed for 2-way and 3-way matching under PDR. Figure 1 presents the average number of the trails in both cases. These experiments were done by assuming a pool of one million 4KB pages and we averaged over 10,000 samples. We also tried a pool of 10,000 4KB pages with over 1,000 samples and observed similar results. In the two-way matching, we randomly pick two faulty pages and check if they are compatible. If not, we repeat with another new randomly picked page until we find a compatible set of pages. The average number of tries needed is recorded. We repeat our experiments by bounding our faults in various ranges as shown in Figure 1. The same set of experiments was done for the three-way matching for compatibility of pages. Beyond 80 faults (we refer to this as *three-way fault threshold*), we notice that the number of trials needed for three-way matching is at least twice as the number of the two-way trials and steeply increases from there. Therefore, continuing to operate in PDR with group size of three beyond the *three-way fault threshold* will be expensive for matching algorithm that forms groups of compatible faulty pages.

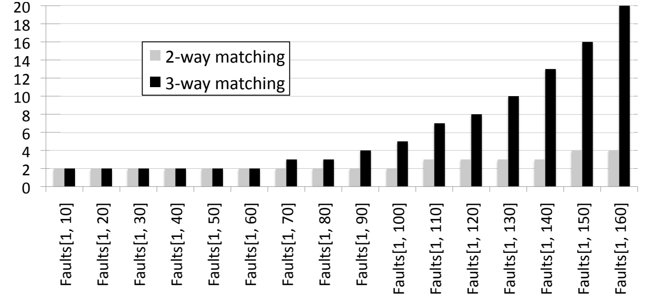
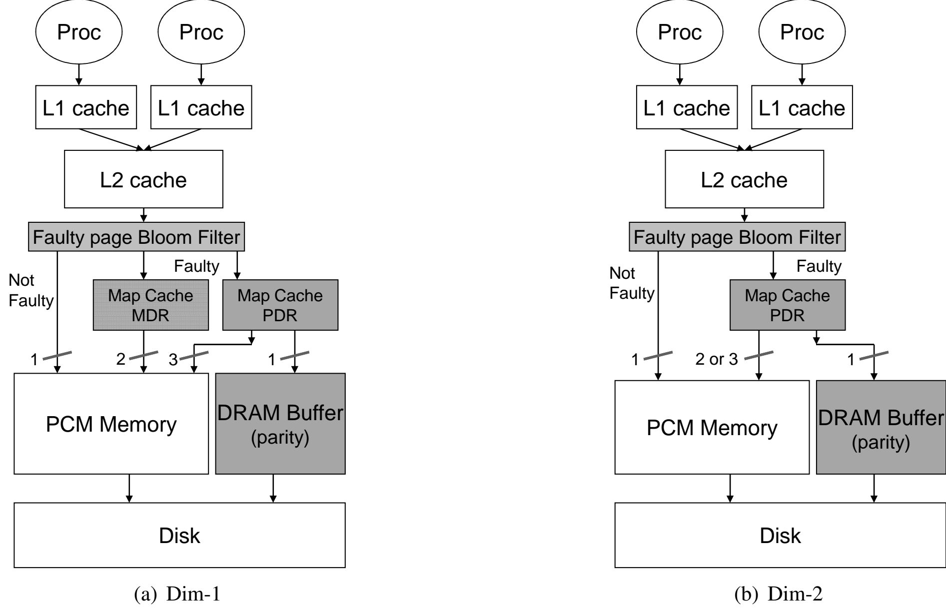


Figure 1. Average number of random trials needed for the two-way and three-way matching between faulty pages. Each pair of bars show the number of tries needed for matching within the fault bounds indicated in the x-axis. Faults are assumed to be randomly distributed within the 4 KB PCM pages.

In order to bound the complexity associated with matching, we explore two design dimensions after a faulty page incurs the number of errors that is greater than the three-way fault threshold:

1) Dim-1: Switch to MDR that employs the mirroring-based dynamic redundancy technique, where we store two identical copies of the data onto two different PCM pages.
2) Dim-2: Reduce the PDR group size to two and continue to operate under parity-based dynamic redundancy mode to tolerate future faults.

We choose to explore the above two dimensions primarily to provide more user-friendly options and allow the users to



(a) Dim-1

(b) Dim-2

Figure 2. Hardware modifications and structures (shown in gray boxes) needed to incorporate RePRAM into the processor hardware. The number of memory requests issued to the memory structure is shown next to the arrows (e.g., PDR always issues 3 PCM requests in Dim-1 for a group size of 3; whereas in Dim-2, the number of requests (2 or 3) depends on the current group size).

decide what is for their best interests. Our experiments quantify the resulting lifetime and the associated performance impact. We note that the end users should be able to choose the configuration that suits their needs.

Dim-1: We switch to MDR mode when one of the newly occurring fault in an already matched PCM page (in PDR) surpasses the three-way fault threshold and renders the group pages incompatible. This new fault will be detected by the read-after-write operations in PCM-based memory [31]. When this happens, we reassign the faulty PCM page to a new pool $POOL_{MDR}$, where pages are waiting to be matched with other compatible faulty pages in the mirroring fashion. We use the MDR technique to tolerate bit faults for the remaining portion of the pages' lifetime and their corresponding mappings are stored in MAP_{MDR}, managed by the OS. Similarly, the mappings can be represented as tuples of $\{k_1, k_2\}$ where k_1 and k_2 are compatible faulty pages that mirror the content of each other. It is worthy to point out that even after MDR mapping is adopted for some PCM pages, other pages that are currently in MAP_{PDR} will continue to be in PDR mode until there is a need for remapping these faulty pages.

