Non-Volatile Memory Based Page Swapping for Building High-Performance Mobile Devices

Duo Liu, *Member, IEEE*, Kan Zhong, Xiao Zhu, Yang Li, Lingbo Long, and Zili Shao

Abstract—Smartphones are getting increasingly high-performance with advances in mobile processors and larger main memories to support feature-rich applications. However, the storage subsystem has always been a prohibitive factor that slows down the pace of reaching even higher performance while maintaining good user experience. Despite today's smartphones are equipped with larger-than-ever main memories, they consume more energy and still run out of memory. But the slow NAND flash based storage vetoes the possibility of swapping—an important technique to extend main memory—and leaves a system that constantly terminates user applications under memory pressure. In this paper, we propose *NVM-Swap* by revisiting swapping for smartphones with fast, byte-addressable, non-volatile memory (NVM) technologies. Instead of using flash, we build the swap area with NVM, to allow high performance without sacrificing user experience. NVM-Swap supports *Lazy Swap-in*, which can reduce memory copy operations by giving the swapped out pages a second chance to stay in byte-addressable NVM backed swap area. To avoid fast worn-out of certain NVM, we also propose *Heap-Wear*, a wear leveling algorithm that distributes writes in NVM more evenly. Evaluation results based on the Google Nexus 5 smartphone show that our solution can effectively enhance smartphone performance and achieve better wear-leveling of NVM.

Index Terms—Smartphone, swapping, non-volatile memory, application relaunching delay

1 Introduction

Smartphones are not just phones any more as more functionality being integrated. With features such as installing third-party applications and multi-tasking, today's smartphones offer unprecedented user experience. However, this comes with a price: the richer functionality an application can provide, the more demanding it is on computing, memory and storage resources. Fast mobile processors and large low power main memories have always been heralding the direction of satisfying such demands and the trend is likely to continue. Moreover, since applications may launch multiple times in smartphones, modern mobile OS, such as Android tends to cache as many as possible applications in memory to accelerate the application relaunching, making the application switching seamless for smartphone users.

However, caching applications makes the memory capacity a crucial factor for smartphone performance. Due to the space and power budget constraint, the memory capacity in smartphone is usually limited and only moderate number of applications can be cached. To reclaim memory space, applications are constantly terminated ("killed") when memory is under pressure, making the terminated ("killed") applications take a long time to restore to their previous states when they are launched again or switched to foreground. Moreover, due to the increasingly complexity of modern applications, it is difficult for applications to restore to the exactly same previous states when they are switched back. Therefore, mobile applications cannot be simply killed when the system's memory is under pressure.

Page swapping is an effective way to extend main memory by enabling the ability of writing inactive pages to secondary storage. With swapping, applications can reside in memory by writing part of their data to swap area. To show the effectiveness of swapping, in Fig. 1, we compare the number of process terminations between swap disabled and flash backed swap enabled when running a mixture of applications for 30 minutes in a Google Nexus 5 smartphone.[1] With different memory capacities on the *X*-axis, swapping can help reduce around 66 to 91 percent of process terminations, greatly lowering the chance of an application being terminated when the system is short for memory.

Nevertheless, there are still several issues that make page swapping disabled by default in Android Linux kernel. First, the NAND flash memory has been evolving very slowly and its speed failing to catch up with the speed of mobile processors. Therefore, loading pages from flash backed swap area is usausly slow. Second, current most

- *D. Liu, K. Zhong, X. Zhu, and Y. Li are with the College of Computer Science, Chongqing University, No. 174, Shazhengjie, Shapingba, Chongqing 400044, China, and the Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, Chongqing 400044, China. E-mail: {liuduo, kzhong1991}@cqu.edu.cn.*
- *L. Long is with the College of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing 400065, China. E-mail: longlb@cqupt.edu.cn.*
- *Z. Shao is with the Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong. E-mail: cszlshao@comp.polyu.edu.hk.*

Manuscript received 24 July 2016; revised 20 Apr. 2017; accepted 11 May 2017. Date of publication 1 June 2017; date of current version 16 Oct. 2017. (Corresponding author: Duo Liu.)
Recommended for acceptance by G. Heiser.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2711620

1. Specifications at `http://www.google.com/nexus/5`

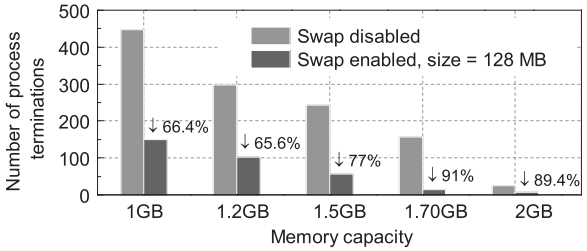


Fig. 1. Comparison of process terminations between swap disabled and swap enabled when running various applications in a Google Nexus 5 smartphone (Android 4.4) for 30 minutes. Swapping can reduce around 66 to 91 percent of process terminations.

mobile devices adopt eMMC[2] as their storage system due to its low-cost and high-density. Since eMMC devices have limited bandwidth, loading/writing pages from/to flash backed swap area can lead to I/O contentions, and thus potentially slowing down the normal I/O operations. Finally, frequent page swap ins/outs potentially exacerbate wear-out and increase garbage collection overhead of flash memory. Therefore, we argue that swapping needs to be redesigned with emerging memory technologies.

Emerging byte-addressable, non-volatile memory (NVM) technologies such as phase change memory (PCM) [2], spin-transfer torque RAM (STT-RAM) [3] and memristor [4] are changing this situation. Compared to NAND flash, these NVM products offer not only faster (near-DRAM) performance, but also larger erase cycles. Therefore, we propose to re-adopt swapping with the help of NVM. Instead of using flash memory, we build the swap area with NVM, to avoid constant process termination while maintaining good performance. Unlike flash memory, NVM is byte-addressable and can be manufactured as DIMMs to be placed on the memory bus, available to `load` and `store` instructions and thus removing all the overhead induced by the storage stack. Such combination of high-performance and low energy consumption makes NVM an attractive candidate for swapping in smartphones.

In this paper, we revisit swapping in smartphones and propose *NVM-Swap*, an NVM-based approach to build high-performance swapping without sacrificing user experience. NVM-Swap re-adopts swapping in smartphones by augmenting the DRAM with NVM and using it as the swap area, thus extending memory capacity. To avoid excessive and unnecessary overheads from the block-based storage stack, we utilize NVM's byte-addressability and attach it directly to the memory bus, so that we can access it via a simple memory interface, instead of building a similar block device found in Compcache [5].

To further reduce page thrashing and swap-in overheads, which involve memory copy operations between DRAM and NVM, we propose *Lazy Swap-in* to allow pages in NVM be read directly. When a swapped out page is accessed, instead of swapping that page to DRAM, Lazy Swap-in gives that page a second chance to stay in NVM by setting up the page table mapping and returning the page in NVM directly. Only when the page is accessed again within a defined time interval, actual swap-in is involved to copy that page back to DRAM. However, due to the high write cost of NVM, we do not allow write pages in NVM directly. When a write happens to a page in NVM, we immediately swap in the page by copying it back to DRAM.

