WL-Reviver: A Framework for Reviving any Wear-Leveling Techniques in the Face of Failures on Phase Change Memory

Jie Fan†, Song Jiang‡, Jiwu Shu†, Long Sun†, Qingda Hu†

†Department of Computer Science and Technology
Tsinghua University
Beijing, China

†Tsinghua National Laboratory for Information Science and Technology
Beijing, China

‡Department of Electrical and Computer Engineering
Wayne State University
Detroit, MI, USA

†{fanj11, shujw, sun-l12, hqd13}@mails.tsinghua.edu.cn

‡sjiang@eng.wayne.edu

***Abstract*—While Phase Change Memory (PCM) has emerged as one of most promising complements or even replacements of DRAM-based memory, it has only limited write endurance. Because of uneven write distribution, PCM is highly likely to have early failures, which can spread over the chip space and leave the entire chip unusable. Wear leveling is an indispensable technique to even out wear caused by the writes. However, because of process variation early failure cannot be fully avoided. State-of-the-art wear-leveling schemes, such as Start-Gap and Security Refresh, cease to function once even a single block failure occurs because their designs require persistent writable address space for wear leveling operations. Existent solutions attempting to address the problem demand substantial OS supports, such as explicit space allocations and data migrations. The demand on substantial OS cooperation creates a barrier to widespread adoption of the PCM technique.**

While fault-tolerance techniques, such as FREE-p and zombie, that remap failed blocks to inaccessible but healthy space have the potential to address the wear-leveling issue by relocating data from failed blocks to healthy ones, they cannot work together with the wear-leveling schemes as data migration may change placement of relocated data. In this paper, we propose a framework, *WL-Reviver*, that allows *any* in-PCM wear-leveling scheme to keep delivering its designed leveling service even after failures occur in its working address space. The design is unique on two aspects: (1) it leverages the fault-tolerance techniques so that they can work together with the wear-leveling schemes; and (2) it requires *no* OS supports additional to what're available to today's DRAM-based memory system. Furthermore, WL-Reviver is a lightweight framework of very low overhead. Our extensive experiments show that WL-Reviver can efficiently revive a wear-leveling scheme without compromising the scheme's wear-leveling effect.

***Keywords*-PCM; Fault Tolerance; Wear Leveling;**

I. Introduction

Resistive memory has emerged as a promising technology when the scaling of DRAM technology to smaller feature sizes (beyond 30 nm) becomes increasingly difficult [10], [13]. With higher scalability and being non-volatile, they are expected to complement or even replace DRAM as main memory in the near future. Among a number of resistive memories currently available, phase-change memories (PCM) is the most likely technology for volume production [1], [2] and has seen the most research efforts [9], [14], [15], [18], [20], [21].

Corresponding author: Jiwu Shu

A. PCM's Limited Endurance and Wear Leveling

One of most challenging constraints on PCM is its limited endurance. That is, after a limited number of writes on a memory cell (on average about 10^7 - 10^8), the cell permanently fails. Because of existence of write locality, cells of PCM memory are probably not uniformly worn. That is, some cells can become unuseful earlier than others, leading to loss of memory space. A computer becomes unavailable once a certain percentage of its memory capacity gets lost. To postpone this effect as late as possible, practitioners must apply wear leveling techniques to spread the wear evenly across the entire address space. A wear leveling scheme may periodically change address mapping so that addresses mapped to more heavily written blocks can be remapped to less written blocks.

While it is possible for the operating system (OS) to conduct the remapping and accordingly to provide PCM wear leveling functionality, an OS-based design is a less desirable solution. Such a design may require substantial changes to OS, including page usage monitoring, page mapping, and data migration. It is much more expensive to preform these operations with software than that in hardware. In addition, as wear leveling is an indispensable operation in the use of PCM, an OS-based solution requires an OS supporting wear leveling as a prerequisite for adopting PCM in the memory system. This creates an artificial barrier to a widespread adoption of PCM-based main memory. Therefore, if it is possible every effort should be made to leave OS out of the execution of wear leveling functionality in PCM. As an analogous example, flash-based SSDs always implement their wearing leveling functionalities within the devices, rather than requiring OS to play the role.

To this end, state-of-the-art PCM wear-leveling schemes,

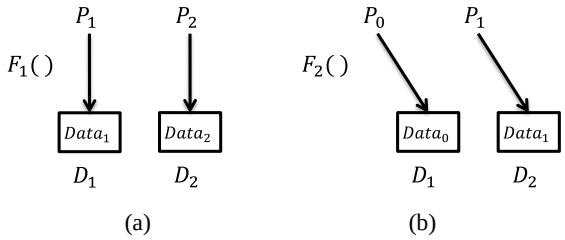

Figure 1. Illustration of changes of mapping from physical address (PA) to device address (DA) because of data migration required by a wear-leveling scheme. (a) Initially user data "$Data_1$" at PA P_1 is stored at DA D_1 when mapping function F_1 is applied; (b) $Data_1$ is migrated into DA D_2, and the mapping function is accordingly changed to $F_2(P_1) = D_2$.

such as Start-Gap [21] and security refresh [22], are all designed to function in the memory controller without any OS involvements. In the meantime, the traditional approach of tracking every memory block's access count, comparing the counts of different blocks, and using an indirection table to perform address mapping, is too expensive and non-scalable to be implemented in hardware. Accordingly the schemes designed for PCM give up the flexibility enabled by the mapping table to avoid the expensive table maintenance cost and table look-up time spent on each PCM access, which could significantly increase access latency and consume substantial energy. Instead, they all use easy-to-compute algebraic mapping functions for the purpose.

B. Wear Leveling in the Face of PCM Failures

In a computer system, software uses physical address, or *PA* in short, to access the memory device, and the memory controller translates this address into its corresponding device address, or *DA* in short. A memory block is persistently identified with a specific device address. As we have explained, for space and time efficiency a wear-leveling scheme uses a PA-to-DA address mapping function, rather than a mapping table, to quickly determine which memory block should be accessed upon a software-initiated PA access. As illustrated in Figure 1, a fundamental and essential operation of any weal-leveling scheme is to periodically migrate data and accordingly change PA-to-DA mapping function. When a condition for the wear-leveling operation is met (e.g., a certain number of writes have been serviced on the PCM), a block of data is migrated from its current storage location (memory block at DA D_1), to a selected new location (memory block at DA D_2). As illustrated, before the migration physical address P_1 is mapped to D_1 using function F_1 ($F_1(P_1) = D_1$) and any software-initiated access to data at P_1 occurs at D_1. After the migration, the access should occur at D_2. For correctness, the mapping function has to accordingly change to F_2 so that $F_2(P_1) = D_2$. When the block of data keeps its migration from one memory block to another one, the function needs to keep changing so that P_1 always refers to the same block of data. It is an invariant in any wearing leveling schemes that the same valid PA consistently refers to the same data no matter where it is physically migrated, unless it is updated by the software via a re-write.

To ensure that writes are evenly spread over a given range of PCM space, a wear-leveling scheme schedules a block of data to migrate into every memory block in the space over a certain period of time. In other words, any device address D_i, where $i = 0, 1, ..., N-1$ and N is the total number of blocks in the space, can be the mapping target of any physical address P_j, where $j = 0, 1, ..., N-1$. If the memory block at D_i fails, the wear-leveling scheme does not function as designed because (1) a block of data cannot be migrated into D_i, and (2) attempt for accessing P_j, which is mapped to D_i, will fail. Even worse, because every physical address (P_j) can be mapped to the failed block (D_i) at different time periods of the wear-leveling execution, the OS would be informed that P_j is a failed address and ultimately be misled to believe that all memory blocks fail.