Dim-2: In this case, throughout the lifetime of the device, we continue to operate in the PDR mode where a group of PCM pages have an associated parity page. The only distinction is when a newly occurring fault surpasses the three-way fault threshold and renders the already existing group incompatible, where we downgrade the group size to two. This results in the need for configuring the memory controller to monitor for the group sizes corresponding to faulty PCM page and MAP_{PDR} tables to handle the relevant group size information.

On a read to a main memory page k_1, we first determine whether the target page is faulty or not. If k_1 is not faulty, the memory request proceeds as usual. If it is faulty, we determine whether k_1 is in MAP_{PDR} or in MAP_{MDR}. The accesses to MAP_{PDR} and MAP_{MDR} can be performed in parallel. As the result, we obtain the corresponding information on this group of blocks k_2, k_3, and P for PDR, or alternatively, the information about the group block, k_2 for MDR. Extra memory requests are issued by the memory controller for the corresponding group blocks depending on the PDR/MDR mapping. Upon reading the blocks, we use the faulty byte vector (reused ECC parity bits) to determine the locations of faulty bytes. Once the memory requests are satisfied, the data block for k_1 is reconstructed based on the PDR parity or the MDR mirror. We note that these extra requests could potentially lead to a performance bottleneck by saturating the bus bandwidth and delaying the reads that are on the critical path. In order to reduce the performance overheads, additional optimizations such as a temporary read buffer that stores reconstructed memory blocks shall be used. Such optimizations can offer faster read accesses to otherwise faulty memory blocks, however, care should be taken to ensure that write accesses properly update the actual PCM and parity blocks, keeping them consistent at all times. To ensure data consistency, every write operation should make sure to invalidate or update the appropriate read

buffer entries. Since writes are off the performance-critical path, lazy invalidate or update schemes can be used for read buffer entries.

On a write to a main memory page k_1, we also determine whether the target page is faulty or not. If faulty, we find the corresponding group blocks from the mapping tables. In case of MDR, we issue two memory requests to k_1 and k_2. For PDR, we need to update only k_1 and corresponding parity, P. Therefore, the number of memory requests needed for a write under RePRAM is always two, instead of $(n+1)$ memory requests needed for a read.

B. Hardware Implementation

The primary goal for RePRAM implementation is to lower the cost and complexity of hardware design. Also, the performance impact resulting from our proposed hardware changes should not adversely affect the normal processor performance. Figure 2 shows the hardware structures that are needed to implement the RePRAM scheme in a cost-effective manner with low performance overheads.

The first task is to find out whether a page is pristine or has faults. A naive approach to detecting faulty pages is to make memory controller perform additional readback after a write to determine whether the write succeeded [9]. However, performing repeated read-after-write requests during every write operation can be time consuming and hence, we use compact structures such as Bloom Filters [2] to record information about the faulty pages. Once we determine the faulty/pristine status of the page, we can allow the pristine (fault-free) pages to directly access the memory as usual, while the faulty pages can be directed to lookup one of the MAP structures. One caveat with using structures such as Bloom Filters is that they have a potential for false positives, i.e., a page may be reported to be faulty when it actually does not have faults. Fortunately, our experiments show a negligible rate of false alarms, and usually such cases can be addressed by checking if the page has an entry in one of the MAP tables or by directly reading the faulty byte vector associated with the PCM page.

Once the pages are determined to be faulty, the next step is to check the associated mapping under MAP_{PDR} (and MAP_{MDR} too for Dim-1). This is to reconstruct the original data page for a read or update the necessary parity information on a write corresponding to the faulty PCM page. The mapping information is managed by the OS and frequent invocation of the OS for mapping lookups can result in expensive overheads. Hence, to speedup this process, we use limited-entry caches, $CACHE_{PDR}$ and $CACHE_{MDR}$ to temporarily store MAP_{PDR} and MAP_{MDR} respectively. In our current implementation, we use two small 1024-entry buffer caches to store the mappings. This organization is similar to Translation Lookaside Buffers (TLB). Upon a miss in both the mapping caches, the OS service routine is invoked and a mapping lookup is performed. An entry is created in the corresponding mapping cache depending on the dynamic redundancy scheme under which the requested page is mapped. We may remove an entry from the mapping caches for the following reasons: (1) when one of the PCM pages has suffered permanent failure (more than 160 bit errors) and needs decommissioning, or (2) when PDR to MDR remapping needs to be done due to increased matching complexity associated with PDR. Note that, under Dim-2, we would continue to remain in PDR mode, but with different group sizes. However, due to decreasing group size from three to two, previously formed groups may need to be reorganized and would require us to flush the corresponding entries from $CACHE_{PDR}$.

We use a separate DRAM buffer to store the parity for faster access by the multi-core processor, which we believe is more cost-effective than simply investing on additional PCM capacity for parity. This is mainly motivated by two facts: (1) DRAM provides fast data access and does not suffer from write endurance problem. Note that the parity information needs to be accurate and cannot have errors in order to recover the data from faulty PCM block. DRAM buffer offers a much better alternative to PCM in this regard. (2) The parity information is much smaller than data itself - for a group of n data pages, there is only one corresponding parity page. So, DRAM buffer can be a lot smaller than the actual PCM-based RAM. A caveat, however, is that parity information could be lost during power failure as DRAM buffer only offers volatile storage. To counter this problem, prior techniques such as Brant et al. [3] have proposed a low cost, battery-powered Flash RAM backup to store the contents of DRAM. As we will observe in our evaluation (Section IV), fast accesses to parity offered by DRAM buffers can easily outweigh the demerits of using DRAM buffer (that can be handled through other low-cost mechanisms). In this work, we use off-the-shelf components in our hardware design without extensive modifications to the hardware structures. We note that both the data and parity pages would share the lower level disk for backup storage.