Despite these advantages, NVM is not perfect. In particular, most of them are vulnerable to unbalanced writes (e.g., PCM has limited number of programming cycles of 10^8-10^9 [2]). Pages that are swapped out could hit arbitrary swap slots in an unbalanced manner, which shortens the lifetime of NVM, making NVM-Swap unpractical. To eliminate the negative effect brought by this issue, we propose a heap-based wear leveling technique called *Heap-Wear*, which evenly distributes pages across the whole NVM space. As mentioned earlier, a lot of wear leveling techniques have been proposed to mitigate the endurance problem of NVM [6], [7], [8], [9], [10]. However, different from those existing work, which usually depends on data comparison write (DCW) [7], [11] or segment switching [12], Heap-Wear uses a space-efficient heap structure with swap-specific information to handle write requests. We assign each page that is being swapped out a "young" swap slot using the age information maintained in the heap structure. Write requests are then more evenly distributed across all the swap slots, resulting in a more balanced write pattern and a more durable NVM-based swap area.

We implement NVM-Swap in Google Android 4.4.2 based on the Google Nexus 5 smartphone. In this paper, we only focus on performance and endurance issues. We have discussed energy related issues in previous work [13], [14]. Experimental results show that NVM-Swap can avoid terminating most processes and respectively reduce application relaunching delay and application execution time by around 12-46 percent and 14-45 percent when compared to NAND flash-backed swapping. In addition, with Heap-Wear, swap slots are more uniformly written, giving us a durable NVM-Swap.

In summary, we make the following contributions:

- We revisit swapping in smartphones and propose NVM-Swap, which uses emerging NVM for swapping, to extend memory without sacrificing performance;
- We propose Lazy Swap-in to avoid page thrashing and reduce swap-in overheads, furthering improving performance;
- To make NVM-Swap practical, we propose Heap-Wear, which is a space-efficient wear leveling algorithm that can extend the lifetime of NVM.

The remainder of this paper is organized as follows. Section 2 outlines the background on swapping and NVM in smartphones, and gives the motivation. Section 3 details the design of NVM-Swap. Evaluation results are shown in Section 4. We discuss related work and conclude in Sections 5 and 6, respectively.

2 Background and Motivation

In this section, we first give the background on non-volatile memory and swapping in smartphones, then we present our motivation.

2. Embedded Multi Media Card (eMMC), usually comprised of a NAND flash chip and a controller.

2.1 Emerging Byte-Addressable NVM

Non-volatile memory technologies have been discovered by computer architects to replace DRAM and even flash memory because of their low power consumption, high density and byte-addressability. Compared to DRAM, NVM keeps data by changing the physical state of its underlying material without maintaining constant current. One promising candidate is phase change memory [2], which stores data by changing the state of the phase change material (e.g., GST) between amorphous and crystalline. It exhibits longer access latency (80ns-1μs) than DRAM (20-50 ns) and also limited write endurance (around 10^8-10^9 programming cycles) [9], [10]. To overcome these problems, various wear leveling and caching schemes have been proposed [6], [7], [8], [9], [10]. It is widely accepted that the mature PCM products will feature a small DRAM/SRAM cache to provide both fast data access and reasonable lifetimes. However, these wear-leveling techniques bring extra hardware overheads and are not the perfect choices in the context of swapping, especially in resources constrained mobile devices. We demonstrate the write distribution of a DRAM-based swap area with Android default swap area management algorithm (in Fig. 9) and find that the endurance of NVRAM is a must solving issue to make a practical NVM-based swap area.

Other promising NVM technologies include spin-transfer torque RAM [3] and memristor [4]. Both STT-RAM and memristor have the potential of providing at least DRAM performance. However, their performance numbers are still in constant change. For example, the reported latency for memristor ranged from hundreds of picoseconds to tens of nanoseconds [15]. Because of its low latency (10-25 ns [3]) and non-volatility, STT-RAM can be used to build both on-chip cache and main memory [16]. These newer NVMs also have better endurance behavior than both PCM and NAND flash. Should they become mature in the future, a unified NVM system that features both NVM-based main memory and swap area would become possible for even higher performance.

We argue that in general these byte-addressable NVMs are suitable for building a high-performance swapping device in smartphones. Despite different internals, they could all be built as DIMMs. They also share such properties as near-DRAM performance, better endurance than flash and low power consumption. Thus, we do not target at any specific type of NVM in this paper, nor do we rely on any specific timing or endurance constraints to design our system.

2.2 Swapping

Swapping is an effective way to extend memory by borrowing space from I/O devices (e.g., flash and disks) in modern operating systems [17]. Originally, swapping refers to moving all of the memory pages of a process to storage to make space for others. Paging, on the other hand, refers to moving just pages of memory. With virtual memory based on pages, swapping and paging have become synonyms. However, swapping still means processes could be swapped in and out in units of non-contiguous pages. To correctly describe the proposed technique, we therefore use the term "swapping" in this paper. When the system is under memory pressure and finds itself difficult to satisfy memory allocation requests, the OS will choose to write ("swap") some inactive memory pages to some I/O devices (the "swap area") and allocate these page frames to the requesting applications or OS components. Because of the scarcity of DRAM and the abundance of disks and flash memory in capacity, the swap area is usually backed by these two types of devices via a block interface. Pages that are being swapped out must go through all the storage stack to be written in the storage medium. Therefore, the underlying storage becomes an important factor on swap performance. For better swap performance, the server market has seen high-performance flash memory based solutions, such as FlashVM [18]. However, the situation in mobile devices is different, despite they can also use NAND flash memory as the swap area.

```
WordPress-PROFILING: Startup: begin
WordPress-PROFILING: Startup:      886 ms, WPLaunchActivity.onCreate
WordPress-PROFILING: Startup:      234 ms, WPMainActivity.onCreate
WordPress-PROFILING: Startup:      558 ms, WPMainActivity.onResume
WordPress-PROFILING: Startup: end, total time = 1678 ms
```

Fig. 2. The profiling output of Android logcat in measuring application launch time.

Different from enterprise-level flash-based solid state drives, mobile NAND flash storage performs notoriously worse [19], [20]. Therefore, swapping is usually disabled in smartphones to avoid sacrificing performance. Take Android as an example, instead of using a flash-based swap area, it by defaults uses a so called "low memory killer" that constantly monitors the system and terminates processes to make room for incoming memory requests. Since Android 4.4, Android allows "swap to zRAM" (a.k.a "Compcache") [5], which compresses memory pages and saves them in a dedicated RAM disk to save memory space. However, smartphones do not have "unlimited" energy like general purpose systems. Usually they are powered by batteries with a capacity of only around 2000-3000 mAh due to various reasons, such as size and weight. Moreover, most smartphones do not fully close an application (thus resources not released) when it is switched to the background for faster switch-back. Various daemons also keep running all the time to provide useful information for the user (e.g., notifications for new instant messages). These features make battery capacity scarce in smartphone, and yet "swap to zRAM" will worsen the situation: compression needs more computation power from the CPU, while the RAM disk requires even larger memory. Both requirements will inevitably lead to more energy consumption, making it even faster to drain the battery which is already "always short on capacity". Therefore, zRAM is usually disabled in most smartphones.