To address this issue, one might attempt to change the data migration schedule to keep data from migrating into the failed memory blocks, or DAs, and to modify the mapping function to exclude the DAs from being mapped to. However, as failed blocks can appear at any random locations and are accumulating, it is unlikely to design a set of functions, rather than to use a mapping table, to cover scenarios of various failure distributions. Though the state-of-the-art PCM wear-leveling schemes can achieve a near-perfect wear-leveling effect even with writes of strong spatial locality and malicious wear-out attacks [21], [22], they stop functioning with even a single block failure, leading to accelerated failures and loss of memory space. Furthermore, due to process variations there is a high variability in PCM cells' lifetime and some cells are subject to early failures [18]. Therefore, it is critical to revive the wear-leveling techniques that cease to function with failures.

C. Page-recovery Schemes vs. the Wear-leveling Issue

One of the studies on PCM's fault tolerance is concerned with recovery of an OS page when some blocks of the page fail. This study is important as it keeps an entire page from being lost due to failure of some of its blocks. An OS page can be 4KB while a block is usually of a cacheline size. Representative schemes from this study include FREE-p [23] and Zombie [8]. Their idea is to link a failed block to a healthy block in another page that has been considered by OS as a failed one and is dedicated for supplying its healthy blocks for being linked to other pages' failed blocks.

By making a failed block appear to be available for storing data via its linked healthy block, these page-recovery schemes have the potential to allow a wear-leveling scheme to continue functioning after occurrence of block failures. After the linkage, any access, including that for data migration due to wear-leveling operations, to the failed block is redirected to the linked healthy but software-unreachable

block[1]. We name this healthy block as a *shadow block* of the corresponding failed block. These shadow blocks are from a pre-reserved space, such as the page considered as failed one by OS, dedicated for the purpose. Because blocks in the space are not used to store data until they are linked to failed blocks. the space must be incrementally obtained to maintain a high space utilization.

D. Challenges on Reviving Wear-leveling Operations

While the approach suggested by the page-recovery schemes for reviving wear-leveling operations in the face of block failures sounds promising, it has three critical issues challenging its feasibility and efficacy.

The first issue is how the reserved space can be incrementally obtained. For the sake of space efficiency, one must reserve a relatively small amount of additional space once currently reserved space has been used up and failing of a block demands for a shadow block [12], [23]. This means one block used to be mapped into the software's PA address space and was accessible to the software needs to become unreachable. Because only operating system (OS) is able to determine which PA address space is accessible via page allocation and deallocation, any explicit request for additional reservation requires OS to change its memory allocation and possibly move data among its pages. As mentioned, the demand of such an OS support is highly undesirable and should be avoided if possible.

The second issue is whether to involve unlinked blocks in the reserved space in the wear leveling operations. Once physical blocks, or their corresponding DAs, are reserved, they have to remain unused and be ready to be linked to failed blocks and store their data. This means that these blocks cannot participate in the wear leveling, which may migrate data into them. Excluding them from the wear leveling can compromise entire PCM chip's leveling efficacy. A more serious issue of the reservation method is that the wear leveling scheme may have to be substantially modified [12], assuming such a modification is possible. Because mapping function of a scheme is designed to accommodate its scheduled data migration, DAs of the reserved blocks, whose data are not involved in the data migration, have to be removed from the functions' target set (or codomain). Accordingly, the mapping functions have to be revised after the removal. Even worse, some wear-leveling schemes, such as security refresh [16], requires its mapping functions' target set (or mapped DA space) be of size of power of 2 in terms of block count. In this case, the schemes have to resort to mapping table [12] if each reservation did not take away half of a scheme's target set.

The third issue is that the page-recovery schemes themselves can be at odds with the wear-leveling operations if shadow blocks are allowed to participate in the wear leveling. Data stored in a shadow block actually belongs to its corresponding failed block, and the failed block records its shadow block's address. When the data in the shadow block is migrated to another block in a wear-leveling operation, the failed block cannot find its data via its recorded address. Because in both FREE-p [23] and Zombie [8] a shadow block does not record its failed block's address, it can be very expensive to re-link the failed block to the new shadow block. Therefore the wear-leveling operations essentially invalidate the page-recovery schemes.

[1]When a block at a DA is claimed to be unreachable to software, the software, including OS, does not access data at a PA that is mapped to the DA.

E. Our Solution

In this paper we propose a framework in which wear-leveling schemes continue to function with failures and the page-recovery schemes can be used together with active wear-leveling operations. In the design all the aforementioned issues are effectively addressed in a very lightweight manner. Specifically, the framework is unique at four aspects. First, it does not require any new OS supports. Second, it consistently performs wear-leveling operations over any blocks, including those in the reserved space. Third, it does not require any modifications of wear-leveling schemes. Fourth, the page-recovery schemes can continue to function without any efforts of updating links after data migration.

The enabling technique of the framework is to reserve virtual space in the PA address space, instead of memory blocks in the DA address space, to provide shadow blocks. A key observation inspiring the design is the fact that once an access failure is reported to the OS, OS would discontinue its use of the page containing the failure [3], where page is the unit for OS to manage its memory. A page contains multiple PAs, each mapped to a memory block at a DA, which is unit for wear-leveling. While the OS excludes the page, where a failure has been reported, from further accessing, the PAs contained in the page are essentially reserved without an explicit request to the OS. Although the reserved PAs themselves are simply addresses, or they represent virtual space, each of them is mapped to a DA. As long as a PA is reserved and becomes unaccessible to the software, its mapped DA, a memory block, is also reserved and available to serve as a shadow block for storing data on behalf of a failed block. However, a reserved PA is possibly mapped to a different memory block, or DA at different time, as the wear leveling scheme migrates data and changes the mapping. Because a failed memory block is only linked to a reserved PA, their relationship is independent of the data migration and change of PA-DA mapping function.

In this paper we present the proposed design as a framework, named as *WL-Reviver*, to revive any wear-leveling schemes on PCM-based memory in the face of failures.

II. Related Work

To address PCM's issue of limited endurance, many efforts have been made to achieve two objectives. One is to use additional resources to correct errors, and the second is to prevent early failures.

Regarding the first objective, PCM is more likely to have permanent stuck-at faults, rather than transient errors, that can be gradually accumulated over time within the lifetime of a data block. To postpone the occurrence of first uncorrectable fault on PCM, many fault tolerance schemes have been proposed, including ECP [20], SAFER [21], and RDIS [15]. In particular, ECP corrects errors by permanently encoding the locations of failed cells into a vector and assigning other cells to replace them. With additional metadata the schemes can correct a certain number of errors in a bit group before the group is declared as failed. A bit group usually consists of 64bits to 512bits. WL-Reviver is designed to help manage failures that cannot be tolerated by the schemes.

The second objective is at least as important as the first one, because a PCM chip or an entire PCM-based memory can become unavailable with only a small percentage of blocks failed and most of them still alive. There are two sources leading to early failures. One is the process variation, which causes high lifetime variability across PCM cells. The other is non-uniformly distributed writes. The proposed solutions can be categorized into three groups.

The first group of solutions is to dynamically allocate hardware resources for error correction so that early-failed blocks can stay alive longer for storing data. As an example, Pay-As-You-Go (PAYG) reserves a certain amount of metadata space for a set of bit groups and uses the space for any of the groups only when their faults cannot be corrected by their local metadata [18].