Finally, we would need to discard pages that have more than 160 bit errors and decommission these pages for permanent failure. To do so, during rematching of pages (when pages become incompatible upon errors in the corresponding same byte position), we determine if the candidate pages have more than 160 bit errors. We permanently mark such pages for removal, and invoke the remapping algorithm to reconstitute the other group pages associated with the failed page under MDR or PDR.

C. Software Support for RePRAM

In RePRAM, the OS is responsible for managing the mapping of faulty PCM pages, as well as parity pages. The OS maintains two lists, the *free list* for DRAM parity buffer, and the *candidate list* for faulty PCM pages. In the beginning, the free list contains all the pages in the DRAM

buffer. Upon forming a new group of compatible faulty PCM pages, one entry from DRAM buffer free list is allocated for storing parity and this information is maintained in the OS. When the group is disbanded due to incompatibility among the pages, the OS will put the corresponding parity page back into the free list. When the free list is empty, the OS will stop the matching process.

The OS invokes the matching algorithm every time when the memory controller detects a new faulty page or if a page becomes incompatible with its group pages. This new faulty PCM page is inserted into the candidate list where compatible pages can be paired. The OS performs random selection from the candidate list to quickly form a group of the compatible pages. After a page is successfully matched, it will be removed from the candidate list. Note that our experiments show that not every fault result in incompatibility between the group pages (see Section IV-D). Remapping happens far less frequently than the number of faults occurring in pages, which greatly reduces the OS overhead for page remapping.

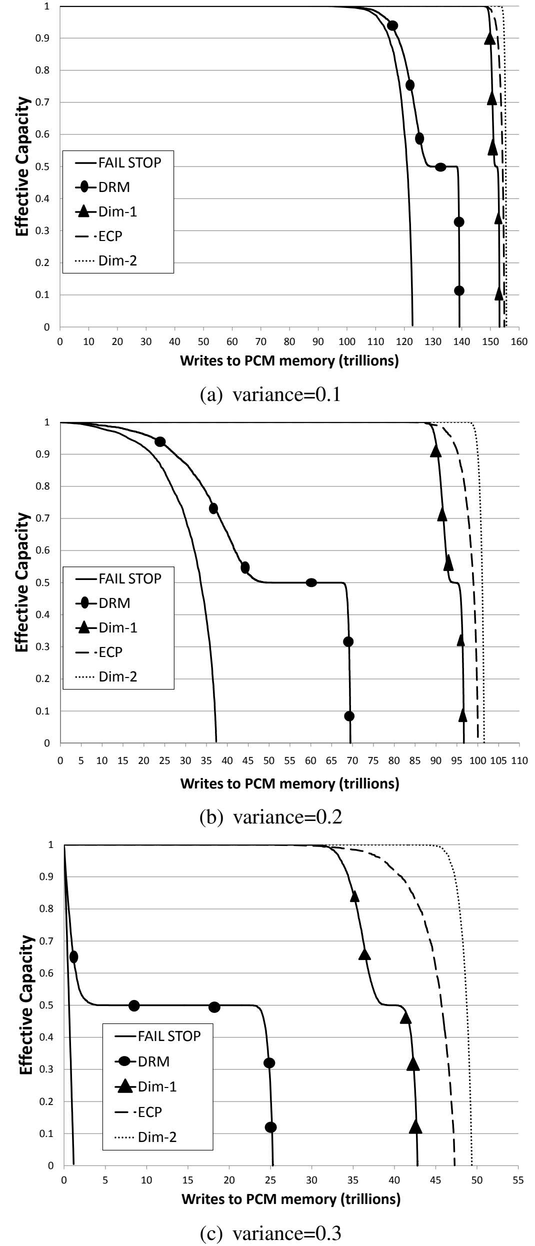
The OS manages the mapping between faulty PCM pages using a hash table that is stored in a reserved area of the memory address space. Assuming that we need to maintain the mapping information for every page, the worst case storage overhead for a 4GB PRAM with 4KB pages would only be 2.5MB (20 bits per page for 1M pages). Also, that the OS interventions for accessing and updating the mapping information should be kept minimal in order to avoid high performance overheads. As we show in Section IV, with the help of mapping caches and dynamic redundancy levels, RePRAM is able to minimize performance impacts on a wide range of the benchmarks.

IV. Evaluation

We first perform the experiments to analyze the extended lifetime that can be achieved through RePRAM, and next study the performance impact arising from our proposed hardware changes. Finally, we present the sensitivity studies that show the performance of our system under various configurations. To present fair comparison results with prior schemes, our configuration parameters are similar to, and derived from earlier works such as DRM [9] and ECP [24].

A. Lifetime

Figure 3 compares the *lifetime (measured as the total number of writes that can be performed to the PCM)* along with the diminishing effective capacity left in the PCM device. We show the results corresponding to both Dim-1 and Dim-2 design choices, and contrast them with prior schemes such as Fail_Stop (that does not have any Error Correction capabilities and discards a PCM block as soon as the first fault occurs), DRM and ECP schemes. We assume a baseline of 4GB PCM memory with 4KB page size, and write operations happen on a granularity of 64Byte blocks.