2.3 Motivation

To see how swapping will affect the performance of applications in smartphones. We measure the application relaunching time when the application is under different states. To measure the application relaunching time, we select an open source application—WordPress,[3] and analyse its launch process. We find two activities are required during its launching, thus we measure the launching time by instrumenting the application source code and profiling the launching time use Android logcat. Fig. 2 shows the profiling output of Android logcat when measuring the launching time of WordPress.

3. Code obtained from https://github.com/wordpress-mobile/WordPress-Android

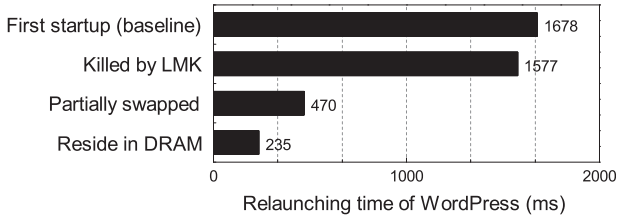


Fig. 3. Relaunching time of WordPress installed on a Nexus 5 smartphone with different application states. The size of the flash backed swap area is 128 MB.

Fig. 3 illustrates the relaunching time of WordPress under different states. As shown in the figure, when the application is killed in background due to insufficient memory space, the relaunching time is as long as its first startup, which is nearly 94 percent of the baseline. In the contrary, when it is not killed and all its data is cached in DRAM, the relaunching time is the shortest, which is only 235 ms. However, with a flash backed swap area, when there is no sufficient memory, inactive applications are swapped to swap area rather than be killed or cached in DRAM. In this case, the relaunching time of WordPress is 470 ms. Compared with the case that WordPress be killed by LMK, the relaunching time of WordPress when it is partially swapped is reduced 75 percent, indicating that swapping can shorten application relaunching time under low memory circumstances. The main reason is that for partially swapped application, only the swapped data need to be reloaded into main memory, which is more efficient compared to forking new process and loading the entire application data if the application is killed.

Nevertheless, as discussed in Section 1, flash backed swapping is not practical in mobile devices due to the performance and lifetime issues. Motivated by the advantages of NVM and benefits brought by swapping—reducing application terminates and accelerating application relaunching, in this paper, we designed a novel swapping mechanism that is backed by emerging fast NVM to extend the memory space.

3 NVM-Based Swapping

We build NVM-Swap by utilizing the byte-addressability and non-volatility of NVM. Instead of managing NVM as a block device and adopting the existing swap infrastructure in modern OSes, we attach NVM directly to the memory bus, eliminating I/O and the whole storage stack overheads. As we have discussed, NVM-Swap features two additional optimizations: (1) Lazy Swap-in which reduce memory copy operations and (2) Heap-Wear which evenly distributes write requests across the whole swap area for enhancing NVM durability. In the rest of this section, we first highlight the architecture of our system by comparing it with the existing approach. We then discuss Lazy Swap-in and Heap-Wear in detail. Finally, we introduce the implementation of NVM-Swap in Android Linux kernel.

3.1 Architecture

In our previous work [13], we proposed the in-memory paging architecture, in which NVM is used as the swap area and shares part of the physical address space with DRAM.

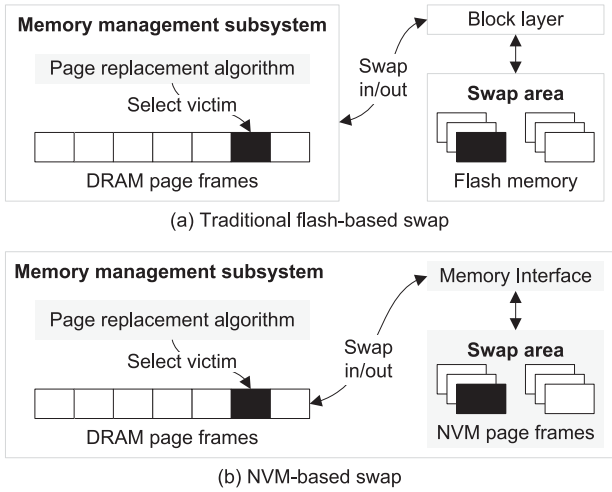


Fig. 4. Comparison between traditional flash-based swapping and NVM-Swap. (a) A flash-based swap area requires swapping requests go through the whole storage stack. (b) NVM-Swap uses memory interface to access the swap area, which is backed by NVM (attached to the memory interface). Note that in NVM-Swap, the swap area becomes part of the memory management subsystem and requires no I/O operations.

The architecture of using emerging byte-addressable NVM as swap in smartphones differs from using a traditional I/O device based swap (e.g., flash). We highlight this difference in Fig. 4. In Fig. 4a we show how a traditional flash-based swapping approach works in smartphones. As part of the memory management subsystem, the page replacement algorithm scans DRAM page frames for a victim when memory is under pressure. The victim is then evicted from DRAM and written to the swap area, via the block layer, which is part of the storage stack. Note that swap area is backed by flash memory, which is attached to the I/O bus. Therefore, the swapping in/out processes must go through the whole storage stack.

In an NVM-based swap area, as shown in Fig. 4b, swapping will not involve any I/O operations as we attach the NVM to the memory bus. Besides DRAM, the OS also sees an NVM area which shares the same physical address space with DRAM. We focus on the software side in this paper, but expect the memory controller to report to the OS on the partitioning of the physical address space, such that the OS could know which address space range belongs to DRAM and NVM upon start. The OS can then manage NVM via page tables. In our system, we still use DRAM as main memory and flash as storage. Note that in Fig. 4b the swap area is accessed via a memory interface and becomes part of the memory management subsystem. When the system is under memory pressure and finds it difficult to allocate more memory, the page frame reclaim algorithm will try to pick victim page frames from running applications and swap them out to NVM. The page replacement algorithm works in the same way as before. Victim pages are directly written to the swap area through simple `memcpy` calls, instead of via expensive I/O transfers.

Compared to hybrid memories, which treat NVM as part of main memory, swapping using NVM effectively re-uses the infrastructure that already existed in mobile OSes and is much less intrusive to implement. With NVM-Swap, swapping becomes pure memory operations, instead of I/O

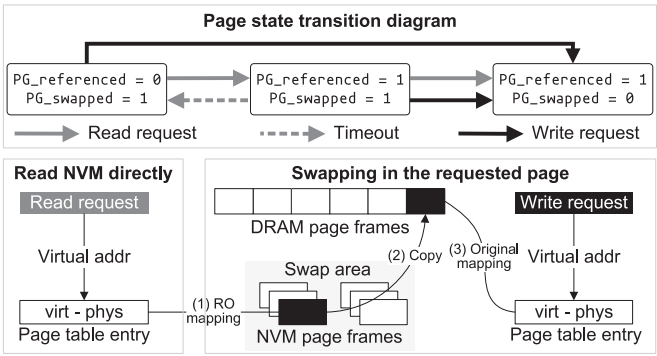


Fig. 5. Lazy Swap-in. (1) For the first access, the page fault handler sets up read-only (RO) permission for the requested page in NVM and the page is read directly, thus no data are transferred between NVM and DRAM, avoiding memory copying operations; (2-3) Two consecutive reads or any write attempt to the page makes the page be copied from NVM back to DRAM, and then the corresponding page table entries are updated by the OS.

requests, eliminating the need to go through all the storage stack to access data in the swap area, and allowing better utilization of NVM's high performance. Moreover, the consistency and persistence concerns of using NVM found in hybrid memory or NVM-specific library proposals [21] do not exist in NVM-Swap, since swap area is discarded after use, and will be re-initialized when it is setup.