The second group is to prevent pages from early loss due to failure of its blocks. One such scheme is FREE-p, which acquires some reserved data space and uses the space to hide any failed blocks via pointers embedded in the failed blocks [23]. Zombie [8] pairs a failed block in a working memory page with a spare block in a disabled page. Zombie also uses a pointer (or a link) to record the pairing relationship. This pairing may enable various error correction codes between the failed and spare blocks. As we have discussed, migration of data in the spare blocks in the wear leveling operations can foil the efforts of these page-recovery schemes and WL-Reviver provides a solution to this issue.

The third group is to level wear on the PCM. Though the technique can be applied at the bit level by shifting data placement offset in a bit group [20], [24] and at the byte level [25], wear leveling techniques are most commonly used at the cacheline level for migrating data across memory blocks. Start-Gap [21] and Security Refresh [22] are two representative schemes, which also consider malicious attacks that keep writing at the same set of addresses. Though they are effective and play an irreplaceable role for extending PCM's lifetime, they cease to function with occurrence of the first block failure in their working address spaces as a contiguous address space is required. WL-Reviver effectively addresses the issue with an efficient address re-mapping technique.

LLS [12] is a scheme sharing a similar goal with WL-Reviver. It dynamically reduces software-useable address space to acquire reserved space for re-mapping failed blocks. Compared with WL-Reviver, LLS is inadequate in four aspects. First, LLS has to rely on OS's support to obtain reserved space. To maintain contiguous address space for wear leveling, LLS has to constantly acquire space from either higher or lower device addresses. Consequently, to reserve space for LLS, an OS may have to conduct expensive data relocation operations among pages. Second, LLS represents the mapping relationship between failed blocks and their backup blocks, which are equivalent to shadow blocks in WL-Reviver, by maintaining the same relative order in their respective block lists. This may constantly incur block insertion operations, and consequently expensive data shifting operations. This also requires LLS to maintain a bitmap that has to be read upon each access of data on a backup block, which can substantially increase PCM's access time. Third, it is less flexible for LLS to manage address space. LLS uses chunk, which is 64MB by default in the paper, as the unit for expanding the reserved space. To maintain contiguous wear-leveling space, LLS partitions blocks into salvaging groups and dictates that a failed block can only use block in the same group as its backup one. Because a group consists of blocks from different chunks, LLS may have to move a new chunk into the reserved space while many idle blocks are in the reserved space without being mapped to failed blocks. Therefore, software-usable space can be unnecessarily reduced. Furthermore, because these idle blocks do not participate in wear leveling, the wear leveling effect is compromised. Fourth, to integrate existent wear-leveling schemes into LLS, one may have to conduct substantial adaptation of the schemes. In contrast, WL-Reviver requires almost no OS support, no data migration other than that demanded by the wear-leveling scheme itself, and minimal data access overhead. It conducts implicit, incremental, and flexible space reservation without requiring any modifications of the wear-leveling schemes for them to be in the framework.

III. The Design of *WL-Reviver*

WL-Reviver is a framework to revive any wear-leveling scheme that becomes incapable to perform its leveling operations due to block failures in its working space. Its strategy is to hide the failures from the scheme by redirecting accesses that are originally to the failed blocks to the reserved

healthy blocks without any new OS supports. To serve as a framework accommodating any wear-leveling scheme, WL-Reviver assumes only one fundamental operation common to any of such schemes, which is to migrate data into a memory block. Below we will describe when and how reserved space is acquired and how a failed block is linked to a reserved block.

A. Acquiring Spare Space to Hide Failed Blocks

There are two issues to address in the acquisition of spare space for hiding currently failed blocks and for being used as reserved space to cover future failures. The first issue is about condition and timing of this acquisition, and the second one is about acquisition method.

As we have discussed, for the sake of widespread and quick adoption of PCM as main memory, minimal involvement of OS is preferred. To this end, WL-Reviver does not modify existent interface between the software and the memory hardware. In addition to supporting read and write commands, the interface also includes mechanism of generating exception to OS when an access error is detected. A standard procedure for OS to handle the exception is to exclude the page associated with the error from its allocation pool and to prevent future access of data in the page. Some system architectures, such as HP Memory Quarantine [3], use firmware to mark memory locations with errors as bad and unavailable before passing the errors to the OS. To avoid computer downtime, especially for servers running mission-critical applications, the OS would make every effort to recover from the errors without shutting down the system or even without closing down the affected threads or processes. To facilitate this effort, WL-Reviver does not require any changes of the interface. The only assumption it makes to acquire spare space is that the page associated with the error reported in the exception will not be accessed by the software. As the assumption had been supported, WL-Reviver is not intrusive to existent systems at all.

Note that wear-leveling is conducted at the unit of memory block to match the memory access unit, which is the cacheline size of the last-level cache, and OS uses memory at the unit of page. As an example, for a 64B memory block, acquisition of a 4KB page is equivalent to receipt of 64 physical addresses (PAs), each mapped to a different memory block at a device address (DA). If one of the PAs is mapped to a healthy memory block, the healthy block can be served as the shadow block of a recently failed block that is demanding for a spare block to hide the failure, or to transparently redirect accesses to the failed block to the shadow block. A salient feature of WL-Reviver distinguishing it from other schemes is that a failed block is *not* directly linked to its shadow block. Instead, the failed block is only linked to a PA that is excluded out of the scope accessible to the software. This PA is named as *virtual shadow block*, as it has to be mapped to a shadow block addressed by a DA to serve the purpose of hiding failures.

For the first block failure, WL-Reviver allows PCM to report it to the OS so that it can acquire a number of PAs included in the page associated with the failure to serve as virtual shadow blocks. For this particular failure, only one virtual shadow block is needed to be linked to the failed block. For a number of following block failures, WL-Reviver does not report them to the OS. Instead, it uses the unlinked PAs as virtual shadow blocks to hide the failures. Only after the acquired PAs are used up, a new block failure will trigger another exception to the OS for another page acquisition. Apparently WL-Reviver incrementally acquires a small set of PAs at a time for two benefits. First, it avoids reservation of a large chunk of memory at once to minimize reduction of software-usable memory space. Second, it can handle most of failures without any interaction with the OS . To facilitate the acquisition method, WL-Reviver sets up two registers. One records the PA currently available to serve as a virtual shadow block, and the other records the last PA available for the purpose. PAs in the range between the current PA and the last PA represent the reserved virtual spare space. When a failed block is detected, the PA recorded in the first register is employed as a virtual shadow block, and then the register increments its recorded PA value by one. If the PA in the first register exceeds that in the second one and an access error occurs, the error is reported to the OS, which discontinues its use of the page that is associated with the error, and the two registers are accordingly set.

In addition to failures detected in the service of accesses from the software, there are failures detected in the execution of data migration due to wear leveling within the PCM. If there are unlinked virtual shadow blocks, the failures can be hidden transparently without interruption to the OS. However, if there are not any unlinked PAs ready for use, there are two options about the timing to acquire the PAs. One is to immediately interrupt the OS to acquire the PAs. The drawback of this option is the need to change the OS. It requires PCM to send the OS an interruption to proactively report errors and explicitly acquire spare space. Apparently the OS needs to be accordingly modified to handle this new type of interruption. The other option is to leave the space acquisition as a reactive operation, or reporting errors only in response to access requests from the software, to keep the OS unchanged. This option requires a delayed space acquisition.