(a) variance=0.1

(b) variance=0.2

(c) variance=0.3

Figure 3. Effective capacity versus the total number of writes issued to the PCM main memory.

In addition, we assume a 50% probability that any single write operation would flip a particular bit. *We model the PCM cell lifetime to follow a normal distribution with a mean of* 10^8 *and three different process variation factors of 0.1, 0.2, and 0.3* (shown in Figure 3).

Our experiments show that with a 4KB page size, there is a high probability of at least one byte having a lifetime at the tail-end of the normal distribution. This makes the Fail_Stop scheme quickly decommission all of the pages within a short window of writes before the PCM's effective capacity drops

to zero. This effect is especially more pronounced at higher levels of process variation, as shown in our experimental results. Although DRM is able to tolerate up to 160 faults in each page before decommissioning the page for permanent failure, the main disadvantage with DRM is that replicated writes to two different pages speeds up the aging process of both pages. DRM offers an extended lifetime of approximately 0.13× to 22× over Fail_Stop scheme depending on the process variation factor. The third scheme, ECP, can tolerate up to 6 faults in each 64 Byte block and a 4KB PCM page is decommissioned as soon as the seventh fault occurs in one of its constituent 64 Byte blocks. This lets ECP achieve a lifetime improvement of 0.26× (variance=0.1) to 41× (variance=0.3) over Fail_Stop.

Our RePRAM schemes achieve the lifetime results that are comparable to, or better than, the ECP scheme at a small fraction of the ECP's hardware cost. In both dimensions, we first deploy PDR that stores parity separately from PCM data pages. Therefore, under PDR, PRAM capacity shrinks only when PCM pages have reached 160 bit failures. This lets the PRAM capacity to degrade gracefully in PDR. However, under MDR, PRAM pages are mirrored onto each other and the PRAM capacity degrades more rapidly. In Dim-1, we initially deploy PDR with the group size of 3 to tolerate faults, and after the PCM blocks begin to exhibit 80 faults (three-way fault threshold shown in Figure 1), we switch to MDR for the rest of PCM block's lifetime. We can see that Dim-1 design yields the lifetime results that are slightly worse than ECP (1% when the variance is 0.1 to 9.5% when the variance is 0.3). This shows a good trade-off from the hardware cost perspective, as our design involves using most off-the-shelf components versus the extensive modifications to main memory design needed by ECP. Under Dim-2, we also deploy PDR with the group size of 3 in the initial stages, and after the PCM blocks begin to exhibit 80 faults, we switch to PDR with the group size of 2 for the rest of PCM bock's lifetime. We find that Dim-2 exhibits better lifetime than ECP (0.5% for the variance=0.1 to 4.4% when the variance=0.3). Overall, the proposed RePRAM schemes achieve good lifetime improvements for PCM, ranging from 0.26× (when the variance is 0.1 in Dim-1) to 43× (for the variance of 0.3 in Dim-2) over Fail_Stop.

One might argue about why low-cost schemes such as RePRAM should be advantageous over ECP, when both schemes have comparable lifetime results. We note that ECP requires custom PRAM design, and offers a hardwired solution at about 12% memory overhead where there is less flexibility for users to upgrade to further lifetime-enhancement techniques. In contrast, RePRAM uses off-the-shelf modules and minimizes hardware changes for easier adoption and for future upgrades.

B. Performance Impact

We evaluate the performance impact resulting from RePRAM using SESC [23], a cycle-accurate, execution-driven simulator. Our baseline system models an Intel Nehalem-like four-core processor [7] running at 3GHz, 4-way, out-of-order core, each with a private 32KB, 8-way set-associative L1 and a shared 4MB, 16-way set-associative L2. The L1 caches are kept coherent using the MESI protocol. The block size is 64Bytes in all caches. We model 4GB PCM main memory with 4KB pages with read access latency of 50ns and write access latency of 1μs [16]. For storing parity, we include an additional 16MB DRAM that can accessed simultaneously during the PRAM accesses. When DRAM buffer is full, we model a 96000 cycle (32μs) latency to write parity information to disk.

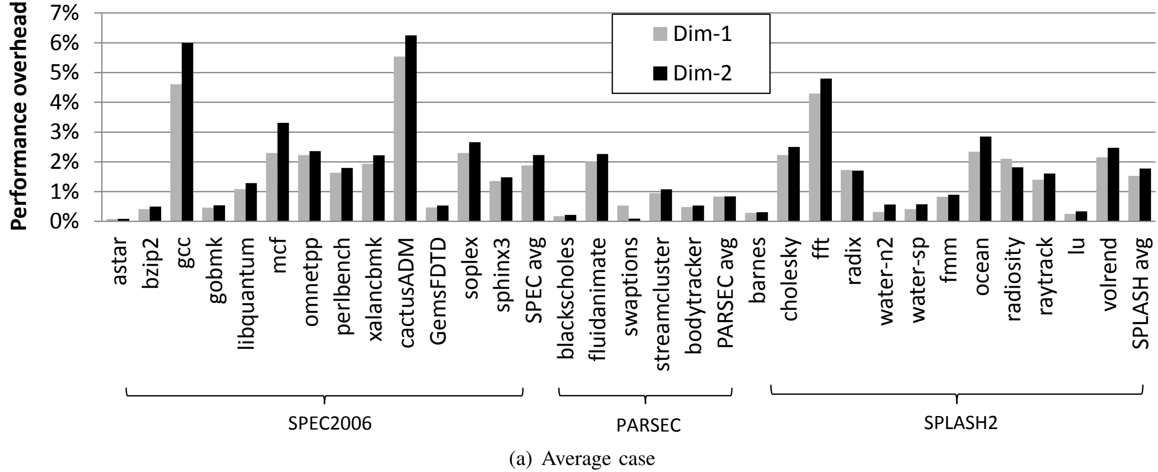
To model RePRAM effects, every L2 access first queries a bloom filter, that has a three-cycle latency to determine whether the page is faulty or not. We use 1024-entry $CACHE_{MDR}$ and $CACHE_{PDR}$ in our experiments, where we observe an average miss rate of 3% (and 8% worst case). Note that we do not assume fixed latencies for memory lookups corresponding to MDR or PDR. Our configuration setup has a common memory bus between cores to fully model memory read/write latencies and bandwidth.