3.2 Lazy Swap-In

Though avoiding the storage stack in swapping and the high-performance nature of NVM eliminate most overhead, we find that page thrashing may exits and incurs extra overheads: pages that are swapped back from NVM may exhibit infrequent access pattern and become inactive, consequently be swapped to NVM again. For instance, suppose that a page in NVM is accessed by a user application, which will trigger a page fault and in the page fault handler, the kernel reads the swap area to fetch the requested page frame, sets up new page table mappings and return to the user application. However, the page may only be read once and then becomes inactive, making the page be swapped to swap area again in page frame reclaiming. In this situation, the whole procedure involves one swap-in (i.e., one NVM page read and one DRAM write) and one swap-out (i.e., one DRAM read and one NVM write). Fortunately, NVM's byte-addressablity provides a great opportunity to reduce memory copy operations for read infrequent pages, since the requested memory page already resides in memory—the NVM—though in a different region (the swap area).

To solve page thrashing problem, we propose Lazy Swap-in, which allows pages in NVM be read directly without copying them back to DRAM and gives pages a second chance to stay in NVM. As shown in Fig. 5, referenced bit (i.e., `PG_referenced`) is used to indicate whether a page is accessed or not and the swapped bit (i.e., `PG_swapped`) is used to indicate whether the page is in NVM swap area. For swapped out pages, the referenced bit is set to '0' and the swapped bit is set to '1', when accessing a page that was swapped out, Lazy Swap-in directly sets up page table mappings for the requested page and updates the reference bit to '1'. The page then becomes available directly to the user space. However, we do not allow consecutive reads to NVM pages since most NVMs, such as PCM, exhibit higher read latency than DRAM [22]. Therefore, as shown in Fig. 5, if the NVM page (i.e., page with referenced bit set to '1') is read again within a time interval T, we do actual swap-in—copy the page from NVM to DRAM and update the mapping with the page's original access permission (i.e., permission before the page was swapped out). Otherwise, the NVM page's referenced bit is set to '0'. For the time interval T, a small T tends to swap in pages from NVM swap area, and thus will reduce the average read latency but increases the memory copy operations. In contrast, a large T tends to keep pages stay in NVM swap area, and thus will reduce the memory copy operations but increases the average read latency. To strike a balance between read latency and memory copy overheads, we currently set the time interval T to 10 seconds, which is usually the time interval for the pageout daemon (i.e., `kswapd`) to wake up to swap out pages periodically.

Due to the high write cost and limited endurance of NVM, we do not allow any direct write to the NVM page which was mapped by Lazy Swap-in via page table. As shown in Fig. 5, by marking the NVM page "read-only" in its page table entry, any modification to the NVM page "swapped in" by Lazy Swap-in will then trigger a page fault and we perform actual swap-in in the page fault handler. Compared to the traditional approach, Lazy Swap-in speed up the swap-in process for pages that are read-only and infrequently accessed by avoiding transferring a whole page between NVM and DRAM. The only overhead left is one write for the page table entry, which only involves writing a 32-bit/64-bit entry in most today's ARM-based smartphones. With Lazy Swap-in, page thrashing can also be reduced by giving pages a second chance to stay in NVM swap area.

3.3 Heap-Wear

Most emerging NVMs are vulnerable to unbalanced writes. For example, a PCM cell can only sustain 10^8-10^9 programming cycles. However, pages that are being swapped out could hit arbitrary swap slots in an unbalanced manner by default as the traditional swap architecture assumes a disk-based swap area. Unbalanced writes shorten the lifetime of NVM and could render NVM-Swap unpractical. To solve this problem, we propose Heap-Wear, a space-efficient wear leveling algorithm. It extends the lifetime of NVM by maintaining the age information of swap slots and only allocating young slots to store swapped-out pages. Heap-Wear categorizes swap slots into three types: `young`, `old` and `zombie` slot. Young and old slots are slots that are used (written) for fewer and more times, respectively. Zombie slots store valid pages that never or infrequently updated. Consequently, zombie slots stay young while other slots may be written frequently and become older, leading to an unbalanced erase pattern and unpractical NVM-Swap.

To solve this problem, Heap-Wear always chooses a young slot as the candidate for swapped-out pages, with the help of a *free slot list* and a heap structure called *min-heap*. The free slot list is a doubly linked list consisting of all unused slots, while min-heap maintains the age information of all the swap slots. Fig. 6 illustrates the structure of these two components. Note that the head slot in the free slot list is always preferred as the first choice, and the top of min-heap always records the age of the youngest slot. Once a free slot in the head is selected for

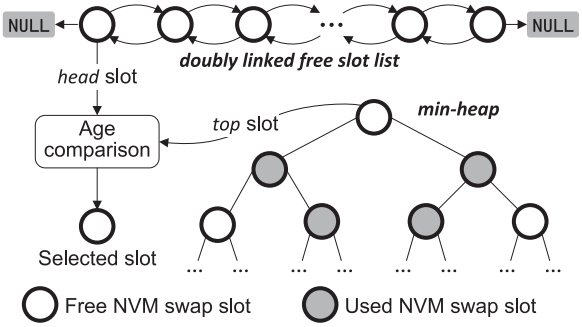


Fig. 6. The free slot list and min-heap data structure. Doubly linked list only maintains the free NVM slots while the min-heap maintains all the NVM slots, including used slots and free slots.

storing a swapped-out page, we compare the age of the candidate to that of the top in the min-heap. If the age difference is greater than a predefined threshold TH (i.e., the selected slot is older), a slot exchange operation is performed (assume the top slot currently holds a valid page), to exchange the page in the selected candidate slot with that of the top slot. Otherwise, write the page to the top slot directly. In contrast, if the selected slot is younger than the top slot, the swapped-out page is written to the selected slot. The detailed process is depicted in Algorithm 1.