A challenge with the second option, which is taken by WL-Reviver, is its risk of losing data that has been successfully stored in the PCM. The sequence of data migration operations, including source and destination of each migration, has been pre-determined in a wear leveling scheme. Without an immediately available shadow block, data would have to be unsuccessfully written into the failed block and get lost. To address the challenge, WL-Reviver

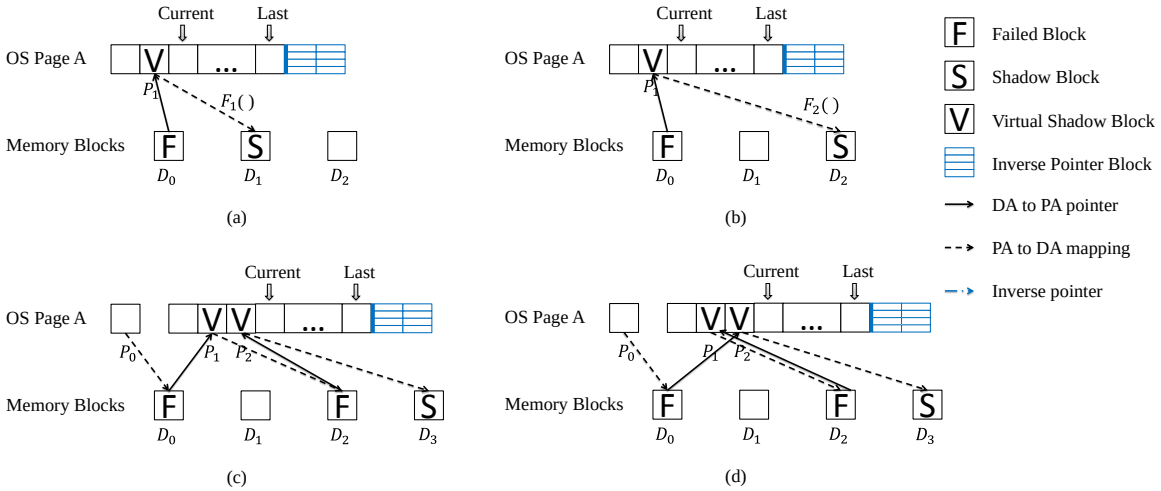

Figure 2. Illustrating how failed blocks, virtual shadow blocks, and shadow blocks are linked. (a) Failed block D_0 points to virtual shadow block P_1, which is mapped to shadow block D_1 via mapping function $F_1()$. (b) After the data in D_1 is migrated to D_2, the mapping function is updated to $F_2()$ so that the shadow block becomes D_2. (c) The software issues write to PA P_0 mapped to the failed block D_0, which has its shadow block D_2. When D_2 fails upon serving the write, D_3 becomes D_0's new shadow block via a new virtual shadow block P_2. (d) Blocks D_0 and D_2 switch their virtual shadow blocks to reduce the chain size from two to one. Now D_2 is on a PA-DA loop and does not have its shadow block. P_0 is a PA accessible from the software. OS page A represents reserved PA spaces unaccessible from the software.

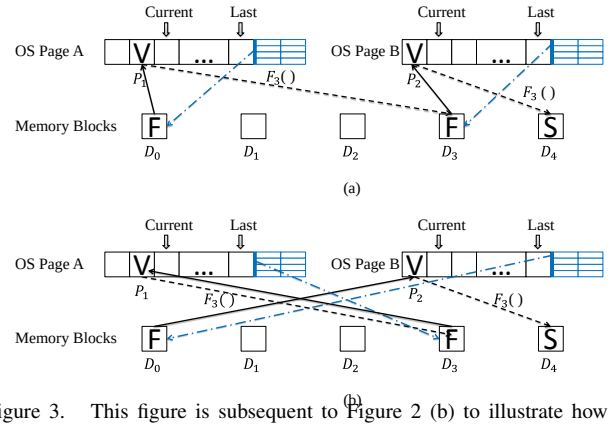

Figure 3. This figure is subsequent to Figure 2 (b) to illustrate how to keep chain length at one after data migration. OS pages A and B represent reserved PA spaces unaccessible from software. (a) When the data in D_2 attempts to migrate into a failed block (D_3), it actually migrates into D_4, or D_3's shadow block. The mapping function is $F_3()$. (b) Leveraging the inverse function of $F_3()$ and reverse pointer in the pointer section of OS page A, P_2 and D_4 become D_0's virtual shadow block and shadow block, respectively. Meanwhile, D_3 has P_1 as its virtual shadow block without having a shadow block, or it is on a PA-DA loop. Reverse pointers are actually stored on memory blocks pointed to by pointers in the pointer section.

temporarily suspends the data migration until the next write request from the software arrives. Then WL-Reviver reports the write to the OS as a failure to obtain spare PA space, even though actually it may not be a failure. With a virtual shadow block, WL-Reviver can resume the data migration. In this strategy, WL-Reviver chooses next write request rather than the next access, which could be a read request, as a victim, because in general a write error is more likely to be recovered than a read error. The OS could redirect an unsuccessful write to an alternative memory location for a retrial. A process encountering a write error can also be recovered by rolling back to its last checkpoint if an execution log is available for reading history data. In contrast, a read failure could crash processes or an entire system by losing critical data, such as log data and system metadata, that have been considered as safely stored ones. Furthermore, the pace of wear-leveling operations is not slowed down by delaying data migration to the next write, because wear-leveling operations are scheduled according to write rate. As an example, one data migration operation is performed for every 100 writes in the Start-Gap scheme [21].

When a system is rebooted, the OS needs to know which of the pages have been providing virtual shadow blocks and to keep the pages from being accessed. To this end, WL-Reviver maintains a bitmap, in which each bit indicates whether the corresponding memory page has been excluded from accesses. As part of memory diagnostics procedure at a system's restart, the bitmap is loaded to inform the OS of the knowledge about PCM's page usage. As only one bit is required for each memory page and a bit is set at most once during a PCM's lifetime, the cost is minimal. To ensure safety of the important metadata, WL-Reviver can keep multiple copies of the bitmap in the PCM with only trivial overhead.

B. Linking Failed Blocks to Virtual Shadow Blocks and Shadow Blocks

To hide a failed block from the wear-leveling scheme, WL-Reviver needs to pair it with a shadow block, which is not directly accessible to the software. Afterwards every access to the failed block is transparently passed to the shadow block. Instead of directly linking the two blocks, WL-Reviver introduces an indirection, which is the virtual

shadow block, to support wear leveling so that migration of data into or out of a shadow block does not break the linkage between the two. Specifically, we need to link a failed block to its virtual shadow block, and further to link the virtual shadow block to the corresponding shadow block, as illustrated in Figure 2.

For the first linkage, WL-Reviver records the PA address associated with a virtual shadow block on its corresponding failed block. To this end, WL-Reviver needs a status bit with each block, indicating whether the block stores a pointer or regular data. We also need to find a space to store the address. There have been proposals about where to store the address in the page-recovery schemes, such as FREE-p [23] and Zombie [8]. In these schemes, they store DA address of a reserved healthy block from the reserved area into the failed block assuming that most bits in the failed block are still available for storing information. For example, FREE-P can reliably store a DA in a failed block with a strong error-correction code, such as 7-modular-redundancy code, as the available space in the block can be much larger than what is needed for storing an address. Zombie uses space in a failed block originally for metadata, such as those for ECC or ECP [20] error correction codes, to store the DA address of a linked healthy block, and may use both the failed block and the healthy block for storing data. WL-Reviver uses the same approach to store an address. The difference is that it records PA address, rather than DA address.

For the second linkage, there is no need to explicitly store a pointer. The PA-to-DA mapping function currently adopted by the wear-leveling scheme provides the link from the virtual shadow block, addressed by a PA, to the shadow block, addressed by a DA. With a migration of data (e.g, from the current shadow block D_1 to another block D_2, as shown in Figure 2(a)), the wear-leveling scheme accordingly updates its mapping function. Consequently, the linkage and shadow block are automatically updated (as shown in Figure 2(b), the mapping function is updated from F_1 to F_2 and the shadow block changes from D_1 to D_2). By using a virtual shadow block, which is simply a PA address, as an indirection, WL-Reviver allows a failed block to be efficiently linked to a constantly moving shadow block without rewriting pointers. When a failed block is linked to a shadow block, any read and write requests to the failed block, including those incurred due to data migration, are served at its shadow block.