For faulty pages, in Dim-1, we access $CACHE_{MDR}$ and $CACHE_{PDR}$ simultaneously each of which have a 2 cycle latency. In Dim-2, we just access $CACHE_{PDR}$. On a mapping cache miss, we model an additional latency of 500 cycles to perform mapping lookup from lower levels of memory hierarchy and store it in the mapping cache for future use. Upon receiving all the pages in a group, we have a 20 cycle penalty for parity reconstruction and recovering faulty bytes from mirror copies.

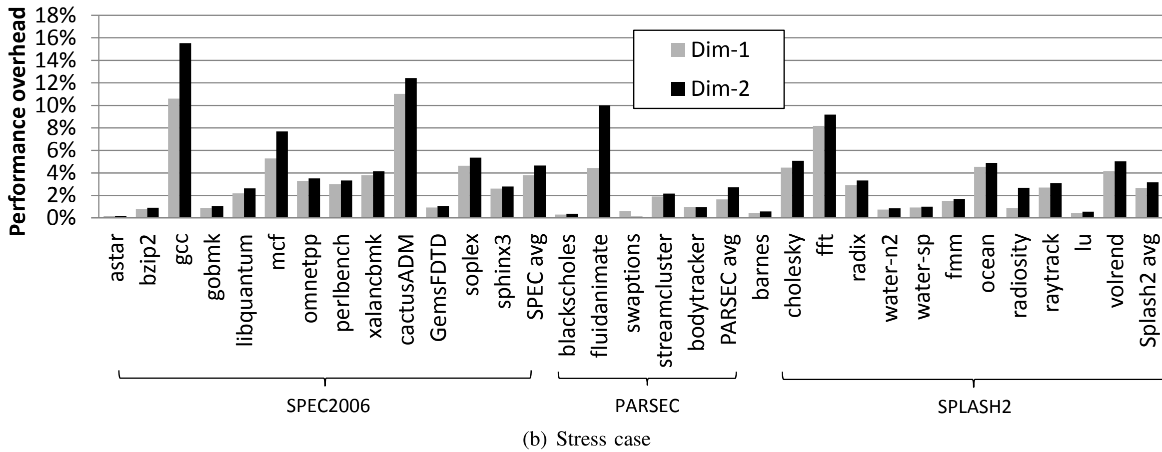
We show performance overheads on two sets of scenarios for Dim-1 and Dim-2 configurations:

- *average case*: In this scenario, we assume that a randomly picked page has 50% probability that it is faulty. Therefore, a randomly picked 50% of the PCM pages are considered to be faulty. Of these, 50% of faulty pages (25% of the total pages) are mapped under PDR (group size = 3), and 50% (25% of total pages) are under MDR in Dim-1 and under PDR (group size = 2) in Dim-2. The remaining 50% of total pages are assumed to be non-faulty.
- *stress case*: In this scenario, we assume that a randomly picked page is always faulty. Of these, a randomly picked 50% of the PCM pages are mapped under PDR (group size = 3). The remaining 50% pages are under MDR in Dim-1, and under PDR (group size = 2) in Dim-2.

We use 21 different memory-intensive and CPU-intensive application mix from SPEC2006 [27], PARSEC-1.0 [1] and Splash-2 [30] benchmark suites with reference input sets.



(a) Average case



(b) Stress case

Figure 4. Performance Overheads of RePRAM on Spec2006, PARSEC-1.0 and Splash-2 applications.

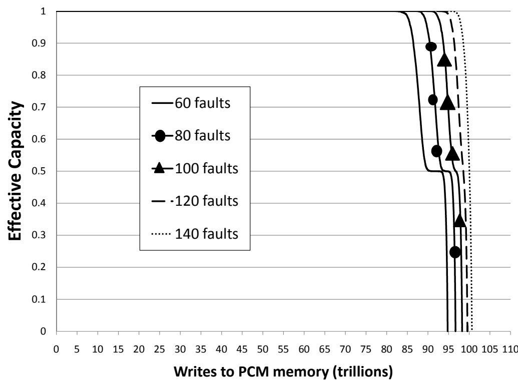


Figure 5. Lifetime comparison for various configurations of Dim-1 (at variance=0.2). For each configuration, we use PDR until the specified number of faults, and later we switch over to MDR for the rest of the PCM block's lifetime.

For SPEC2006 and PARSEC applications, we fast forward the first five billion instructions and simulate the next one billion in detail. For Splash2 applications, we simulate them from start to end. In addition, we run SPEC2006 benchmarks as single-core applications individually one at a time. For PARSEC and Splash2 benchmarks, we spawn four parallel threads, one each on every core that share L2 and lower level memory substructures.