Algorithm 1. Heap-Wear Algorithm

Input: $list$: free slot list, $heap$: slot min-heap, TH: slot exchange threshold;
Output: $slot$: the selected slot;

```
1: head ← list.head;
2: top ← heap.top;
3: if head.age − top.age > TH then
     /* Perform a slot exchange.                    */
4:   if top is used then
5:     Copy the page in top slot to head slot;
6:     head.age ← head.age + 1;
7:     Adjust heap to maintain the heap property;
8:     Remove head slot from list;
9:     Update the slot mapping table;
10:    if top is referenced via page table then
11:      Update all the PTEs which referenced top;
12:  else
13:    Remove top form list;
     /* Put the victim page to top slot.            */
14:  top.age ← top.age + 1;
15:  Adjust heap to maintain the heap property;
16:  slot ← top;
17: else
     /* Put the victim page to head slot.           */
18:  Remove head form list;
19:  head.age ← head.age + 1;
20:  Adjust heap to maintain the heap property;
21:  slot ← head;
22: return slot;
```

In Heap-Wear, if a page is swapped out, the corresponding PTE will record the information of that page. After exchanging the page between two slots, we employ a mapping table to reflect this change. Fig. 7 shows an example of exchanging pages between two slots. Initially, slot *s1* is on top of the min-heap and holds a swapped-out page, which

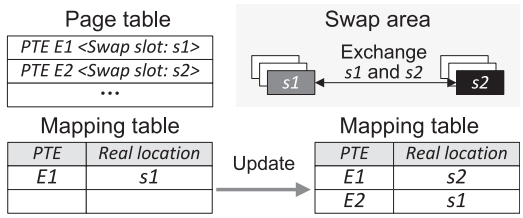


Fig. 7. Example of Heap-Wear. After exchanging the pages in *s1* and *s2*, the corresponding entries of mapping table are updated.

corresponds to PTE *E1*, that means slot *s1* is not referenced via page table. Then to allocate a slot for the victim page, which corresponds to PTE *E2*, slot *s2* is selected from free slot list. Comparing the ages of slot *s1* and slot *s2* finds the slot exchange condition is satisfied. Therefore, as shown, pages in slots *s1* and *s2* are exchanged, with corresponding entries in the mapping table are re-mapped (i.e., *E1* maps to *s2* and *E2* maps to *s1*). Otherwise, if slot *s1* is referenced via page table (i.e., mapped by Lazy Swap-in), we not only perform slot exchanging but also update all the PTEs that are pointing to slot *s1*. We make all new PTEs reference slot *s2* through reverse mapping—a mapping between a physical page and the PTEs of all processes that use the page [17].

Copying a page in NVM induces constant cost, leading to an $O(lgN)$ complexity of Heap-Wear, which is actually for maintaining the min-heap. A 128 MB swap area only needs no more than 1 MB main memory space to store the data structure. With only one extra NVM page copy when exchanging two NVM swap slots, Heap-Wear can avoid zombie slots efficiently. Note that the age counter of each slot is stored at the beginning of the NVM swap area, and the NVM memory space is pre-computed. When the swap area is activated, the age counters are loaded to main memory, and only synchronized periodically to avoid wearing out the underlying NVM cells. Various techniques have been proposed to solve the crash consistency [21], [23], [24], [25], [26] problem. In this paper, we leverage the concept of shadow paging to achieve crash consistency of age counters. In Heap-Wear, we use two age counter arrays—primary array and shadow array. When writing back age counters, the age counters are first written to the shadow age counter array. Only when the write is completed without crash, we simply change the age array pointer to reference the shadow array through atomic operation and make it becomes the primary array. Thus, the current primary array will be used as shadow array in the next synchronization. With one extra age counter array overhead, which is very small compared to the swap area size, the crash consistency of age counters can be achieved.

4 Evaluation

We implement the prototype of NVM-Swap in Google Android 4.4 with the kernel version ARM Linux 3.4 for the Google Nexus 5 smartphone. There are totally around 1,000 Lines of Codes (LOCs) coding efforts within the Linux kernel. In the Linux kennel, we introduce a new memory zone and serve it as the NVM swap area for allocating NVM page frames. The swap subsystem is modified to manage all the pages in this NVM zone with the help of Heap-Wear. Traditional swap in/out operations—flash reads/writes—are replaced with pure memory copy operations. To realize

Lazy Swap-in, we modify the page fault handler to support read the pages in NVM swap area directly. In the rest of this section, we first introduce the experimental setup and workloads applications, then we present the metrics and methodology. Finally, we discuss the experimental results.

4.1 Experimental Setup

We run all experiments with a Google Nexus 5 smartphone. It features Qualcomm Snapdragon 800 processor clocked at 2.26 GHz and 2 GB DRAM as main memory. Our Nexus 5 model has 16 GB internal flash storage. We connect the smartphone to a desktop PC and use the Android debug bridge (`adb`) in the Android SDK to communicate with it. For all experiments, we reboot the phone and wait for a few minutes to ensure the device is idle. During the experiments, the phone is always connected to a charger to make sure it is working in its full performance capability.

In our experiment, we use a DRAM partition to simulate the NVM swap area since NVM products are not yet widely available. Although our system does not rely on any specific type of NVM and can be easily deployed in different NVM-based systems, in the evaluation, we assume PCM is used as the swap area. More specifically, we use Micron LPDDR2-PCM [27], which is a 45 nm technology based PCM product with clock frequency up to 400 MHz. Compared with the DRAM of Nexus 5, which is LPDDR3-SDRAM, the read and write latency of LPDDR2-PCM is around 2X and 12X slower, respectively [14]. Therefore, to simulate the access latency of PCM, for each swap-in operation (i.e., PCM read), we read DRAM 2 times, and for each swap-out operation (i.e., PCM write), we write DRAM 12 times. We believe that future PCM products will provide near-DRAM performance.

We test three different swapping shcemes: (1) flash-based, (2) DRAM-based, and (3) NVM-Swap. For flash-based swap, we use a file in the smartphone's internal NAND flash memory as the swap area.[4] The DRAM-based swapping is essentially a RAM disk and for showing the overhead induced by the storage stack alone. For NVM-Swap, as mentioned above, we use DRAM to simulate NVM swap area by reserving a memory partition from DRAM. For all these swapping scheme, we configure three different swap sizes—64, 128 and 256 MB.

Table 1 lists the workload applications we used to evaluate NVM-Swap in the experiments. Since different kinds of applications may exhibit different memory access patterns, we classify them into six categories, including browser, news, multimedia, social networking, gaming and online shopping. In addition, two mixed categories, namely mix1 and mix2 have been added by combining the applications selected from the above six categories to represent the realistic scenario. For a certain category, we run each application in foreground for 1 minute, and all the applications in that category are run in round robin order for three times. Thus, each application category needs 15 minutes to accomplish the evaluation.

4. Linux allows use a dedicated partition or file as the swap area. Both methods use the same block interface and we use the file approach for simplicity.

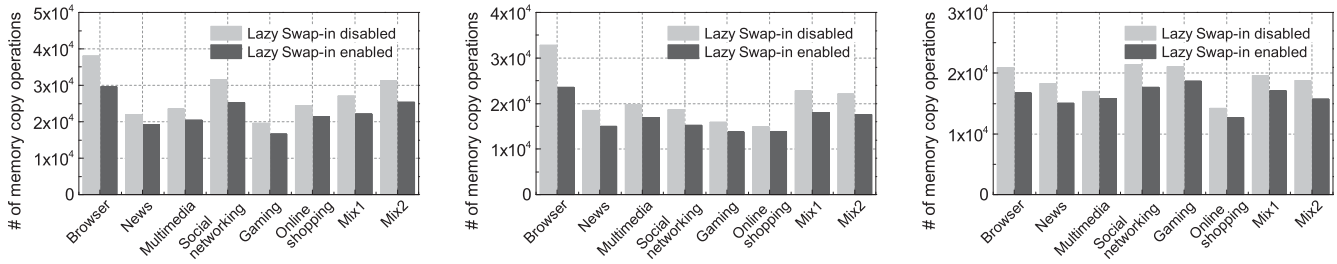
TABLE 1
Workload Applications

Category	Applications
Browser	Google Chrome, Firefox, Opera, Dolphin Browser, UC Browser
News	BBC News, Flipboard, Google Newsstand, Feedly, Yahoo News Digest
Multimedia	KMPlayer, MX Player, Google Play Music, QQ Music, Youtube
Social networking	Facebook, Twitter, Google Plus, Instagram, Pinterest
Gaming	Hill Climb Racing, Temple Run, Boom Beach, Angry Birds, Alto's Adventure
Online shopping	Amazon, TaoBao, eBay, Fancy, Google Play Store
Mix1	Google Chrome, BBC News, Facebook, Alto's Adventure, Amazon
Mix2	Firefox, Yahoo News Digest, Instagram, Angry Brid, TaoBao

4.2 Metrics and Methodology

To evaluate the proposed technique, we collect the results based on the following metrics. The corresponding evaluation methodology for each metric is discussed as well.