We name the path from a failed block to its shadow block via DA-to-PA links and PA-to-DA mappings as the failed block's chain. We further define the path consisting of one DA-to-PA link and its following PA-to-DA mapping as one step. There are two scenarios where a failed block's chain can grow to more than one step. While we aim to minimize access time of failed blocks, we manage to keep all chains to be of only one step.

The first scenario occurs when a fault on the shadow block is detected at the time when the block serves a software-issued write. As shown in Figure 2(c), with this detected fault a new virtual shadow block (P_2) is employed, which is mapped to memory block D_3. Now D_3 becomes D_0's new shadow block with a two-step chain. At this time all involved DAs (D_0, D_2, and D_3) and PAs (P_1 and P_2) are known. WL-Reviver can switch two failed blocks' (D_0 and D_2) virtual shadow blocks (P_1 and P_2, respectively). In this way, D_0 is only one step away from its shadow block, D_3. D_2 is mutually linked by its virtual shadow block, P_1. We name the mutually linked D_2 and P_1 as a *PA-DA loop*. In this scenario, D_2 does not have a shadow block. However, this is not an issue because with current mapping function D_2 can only be reached via P_2, which is not accessible from the software. When the mapping function is updated and P_2 is mapped to a different memory block, D_2 would have its shadow block. It is straightforward to extend the strategy to maintain a one-step chain when new shadow block fails again. To do this, WL-Reviver only needs to let D_0 point to the last virtual shadow block in the chain.

The second scenario occurs with migration of data from a shadow block into a failed block during a wear-leveling operation (e.g., migration of data from D_2 to D_3 as shown in Figures 2(b) and 3(a).) According to the WL-Reviver's design, the data is actually written into the failed block's shadow block (or D_4 in the example shown in Figure 3(a)), producing a multi-step chain. For example, in Figure 3(a) D_4 is D_0's shadow block and D_0 has a two-step chain. Whenever such a chain of two steps with two failed blocks is formed, WL-Reviver reduces it into a one-step chain by switching the two failed block's virtual shadow blocks. As an example, to reduce the two-step chain in Figure 3(a), WL-Reviver switches the virtual shadow blocks of D_0 and D_3, namely P_1 and P_2, respectively. D_4 is still the shadow block of D_0. But now it can be reached in just one step from D_0, as shown in Figure 3(b). D_3 is mutually linked by its virtual shadow block, P_1, or on a PA-DA loop.

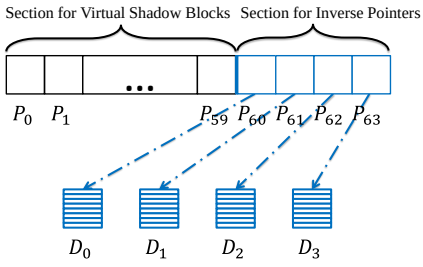

Figure 4. Illustration on where inverse pointers are stored. An acquired OS page of PAs is divided into two sections. The first section, including PAs from P_0 to P_{59}, is used as virtual shadow blocks. The second and the smaller one, including PAs fromP_{60} to P_{63}, is used to indicate memory blocks storing inverse pointers via a PA-DA mapping function.

At the time when a two-step chain is formed, WL-Reviver knows only blocks following the second failed block in the chain but does not know blocks preceding it. To switch two

failed block's virtual shadow blocks in a two-step chain, WL-Reviver needs to know all the blocks. In the example shown in Figure 3(a), WL-Reviver needs to know not only P_2 and D_4 but also P_1 and D_0. Because a mapping function in any wear-leveling scheme is a one-to-one mapping function, there always exists its inverse function allowing us to derive the PA that is mapped to a given DA. In the example, as $D_3 = F_3(P_1)$ WL-Reviver can know P_1 by $P_1 = F_3^{-1}(D_3)$. To know the first failed block, WL-Reviver needs to record an inverse pointer from virtual shadow block to its corresponding failed block. The challenge is where to store the inverse pointer. WL-Reviver does not pre-reserve memory blocks for this purpose. Instead, it uses a method similar to that for acquiring shadow blocks. Once the OS discontinues its use of a new page in response to PCM's access exception, WL-Reviver designates last several PAs in the page as inverse pointer section. The memory blocks mapped to by the PAs are used to store inverse pointers corresponding to the pointers from failed blocks to virtual shadow blocks in the page[2]. Thus, the number of inverse pointers stored in the page is equal to the number of virtual shadow blocks in the page. As an example, for a 64B memory block a 4KB page can accommodate at most 64 virtual shadow blocks. For a 32bit pointer, a memory block can hold 16 inverse pointers. As shown in Figure 4, the first 60 PAs, from PA_0 to PA_{59}, can be used as virtual shadow blocks and the last four PAs, from PA_{60} to PA_{63}, are mapped to memory blocks for storing inverse pointers. When the page of PAs is acquired, the two registers recording the current and last available PAs for virtual shadow blocks are set as PA_0 to PA_{59}, respectively. Compared to the regular data, these pointers are way much less frequently updated, and the blocks storing the pointers are much less worn and less prone to failures. Even in very rare cases where the pointers are lost, they can be rebuilt by scanning the entire PCM.

To guarantee the correctness of WL-Reviver with the aforementioned linkage between various blocks and addresses, we demonstrate the below three statements are true.

Theorem 1. *In WL-Reviver, any software-accessible failed block is backed up by a healthy shadow block.*

Proof. In WL-Reviver, a request presents its access address in the form of a PA (say P_0), which is mapped to a failed block at a DA address (say D_0). Assume D_0's virtual shadow block is P_1. P_1 must not be P_0 because P_0 is software accessible but P_1 is not. Because a one-to-one PA-to-DA mapping function is used, P_1 must be mapped to a block, D_1, that is different from D_0. D_1 must be a block that is considered as healthy at the time of the access. Otherwise, D_1 would be a block of a known failure and store a pointer to a virtual shadow block. If this were true, this virtual shadow block must not be P_1, because P_1 has been D_0's virtual shadow block and a PA can be at most one block's virtual shadow block. This leads to a chain whose length is more than one step. This contradicts the fact that all chains in WL-Reviver are of one step. ■

This theorem ensure that any read and write requests issued by the software to a failed block can be passed to and served at its corresponding shadow block.

Theorem 2. *In WL-Reviver, any unlinked PAs in a reserved OS page, including PAs that have not been pointed to by failed blocks in the section for virtual shadow blocks and PAs that have not been used for storing pointers in the section for inverse pointers, are mapped to heathy blocks, either directly or indirectly.*

Proof. Suppose an unused PA (say P_0) is mapped to memory block D_0. If D_0 is a block currently known as healthy, P_0 is directly linked to a healthy block. Otherwise, if D_0 is a block of known failure, it must have stored a pointer pointing to PA P_1. Because P_0 is unlinked, P_1 is not P_0, and D_0 is not on a PA-DA loop. Therefore, D_0 must have its shadow block, a block currently known as healthy. In this way, P_0 is indirectly linked to a healthy block. ■

This theorem suggests that as long as WL-Reviver can obtain an unlinked PA, it is guaranteed that it has a memory block currently known as healthy to store a block of data. Note that actually writing data into the memory block may be a failure. WL-Reviver can follow the protocol as described before to handle the newly detected block failure.