Figure 4(a) shows the performance impact of our RePRAM schemes in the average case scenario. We see that, all 21 of our applications show the overheads less than 7%, and the highest performance overhead (6.2%) occurs in cactusADM, followed by 5.9% for gcc in Dim-2. Under Dim-1, we notice an average of 1.87%, 0.8%, and 1.5% across SPEC2006, PARSEC and Splash2 respectively, whereas the corresponding averages under Dim-2 are 2.2%, 0.8%, and 1.7%. We notice an increased memory access rate in Dim-2 (due to continuous use of PDR configuration) along with a higher rate of misses in mapping caches contribute to increased performance impact than Dim-1. This effect is especially pronounced in benchmarks such as gcc, cactusADM, and mcf. Also, issuing multiple requests for group pages creates contention for memory bus bandwidth and degrades performance in benchmarks such as fft, ocean, and volrend.

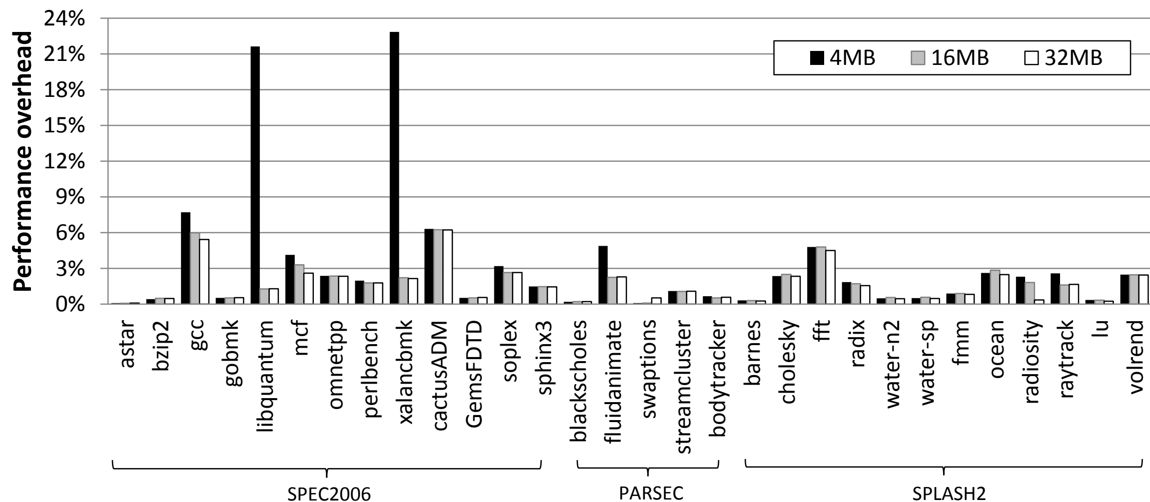


Figure 6. Performance overheads for different sizes of DRAM buffer in PDR (average case).

Figure 4(b) shows the performance impact of our RePRAM schemes in the stress case scenario. We see that, all 21 of our applications show the overheads less than 16%, and the highest overhead (15.5%) occurs in gcc, followed by 12.4% for cactusADM in the Dim-2 design. In Dim-1, we notice an average of 3.78%, 1.64%, and 2.65% across SPEC2006, PARSEC and Splash2 respectively, whereas the corresponding averages in Dim-2 are 4.66%, 2.71%, and 3.16%. Due to multiple memory accesses (depending on the group size) and demand for mapping cache entries, we notice an increased average performance impact in Dim-2 than Dim-1 across various benchmark suites. Particularly, this effect is seen in benchmarks such as gcc, mcf, and fluidanimate. Similarly, issuing multiple requests incurs performance impact in benchmarks such as cactusADM, fft, and ocean.

C. Area Overheads

We use Cacti 5.3 [13], an integrated model for cache and memory access time, cycle time, area, leakage and dynamic power. We use this tool to model our two 1024-entry mapping caches, and a compact bloom filter that has space to store 1 million PCM page entries inside the processor chip. We assume that these hardware structures would be integrated with an on-chip memory controller. We use 45 nm technology node in our experiments.

Table I shows the area estimates of our proposed RePRAM hardware. We find that our area overheads both on-chip and off-chip are less than 1% and our proposed hardware can be easily integrated into the existing processors with minimal changes. We note that this is much cheaper than prior works such as ECP [24] that incur substantial area overheads to store replacement bits along with additional circuitry like row-decoders in order to store the error correcting pointers.

Processor Die	263 mm^2 [7]
Bloom Filter	2.25 mm^2
$CACHE_{PDR}$	0.08 mm^2
$CACHE_{MDR}$	0.08 mm^2
On-chip area overhead	0.92%
16 MB DRAM Buffer overhead (compared to 4GB PRAM)	<0.5%

Table I
AREA OVERHEADS OF ON-CHIP MAPPING CACHES, BLOOM FILTER AND OFF-CHIP DRAM BUFFER IN REPRAM.

D. Sensitivity Experiments

In this subsection, we present further analysis to show how our RePRAM schemes behave under various parameters and possible configurations. We show that such analyses present key insights to the user along with experimental justification for choosing user-desirable set of parameters toward building the RePRAM system.

1) Lifetime vs. Redundancy Levels: In Dim-2, PDR is used throughout the lifetime of PCM device. An advantage of using just PDR is that every PCM block incurs only the write operations directly intended for them, i.e., the use of PDR configuration itself does not impose additional writes to the PCM blocks. However, in Dim-1, we switch to MDR beyond a specific point in time and this may likely accelerate the wear-out of PCM blocks. Since mirroring duplicates the writes to two different PCM blocks, every write ages two separate PCM blocks, contributing to the diminishing capacity of the PCM device. Therefore, we seek to investigate when to switch to MDR during the lifetime of the PCM block.