1) *Number of memory copy operations:* In NVM-Swap, Lazy Swap-in is designed to reduce the number of memory copy operations (i.e., total number of swap-ins and swap-outs), thus we use this metric to measure the effectiveness of Lazy Swap-in. We run all the applications in each category shown in Table 1 for 15 minutes. All the applications are run in the variant Lazy Swap-in enabled and Lazy Swap-in disabled, respectively. We count the total number of memory copy operations between DRAM and NVM swap area of each category in both configurations. To achieve more accuracy, we run each category in both configurations for five times and calculate the average memory copy operations.
2) *NVM wear-leveling:* We use a synthetic workload to evaluate the effectiveness of Heap-Wear and its performance in NVM-Swap. In detail, we add two system calls: `swap_write()` (writer) and `swap_read()` (reader). The writer invokes the `scan_swap_map()` function to get a swap slot and writes a pre-allocated page to the slot. The flash and DRAM based variants use the default version of this function in the Linux kernel, while the function in NVM-Swap is modified to use Heap-Wear. The reader randomly selects a swap slot that is in use and then frees it. Note that this experiment is synthetic and tries to evaluate the wear leveling behavior of different swapping schemes by stressing the swap area. We also measure the time consumed by both the writer and reader. The writer writes 128 GB data in total and the reader repeats until all data are read out. We find that the amount of data (128 GB) is large enough to make sure all swap slots are used during our experiment.
3) *Application relaunching delay:* Application relaunching delay is an important performance metric for



(a) Swap size = 64MB (b) Swap size = 128MB (c) Swap size = 256MB

Fig. 8. Comparison of number of memory copy operations between Lazy Swap-in disabled and Lazy Swap-in enabled.

smartphone users. Application relaunching may cause inactive pages to be swapped to swap area and requested pages to be loaded from swap area, especially when the system is under memory pressure, in which situation application relaunching may trigger lots of swap-ins and swap-outs. To see the performance improvements of NVM-Swap over flash backed swapping, we run all the applications shown in Table 1 and compare the results between different swapping schemes. To measure the application relaunching delay, we use SwapBench [28] to perform applications auto switching and results collecting.

4) *Application execution time:* Application execution time is another important runtime performance metric. We select one application from each application category as the evaluation benchmark, and the execution time is defined as the time consumed by an application to complete a predefined operation. Table 2 shows the selected applications and their predefined operations. To show the affects of different swapping schemes on application execution time, we conduct the evaluation when the memory is under pressure and page swapping is already triggered.

The experimental results based on the above metrics are discussed in Sections 4.3, 4.4, 4.5, and 4.6. Section 4.7 compares the performance and cost of each swapping schemes in terms of performance-cost ratio.

4.3 Number of Memory Copy Operations

Fig. 8 compares the number of memory copy operations between Lazy Swap-in disabled and Lazy Swap-in enabled with different swap size. As shown, Lazy Swap-in can reduce around 10-30 percent memory copy operations, which means the total number of swap-ins and swap-outs is reduced. Particularly, Browser and Social networking exhibit the highest memory copy reduction. In NVM-Swap, pages that are swapped back from NVM swap area may exhibit infrequent access pattern and consequently be swapped to NVM again, which introduce extra memory copy overhead since each swap-in/swap-out involves at least one NVM/DRAM page read, one DRAM/NVM write, and one PTE update. With Lazy Swap-in enabled, pages can be read directly from the NVM swap area without actually swap in the requested pages, the overhead left is for the PTE update, which only involves writing a 32/64-bit entry.

Besides, we observe that the average number of memory copy operations decreases along with the increase of swap size. This is mainly because that a larger swap area can store more inactive pages swapped from DRAM, leading to the DRAM have more free space to satisfy the memory requests of current running applications. Pages have more chance to reside in DRAM and page thrashing has less chance to happen. Therefore, the total number of memory copy operations (i.e., swap-ins and swap-outs) is reduced when the capacity of the NVM swap area becomes larger.

TABLE 2
Evaluation Applications and Operations of Execution Time

Application	Operation
Google Chrome	Launching and then loading a website
BBC News	Launching and then opening an news
MX Player	Openning a 1080P video
Facebook	Launching and updating posts
Angry Birds	Loading the game
Amazon	Launching and loading product detials

4.4 NVM Wear-Leveling

Fig. 9 demonstrates the write distribution of a traditional DRAM-based swap area by running real applications on Google Nexus 5 for a long period. The *X*-axis lists all the swap slots (i.e., DRAM pages), and the *Y*-axis plots the number that the slot on the *X*-axis is written/used. As shown, in DRAM-backed swapping, slots are not used evenly and most writes are concentrated in a certain number of slots, thus writes are not evenly distributed in the whole swap area. The difference between the maximum and minimum numbers of writes is as large as ~2000. This is mainly because that the slot selection algorithm is originally designed for disk devices and does not aware the zombie slots, which store pages that never or infrequently updated, making the swap area exhibits unbalance write distribution.

Fig. 11 shows the same metric of NVM-Swap using HeapWear with different swap area sizes and thresholds. Compared to DRAM-based swapping, NVM-Swap distributes

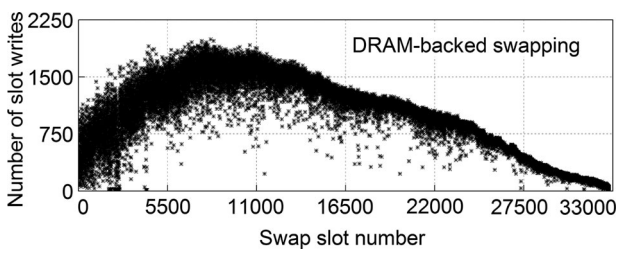


Fig. 9. Write distribution of a 128 MB DRAM-based swap. Due to the lack of NVM awareness, writes are concentrated to certain swap slots, instead of being evenly distributed. Such an unbalanced write pattern greatly reduces the lifetime of NVM with limited endurance.