In WL-Reviver, the only failed blocks not backed up by shadow blocks are those on PA-DA loops. This is not a concern for accesses from the software because their linked PAs are unaccessible to the software. However, it could be an issue if data possibly need to be migrated into such a failed block for wear leveling. The below theorem eliminates the possibility.

Theorem 3. *In WL-Reviver, a wear-leveling scheme does not migrate data into a memory block on a PA-DA loop.*

Proof. The wear leveling scheme determines the mapping function from a PA to DA regardless of whether the PA is accessible or whether the DA is a failed one. Therefore, the scheme has to assume that a memory block at a DA could store regular data if it is mapped to by a PA that could be accessible to the software, even if its access is canceled later due to block failure. While migration of data from one memory block D_0 to another memory block D_1 would destroy data currently stored in D_1, the scheme must make sure that D_1 is impossible to store regular data. Equivalently the scheme has to make sure that D_1 is not mapped by any PA. Therefore, D_1 is not on a PA-DA loop, because a DA on the loop is mapped to by a PA. ■

According to this theorem, any wear-leveling scheme has to assume a buffer block, either explicitly, such as GapLine in Start-Gap [21], and implicitly, such as data swapping in

[2]We will prove that the memory blocks are ready for storing data, either directly or indirectly.

Security Refresh [22], and the buffer block is not mapped to by any software-accessible PA.

IV. Performance Evaluation

In this section we evaluate efficacy and cost-effectiveness of WL-Reviver as a framework to revive wear-leveling operations discontinued due to failures of the PCM blocks. In the evaluation we'd like to answer two questions. The first is whether it is indeed imperative to revive the wear-leveling mechanism after the occurrence of failures. The second question is how efficiently WL-Reviver keeps a wear-leveling scheme alive with PCM faults. Regarding the first question, there are a number of fault-tolerance solutions that can effectively postpone the timing for the first failure to be exposed to a wear-leveling scheme. The solutions can be represented by PAYG [18], which dynamically allocates metadata for error corrections according to error distribution, and FREE-p [23], which hides block failures using blocks in a pre-reserved space until the space is used up. To allow FREE-p to be compatible with the wear-leveling operations, we have to disallow dynamic reservation of space in the experiments. Regarding the second question, WL-Reviver needs an additional read when a failed block is accessed by the software. We compare WL-Reviver with LLS [12], a design integrating wear leveling schemes with fault-tolerance solutions to keep wear-leveling from being disrupted upon a block failure, in terms of space loss and time overhead.

In the presentation of evaluation results we use only one representative PCM wear-leveling scheme, which is Start-Gap [21], though there can be more such schemes, including Security Refresh [22], because of space constraint. However, this methodology is not an issue. WL-Reviver, as a framework, interacts with any such schemes only via one basic operation, data migration, that is common to any of the schemes. The way for WL-Reviver to perform its operations and its associated efficiency are not subject to any design choice of a particular weal-leveling scheme other than this common operation. Furthermore, WL-Reviver neither compromises nor improves a scheme's weal-leveling efficacy. Instead, it only restores an existent scheme's function.

A. Experimental Setup

we use trace-driven simulations in the evaluation. Memory traces are collected by running a number of commonly used programs, listed in Table I, from benchmark suites PARSEC [5], NAS Parallel Benchmarks(NPB) [6], and SPLASH-2 [7], with Pin [4]. These benchmarks represent various distributions of writes over the memory blocks, which are quantified by CoV (Coefficient of Variation) as shown in the table. A larger CoV value means less uniform write distribution and higher probability of early failures on the PCM. In each experiment a program is assumed to run for multiple times to produce required wear-out effect. In the setup, we assume each PCM cell can sustain 10^8 writes with a normal distribution and a lifetime CoV of 0.2. The memory block is 64B, which is also the cacheline size for the last-level of cache. The OS page size is 4KB. We simulate a 1GB PCM chip.

Name	Description	Benchmark Suite	Write CoV
blackscholes	Option pricing	PARSEC	8.88
streamcluster	Online clustering of an input stream	PARSEC	11.30
swaptions	Pricing of a portfolio of swaptions	PARSEC	13.17
mg	Multi-Grid on communication	NPB	40.87
fft	fast fourier transform	SPLASH-2	13.87
ocean	large-scale ocean movements	SPLASH-2	4.15
radix	integer radix sort	SPLASH-2	5.54
water-spatial	molecular dynamics N-body problem	SPLASH-2	5.44

Table I
SUMMARY OF THE BENCHMARKS.

B. Impact of Wear Leveling on PCM's Reliability

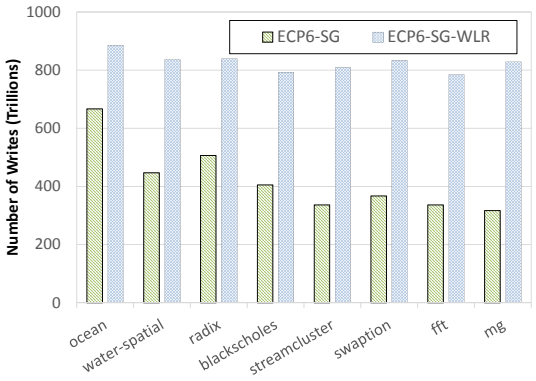

Figure 5. Number of writes required to fail 30% memory blocks of the PCM when different benchmarks and PCM life-extending schemes are used.

We first observe how the wear-leveling scheme (Start-Gap) responds to variation of the benchmarks' write CoVs. In the experiment we choose ECP6, an ECP scheme that can correct up to six errors in a 512b bit group, as the base error correction scheme within each bit group. Figure 5 shows how many writes each of the benchmarks has to issue to fail 30% of memory blocks of the PCM. Here we assume an entire memory is considered unavailable when it loses 30% of its space. In this sense this number of writes represents the PCM's lifetime. As shown, if only Start-Gap is used with ECP6 (see the "ECP6-SG" bars in the figure) the PCM's lifetime is highly related to benchmarks' write CoVs. A benchmark with a highly biased write distribution, such as *mg* with a CoV of 40.87, causes the first block failure to occur much earlier than that with a more uniform write distribution, such as *ocean* with a CoV of 4.15. Without

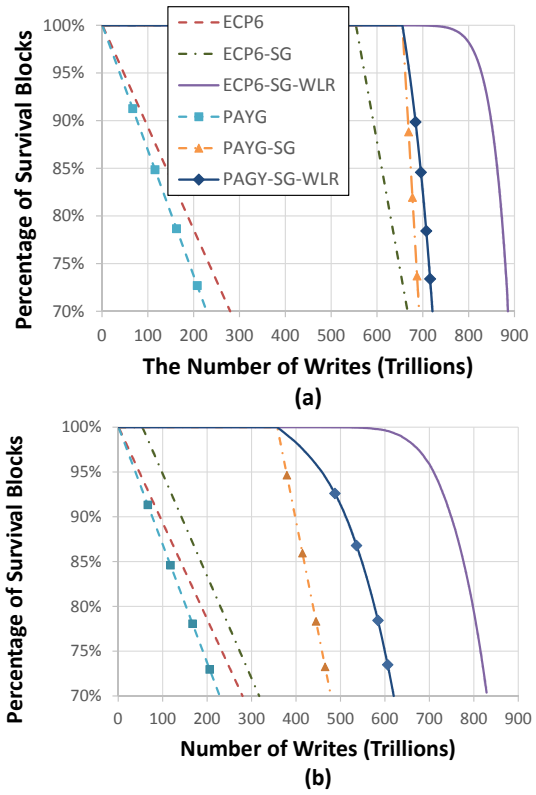

Figure 6. Percentage of survival memory blocks after certain numbers of writes from benchmarks (a) *ocean* and (b) *mg* under different life-extending schemes.