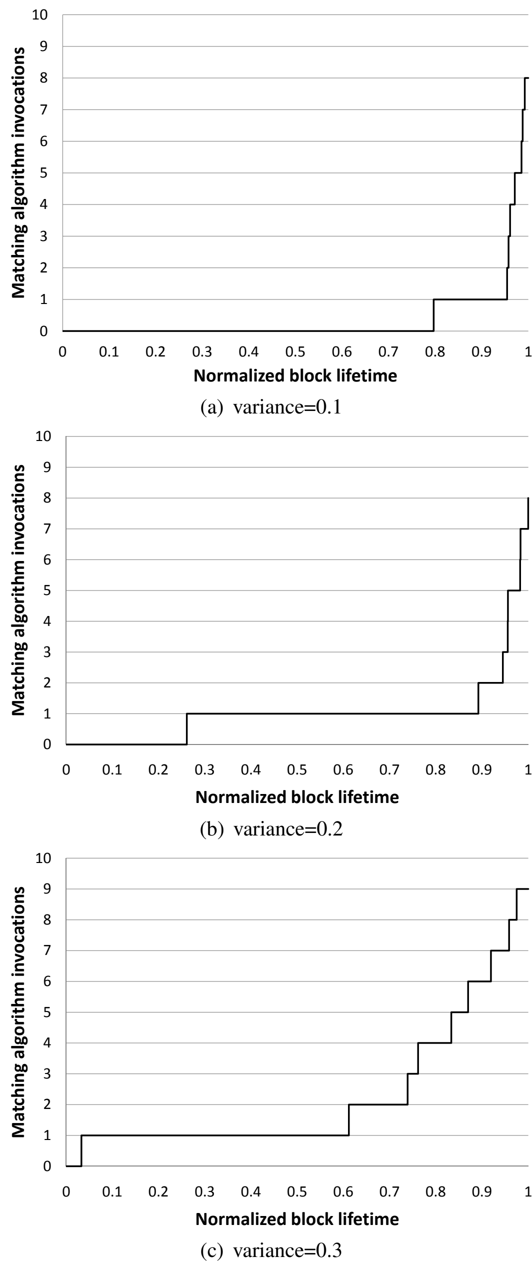


Figure 7. Frequency of invocation of matching algorithm through PRAM block's lifetime.

Figure 5 presents comparison of lifetime for various configurations of Dim-1. In each case, we begin with PDR with group size of 3. After we cross a "specific number of faults" shown in each configuration, we switch to MDR where data is mirrored onto two PCM blocks. It is important to note that, as we begin to operate in PDR mode with higher number of faults, the cost associated with three-way mapping increases sharply (Section III-A). For example, the average number of random trials to complete a three-way matching for PCM blocks with up to 80 faults is one more than three-way matching for up to 60 faults and the corresponding lifetime improvement is 2.1%; whereas, the number of trials to do three-way matching for PCM blocks with up to 140 faults is ten-fold compared to the blocks with up to 60 faults and the corresponding lifetime improvement is 5.2%. Our experiments clearly show that there are diminishing returns in PCM lifetime improvement if we try to remain longer under PDR with higher order group sizes.

2) Sensitivity of PDR to DRAM Buffer Size: A factor that could be critical to minimizing the performance impact in PDR is the size of DRAM buffer (that we use for parity lookup). To investigate this effect, we present additional experiments to determine the size of DRAM buffers needed for storing the parity in our benchmarks. Due to smaller ratio of parity to PCM data pages, we find that relatively smaller DRAM buffer sizes (compared to PCM main memory) work very well toward minimizing the overall performance impact. Figure 6 shows the overheads experienced by PDR when we experiment with 3 different DRAM buffer sizes, viz., 4MB, 16MB and 32MB. Our results show that 16MB is sufficient to keep the performance overheads at less than 7% (average case), while the 4MB DRAM buffer incurs high performance overheads of up to 23% in libquantum and xalancbmk benchmarks. A 32MB DRAM buffer shows negligible benefit over 16MB for performance overheads in almost every benchmark, indicating that DRAM capacity (above 16MB) is no longer a bottleneck beyond the initial compulsory misses to parity information. This result shows that the amount of DRAM buffer needed to maintain low performance overheads across the three benchmark suites is just $\frac{1}{256}$th of the capacity invested in PCM main memory. Furthermore, this 16MB DRAM buffer has shown less than 16% overheads even for the stress case (seen in Figure 4(b)).

3) Frequency of Matching Algorithm Invocations: To understand the OS and software overhead, we perform the experiments to quantify the average number of times that matching algorithm needs to be invoked during a PRAM block's lifetime. Figure 7 presents the results for process variation factors of 0.1, 0.2 and 0.3. In this test, we count the number of writes (presented as normalized lifetime), that has been performed to a PRAM block, before the block encounters the first bit fault. Now that the block is no longer pristine, the matching algorithm is invoked to find a group of compatible blocks. As additional writes are performed to this block and its group pages, they incur more bit faults, and eventually become incompatible due to faults in the same byte positions. At this point, the matching algorithm needs to be invoked again to find a new set of compatible pages for this faulty PCM block. We repeat this matching process until the PRAM block has exceeded 160 bit faults (when we discard the block permanently). From Figure 7, we can see that matching algorithm invocations are often clustered towards the end of PRAM block's lifetime for

process variation of 0.1, while it is more spaced out for process variation of 0.3. In all of the cases, we find that average number of matching algorithm invocations is less than 10 throughout the PRAM block's lifetime. Clearly, the matching algorithm accounts for a small fraction of performance overhead in comparison to the actual writes performed on the PRAM block.