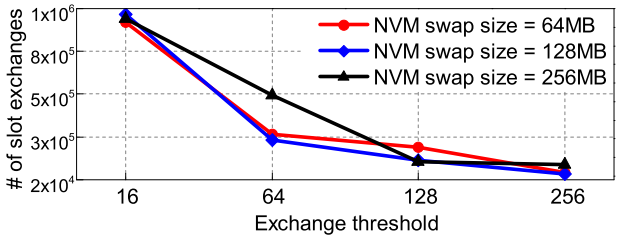


Fig. 10. Number of slot exchanges in Heap-Wear with different swap sizes and thresholds. In general, the larger the threshold, the fewer slot exchanges.

writes much more evenly across the whole swap area. The threshold determines how frequent the slot exchange would happen, which in turn determines the degree of wear leveling (i.e., how evenly the writes could distributed). A smaller threshold will distribute writes in the whole swap area more evenly, but require more slot exchanges operations. As shown in Fig. 11, in general a larger threshold leads to more variations (larger difference between the maximum and minimum numbers of writes). In Fig. 10, we plot the number of slot exchanges with different thresholds and swap area sizes for NVM-Swap. As the threshold increases on the *X*-axis, Heap-Wear performs fewer slot exchange operations. The trend is observed for all NVM swap sizes. However, with the same threshold, the NVM swap size does not affect the number of slot exchanges drastically.

We compare the time consumed by swap in/out operations (the "writer" and "reader" experiments described in the Section 4.2) in Fig. 12 with different swap sizes and thresholds. In the figure, "regular write" represents swapping out a page without slot exchange, while "wear leveling write" means swapping out a page with slot exchange. "read time" represents the time needed to swap in a page from the swap area, and "average write time" denotes the average time needed to swap out a page. Regular writes only need one memory copy operation: copy the victim page from DRAM to the swap area. In contrast, for wear leveling writes,

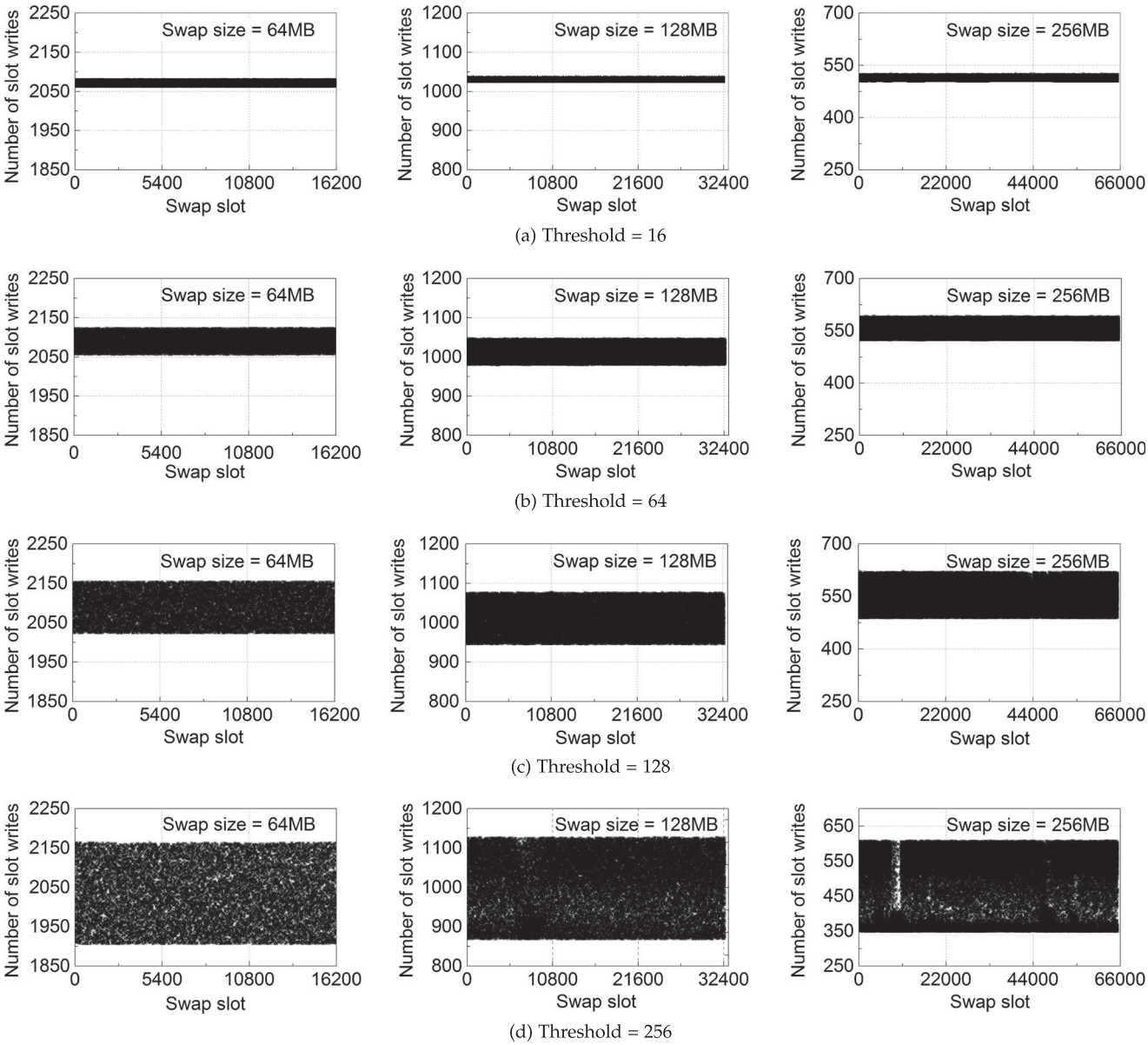


Fig. 11. Write distribution of NVM-Swap with different swap area sizes and thresholds.

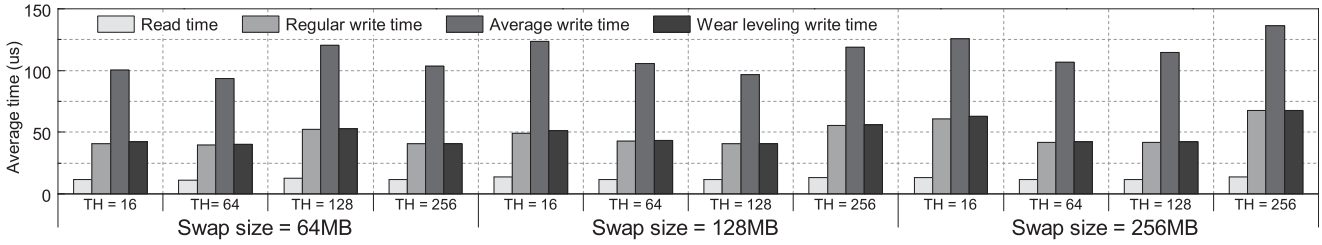


Fig. 12. Access delay of NVM swap area.

one memory copy is needed in slot exchange, and another is needed to copy the victim page from DRAM to the swap area. Besides these two memory copy operations, wear leveling writes also need to update the slot mapping table and the PTEs for page was mapped in paging space.