WL-Reviver, the first failure cripples Start-Gap. Once wear leveling is suspended, writes of higher CoV apparently can fail the remaining healthy blocks more quickly, or fewer writes are needed to make the PCM unavailable. With WL-Reviver (see the "ECP6-SG-WLR" bars in the figure) PCM's lifetime is significantly improved for all the benchmarks (from 36% to 325%). In fact, this lifetime shows much less variation across the benchmarks because it is much less sensitive to the write distribution when wear leveling is consistently available. Below we will use *ocean* and *mg* to represent benchmarks with moderately non-uniform and highly non-uniform write distributions, respectively, to present our experiment results.

To see when the first failure occurs and the failure rate after the first failure, we show percentage of memory blocks still alive (or block survival rate) after a certain number of writes from benchmarks *ocean* and *mg* in Figures 6 (a) and (b), respectively. The figures only include results for a survival rate of 70% or higher as a more severely faulted PCM is less likely to be usable in practice. In the comparison, in addition to ECP6 we include PAYG, an error correction scheme making efforts on fending off the first block failure by dynamically allocating error-correction metadata. In the experiments, we adopt a PAYG's default setting, in which ECP1 is its local error correction method and by average 19.5-bit metadata per bit group is used [18]. This space overhead is less than 1/3 of ECP6's overhead, which is 61 bits per bit group.

As shown in the figures, without wear leveling the first failure appears at very early time for both ECP6 and PAYG due to PCM cells' lifetime variation and non-uniform write distribution. With ECP6 and Start-Gap, survival rate can be significantly improved for *ocean*, but not for *mg* (see "ECP6-SG" curves) because *mg*'s highly non-uniform write distribution makes the first failure occur pretty early. As PAYG can effectively postpone the first failure's occurrence, the survival rate is substantially improved (see "PAYG-SG" in Figure 6 (a)). With WL-Reviver, the rate can be further significantly improved. The improvement is much more significant for *mg* than that for *ocean* (see "ECP6-SG-WLR" and "PAYG-SG-WLR" in the figures) as WL-Reviver makes write distribution much less influential on PCM's lifetime. Meanwhile, the improvement of ECP6-SG-WLR over ECP6-SG is much larger than that of PAYG-SG-WLR over PAYG-SG. Note that in the setup PAYG uses only about 1/3 of the metadata used by ECP6 for error protection. With PAYG's dynamical metadata allocation and wear leveling, most of the blocks have been close to depletion of their metadata at the time when the first failure occurs. So the continued wear-leveling enabled by WL-Reviver does not substantially further extend the lifetime. But even so, with highly biased write distribution, such as those with *mg* and malicious attacks, including birthday paradox attack [19], the benefit of WL-Reviver is still substantial.

In general, WL-Reviver not only keeps the survival rate nearly 100% for a longer time period, but also allows the rate to drop at a less radical speed to extend a PCM's lifetime.

C. Comparison to FREE-p on Use of Pre-reserved Space

FREE-p, as designed as a fault-tolerance scheme to incrementally acquire free slots (equivalent to spaces for shadow blocks in WL-Reviver) via OS's support, cannot work together with a wear-leveling scheme, because the free slots' DAs are directly recorded in their respective failed blocks and data migration by the scheme can lead to loss of data stored in the slots. We adapt FREE-p by pre-reserving a certain percentage of PCM space as its remap region to supply free slots. Because these free slots are not visible to the software and out of scope of wear-leveling operations, the adapted FREE-p can work together with the wear-leveling scheme until pre-reserved slots are used up. Figure 7 shows the percentage of user-usable space, or PCM space excluding pre-reserved and failed blocks, with different amount of pre-reserved space for FREE-p, namely, 0%, 5%, 10%, and 15% of PCM space. As expected, as soon as FREE-p uses up the pre-reserved space and has to expose the first failure to Start-Gap, the PCM space is quickly lost because Start-Gap ceases to perform wear leveling. Interestingly, while smaller pre-reservations, such as the 5%

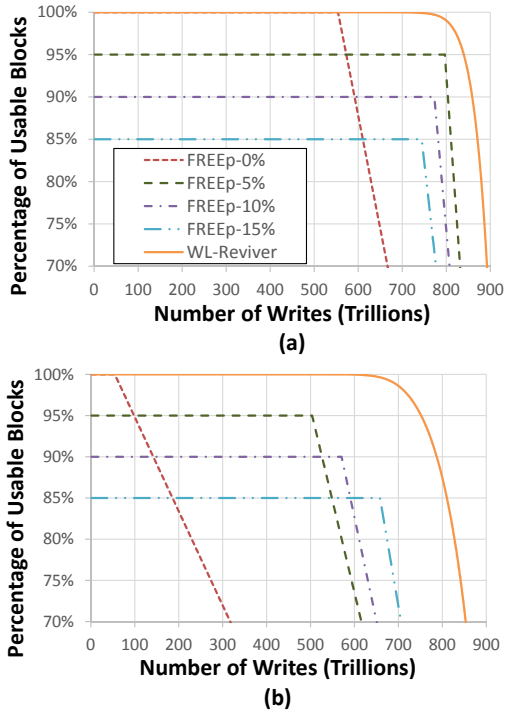

Figure 7. Percentage of user-usable memory blocks after certain numbers of writes from benchmarks (a) *ocean* and (b) *mg* under WL-Reviver and FREE-p with different percentages of pre-reserved space. In all experiments ECP6 and Start-Gap are used.

one, are more effective to postpone occurrence of the first failure for writes with a highly uniform distribution (see benchmark *ocean* in Figure 7 (a), larger pre-reservations, such as the 15% one, are more effective for writes with a biased distribution (see benchmark *mg* in Figure 7 (b)). Pre-reservation has two conflicting effects. It provides free slots to hide failed blocks. In the meantime, it reduces the space for accommodating writes and causes more failed blocks. For writes of highly biased distribution (such as *mg*), a larger number of failures can occur, demanding larger pre-reserved space. For writes of more uniform distribution, it is more beneficial to keep un-reserved space larger to reduce failures. In contrast, WL-Reviver consistently provides significantly more usable blocks. In particular, it makes 100% of the PCM space usable before the first failure.

D. Comparison to LLS on Efficiency

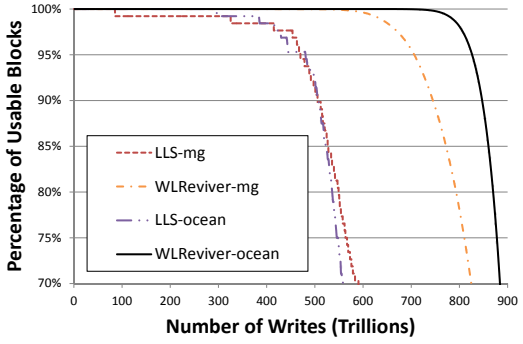

Figure 8. Reduction of software-usable space with ongoing writes. In the experiments ECP6 and Start-Gap are used.

Failure Ratio	Name	Avg. Access Time		Software-Usable Space(%)	
		mg	ocean	mg	ocean
10%	LLS	1.001	1.005	84	85
	WL-Revival	1.001	1.004	89	89
20%	LLS	1.001	1.011	73	73
	WL-Revival	1.003	1.009	79	79
30%	LLS	1.001	1.020	62	63
	WL-Revival	1.004	1.013	68	69

Table II
AVERAGE PCM ACCESS TIME FOR ONE SOFTWARE-ISSUED REQUEST AND PERCENTAGE OF PCM CAPACITY AVAILABLE FOR SOFTWARE TO USE WHEN DIFFERENT PERCENTAGES OF PCM SPACE FAIL. THE ACCESS TIME IS MEASURED IN NUMBER OF PCM ACCESSES.