V. Conclusions and Future Work

In this paper, we explore a number of dynamic redundancy techniques to resuscitate faulty PCM pages and improve the lifetime of PCM-based main memory systems. We explore different design choices along two dimensions namely, 1) switching from PDR to MDR, and 2) reducing the group size in PDR from three to two. We show that, by intelligently combining the use of PDR and MDR schemes, the lifetime of PRAM can be improved by upto 43× over Fail_Stop.

As future work, we plan to extend RePRAM to incorporate application-specific characteristics and system energy awareness. We will focus on capturing the key features of the memory-intensive applications, and tune the hardware to adjust to the performance demands and energy constraints. Furthermore, we will extend this work by investigating other resistive memory technologies, as well as, system-level effects needed to tolerate failures resulting from write endurance limitations inherent in some of these devices.

VI. Acknowledgments

This material is based upon work supported in part by the National Science Foundation under CAREER Award CCF-1149557, and grants CCF-1117243 and OCI-0937875.

References

[1] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Princeton University Technical Report TR-811-08*, January 2008.

[2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[3] William A. Brant, Michael E. Nielson, and Edde Tin-Shek Tang. Power failure responsive apparatus and method having a shadow dram, a flash rom, an auxiliary battery, and a controller. In *US Patent 5,799,200*, 1998.

[4] Jie Chen, R. C. Chiang, H. Howie Huang, and Guru Venkataramani. Energy-aware writes to non-volatile main memory. *SIGOPS Oper. Syst. Rev.*, 45(3):48–52, January 2012.

[5] Jie Chen, Zachary Winter, Guru Venkataramani, and H. Howie Huang. rpram: Exploring redundancy techniques to improve lifetime of pcm-based main memory. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[6] Sangyeun Cho and Hyunjin Lee. Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. In *MICRO*, 2009.

[7] Intel Corporation. Intel core i7-920 processor. *http://ark.intel.com/Product.aspx?id=37147*, 2010.

[8] Dave Hayslett. System z redundant array of independent memory. In *IBM SWG Competitive Project Office*, 2011.

[9] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *ASPLOS*, 2010.

[10] Lei Jiang, Yu Du, Youtao Zhang, B.R. Childers, and Jun Yang. Lls: Cooperative integration of wear-leveling and salvaging for pcm main memory. In *DSN*, pages 221 –232, june 2011.

[11] Nikolai Joukov, Arun M. Krishnakumar, Chaitanya Patti, Abhishek Rai, Sunil Satnur, Avishay Traeger, and Erez Zadok. Raif: Redundant array of independent filesystems. *MSST*, 0:199–214, 2007.

[12] Randy H. Katz. Raid: A personal recollection of how storage became a system. *Annals of the History of Computing, IEEE*, 32(4), 2010.

[13] HP Labs. Cacti 5.3. *http://quid.hpl.hp.com:9081/cacti/*, 2010.

[14] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA*, 2009.

[15] Rami Melhem, Rakan Maddah, and Sangyeun Cho. Rdis: a recursively defined invertible set scheme to tolerate multiple stuck-at faults in resistive memory. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2012.

[16] Numonyx. Phase change memory: A new memory to enable new memory usage models. *White Paper http://www.numonyx.com/*, 2009.

[17] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD*, pages 109–116, 1988.

[18] Devices Process Integration and Structures. International technology roadmap for semiconductors. *http://www.itrs.net*, 2007.

[19] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, 2005.

[20] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *MICRO*, 2009.

[21] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA*, 2009.

[22] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: a scalable technology. *IBM J. Res. Dev.*, 52, July 2008.

[23] Jose Renau et al. SESC. *http://sesc.sourceforge.net*, 2006.

[24] Stuart Schechter, Gabriel H. Loh, Karin Straus, and Doug Burger. Use ecp, not ecc, for hard failures in resistive memories. In *ISCA*, 2010.

[25] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[26] Nak Hee Seong, Dong Hyuk Woo, Vijayalakshmi Srinivasan, Jude A. Rivers, and Hsien-Hsin S. Lee. Safer: Stuck-at-fault error recovery for memories. In *MICRO*, 2010.

[27] Standard Performance Evaluation Corporation. SPEC Benchmarks. *http://www.spec.org*, 2006.

[28] Chris Wilkerson, Alaa R. Alameldeen, Zeshan Chishti, Wei Wu, Dinesh Somasekhar, and Shih-lien Lu. Reducing cache power with low-cost, multi-bit error-correcting codes. In *ISCA*, 2010.

[29] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The hp autoraid hierarchical storage system. *ACM Trans. Comput. Syst.*, 14, February 1996.

[30] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, June 1995.

[31] B. Yang, J. Lee, J. Kim, J. Cho, S. Lee, and B. Yu. A low power phase change random access memory using a data comparison write scheme. In *ISCAS*, 2007.

[32] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P. Jouppi, and Mattan Erez. Free-p: Protecting non-volatile memory against both hard and soft errors. In *HPCA*, February 2011.

[33] Wangyuan Zhang and Tao Li. Characterizing and mitigating the impact of process variations on phase change based memory systems. In *MICRO*, 2009.

[34] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA*, 2009.