As LLS also allows a wear leveling scheme to continue functioning in the face of block failure, in this section we compare WL-Reviver with LLS on how failures in PCM memory would impact the system from the perspective of software, or users of PCM. Figure 8 shows the rate at which software-usable PCM space reduces with ongoing writes. As shown in the figure, though LLS can prevent the precipitous loss of usable space, number of writes the PCM can sustain under LLS is much fewer than that for WL-Reviver. The more uniform write distribution of *ocean* barely helps in this aspect. The major reason is on its modification of address randomization method, which is a component of the PA-DA mapping function adopted in Start-Gap. To eliminate spatial correlation, or spread heavily written blocks uniformly across the PCM space, Start-Gap randomly maps a PA to an intermediate PA. To adapt Start-Gap into the LLS's framework, LLS has to restrict the randomization mapping between the first (or second) half of PA addresses and the second (or first, respectively) half of randomized PAs. This imposed restriction keeps concentrated writes in a region from being fully spread, which makes mapped blocks easier to be worn as data migration causes frequently updated data to migrate into the blocks. Without the need of adapting Start-Gap, WL-Reviver fully keeps its randomization mapping and achieves significantly longer lifetime.

Because there could be one indirection in the access of data on a failed block, WL-Reviver uses two PCM accesses for such an access, one to the failed block and another to its shadow block. However, LLS may need three PCM accesses for data on a failed block, which are to the failed block, a bitmap for calculating location of backup block (equivalent to shadow block in WL-Reviver), and the backup block. Therefore, WL-Reviver has a smaller average access time than LLS. In the LLS paper [12] LLS has an option of using a cache to remove the extra PCM accesses. To be a fair comparison, we configure a 32KB cache for both LLS and WL-reviver, a cache size in a proportion of the PCM capacity suggested in the LLS paper. As shown in Table II, because of very high cache hit ratio both LLS and WL-Reviver achieve the almost optimal average access time, which is one PCM access for each software-issued PCM access request. Table II

also shows the space available to the software when a certain percentage of blocks fail. As shown, WL-Reviver makes almost all of the un-failed blocks usable to the software. However, LLS consistently has smaller amount of software-usable space than WL-Reviver because of its less flexible and less inefficient use of reserved space, as explained in Section II.

V. Conclusion

We propose the design of WL-Reviver, a framework that revives any PCM wear-leveling scheme currently ceasing to work once the first failed block appears. Considering that a wear-leveling mechanism is an indispensable component that has to be continuously functioning in any PCM device, WL-Reviver hides failed blocks with shadow blocks in an efficient manner so the mechanism is always available. Recognizing wear leveling on PCM is a critical issue and more optimized schemes addressing the issue would be proposed, we design WL-Reviver as a framework without requiring any adaptations of the schemes. Meanwhile, relying on modifications of OS to enable wear leveling can become a barrier for widespread adoption of the PCM technique. Accordingly, WL-Reviver is designed to require no additional OS supports by leveraging a well accepted practice, which is that the OS does not access a page once it is notified of an access failure on the page. By guaranteeing that any access of failed block can be served with only one indirection, WL-Reviver makes PCM's average access time minimally deteriorated even with substantial failures.

VI. Acknowledgments

We would like to thank the anonymous reviewers for their comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61327902, 60925006), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), Shanghai Key Laboratory of Scalable Computing and Systems, Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, Huawei Technologies Co. Ltd., and Tsinghua University Initiative Scientific Research Program.

References

[1] Micron Announces Availability of Phase Change Memory for Mobile Devices. In *http://investors.micron.com/releasedetail.cfm?ReleaseID=692563*, July, 2012.

[2] Samsung Ships Industry's First Multi-chip Package with a PRAM Chip for Handsets. In *http://www.samsung.com/us/aboutsamsung/news/newsIrRead.do?news_ctgry=irnewsrelease&news_seq=18828* April, 2010.

[3] Technology Brief: Avoiding Server Downtime from Hardware Errors in System Memory with HP Memory Quarantine. In *http://h20000.www2.hp.com/bc/docs/support/SupportManual/c03179047/c03179047.pdf*

[4] Pin - A Dynamic Binary Instrumentation Tool. In *http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool*

[5] The PARSEC Benchmark Suite. In *http://parsec.cs.princeton.edu/parsec3-doc.htm*

[6] NAS Parallel Benchmarks (NPB). In *http://www.nas.nasa.gov/publications/npb.html*

[7] SPLASH-2 Benchmarks Suite. In *http://www.capsl.udel.edu/splash/*

[8] R. Azevedo, J. D. Davis, K. Strauss, and P. Gopalan, M. Manasse, and S. Yekhanin. "Zombie Memory: Extending Memory Lifetime by Reviving Dead Blocks", In *ISCA*, June 2013.

[9] J. Condit, E. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. "Better I/O Through Byte-Addressable, Persistent Memory", In *SOSP*,October 2009.

[10] Emerging research devices. In *ITRS*, 2011.

[11] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda. "Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories", In *ASPLOS*, 2010.

[12] L. Jiang, Y. Du, B. Zhao, Y. Zhang, B.R. Childers, and J. Yang," Hardware-Assisted Cooperative Integration of Wear-Leveling and Salvaging for Phase Change Memory," In *ACM Transactions on Architecture and Code Optimization*, Vol. 10 Issue 2, 2013.

[13] K. Kim. "Technology for sub-50nm DRAM and NAND flash manufacturing", In *International Electron Devices Meeting*, 2005.

[14] B. Lee, E. Ipek, O. Mutlu, and D. Burger. "Architecting Phase-Change Memory as a Scalable DRAM Alternative", In *ISCA*, June 2009.

[15] R. Melhem, R, R. Maddah, and S. Cho. "RDIS: A Recursively Defined Invertible Set Scheme to Tolerate Multiple Stuck-At Faults in Resistive Memory", In *DSN*, June 2012.

[16] M. K. Qureshi, J. Karidis, M. Fraceschini, V. Srinivasan, L. Lastras, and B. Abali. "Enhancing Lifetime and Security of Phase Change Memories via Start-Gap Wear Leveling", In *MICRO*, 2009.

[17] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. "Scalable high performance main memory system using phase-change memory technology". In *ISCA*, 2009.

[18] M. K. Qureshi. "Pay-As-You-Go: Low-Overhead Hard-Error Correction for Phase Change Memories", In *MICRO*, December, 2011.

[19] A. Seznec. "A Phase Change Memory as a Secure Main Memory", In *IEEE Computer Architecture Letters*, Vol. 9, Issue 1, 2010.

[20] S. Schechter, G. Loh, K. Strauss, and D. Burger. "Use ECP, not ECC, for Hard Failures in Resistive Memories", In *ISCA*, June 2010.

[21] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H. S. Lee. "SAFER: Stuck-At-Fault Error Recovery for Memories", In *MICRO*, 2010.

[22] N. H. Seong, D. H. Woo, and H.-H. S. Lee. "Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping", In *ISCA*, 2010.

[23] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, "FREE-p: Protecting Non-volatile Memory against both Hard and Soft Errors." In *HPCA*, 2011.

[24] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology", In *ISCA*, 2009.

[25] W. Zhang, and T. Li. "Characterizing and mitigating the impact of process variations on phase change based memory systems." In *MICRO*, 2009.