In the figure, for each swap size and threshold combination, we observe the similar trend that the wear leveling write time is roughly more than twice of a regular write, and the average write time only increases slightly. The reason is that slot exchanges only comprise a small part of the total swap-outs. Table 3 reports the detailed results of time consumed by writing data to NVM swap area. The "Time" column shows the average time consumed by writing a page to NVM swap area during regular writes and wear leveling writes, respectively. As shown, slot exchanges only comprise a small part of the total swap-outs. For instance, for a 128 MB NVM-based swap area with a threshold of 256, slot exchanges only comprise less than 0.2 percent of total swap-outs. The "Number of writes" column shows the total amount of writes during regular writes and wear leveling writes, respectively. The write amounts for regular write are all similar (though slowly increasing) given increasing thresholds. However, a larger threshold significantly helps reduce the amount of writes and access time for wear leveling writes.

4.5 Application Relaunching Delay

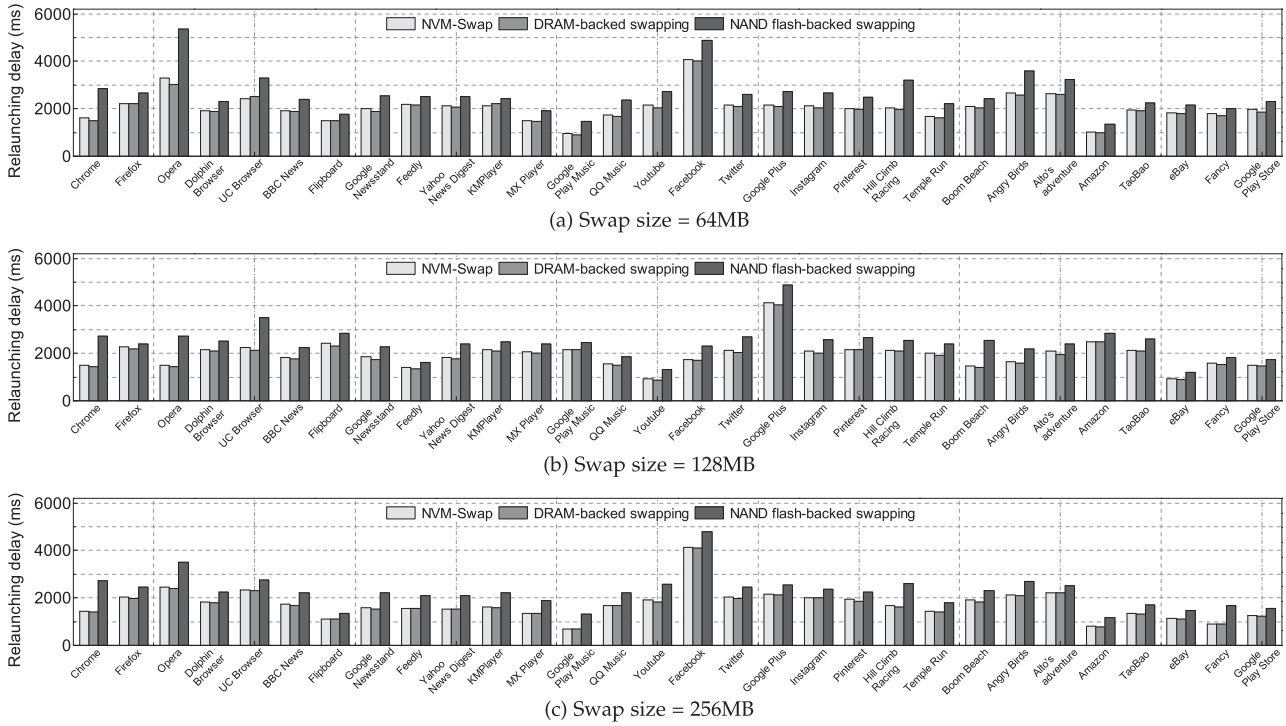
When relaunching an application, if the application is still cached in DRAM, the OS simply bring the application to foreground and start running. Otherwise, the application data has to be reload from secondary storage (i.e., flash memory if the application is killed and swap area if the application data are swapped out), leading to a comparatively high relaunching delay. During the relaunching experiments, applications may be cached in DRAM, partially swapped out or even killed. Therefore, to show how different swapping schemes affect the application relaunching delay, we collect the relaunching delay in each run and report the average average relaunching delay for each application. Fig. 13 compares the application relaunching delay between different swapping schemes with different swap size. As shown in the figure, except for Facebook, which takes around 4 seconds to relaunch (i.e., switch to foreground), the relaunching delays of most applications are 2 seconds. In these three swapping scheme, DRAM-backed swapping exhibits the lowest relaunching delay while NAND flash-backed swapping exhibits the highest relaunching delay.

In NAND flash-backed swapping (i.e., the baseline scheme), writing/reading a page to/from swap area needs to go through the whole storage stack, which is much slower than writing/read a page to/from DRAM or NVM, making NAND flash-backed swapping achieves the highest application relaunching delay. NVM-Swap improves the relaunching speed by replacing slow I/O operations with pure memory copy operations. As shown in Fig. 13, compared to the baseline scheme, the application relaunching speed of NVM-Swap is around 12-46 percent faster. In particular, for Google Chrome and Opera, the relaunching delays of NVM-Swap are respectively 44 and 34 percent faster than that of baseline scheme on average, which means a great improvement on user experience. Therefore, we conclude that NVM-Swap can improve the performance of mobile devices.

As shown in Fig. 13, since DRAM is faster than NVM, DRAM-backed swapping, which indeed is `ramdisk` based, can reduce more relaunching delay than NVM-Swap when compare them to the baseline scheme. However, in DRAM-backed swapping, though it finally calls `memcpy` to swap

TABLE 3
The Results of Writing 128 GB Data to NVM Swap Area

Swap size	Threshold	Regular write		Wear leveling write			Average write time (μs)
		Time (μs)	Number of writes	Time (μs)	Number of writes	% of Total	
64MB	16	40.52	31,847,385	100.24	920,615	2.81	42.19
	64	39.50	32,485,435	93.52	282,565	0.86	39.97
	128	52.26	32,560,165	120.19	207,835	0.63	52.69
	256	40.56	32,705,784	103.51	62,216	0.19	40.68
128MB	16	49.10	31,802,620	123.47	965,380	2.95	51.30
	64	42.71	32,521,793	105.40	246,207	0.75	43.18
	128	40.57	32,638,017	96.45	129,983	0.40	40.78
	256	55.63	32,714,292	118.71	53,708	0.16	55.73
256MB	16	60.99	31,823,880	125.81	944,120	2.88	62.86
	64	41.50	32,264,143	106.92	503,857	1.54	42.51
	128	41.97	32,645,484	144.44	122,516	0.37	42.24
	256	67.36	32,592,368	136.43	107,417	0.33	67.59



(a) Swap size = 64MB

(b) Swap size = 128MB

(c) Swap size = 256MB

Fig. 13. Application relaunching delay under different swapping schemes with different swap size. The baseline is the relaunching delay of applications under NAND flash-backed swapping.

in/out DRAM pages, it still needs to go through a "fake" I/O path to move pages between DRAM and NVM. On the contrary, in NVM-Swap, swapping in and out are pure memory copy operations and the number of memory copy operations is further reduced with the help of Lazy Swap-in. Therefore, due to the overheads brought by the "fake" I/O path, as shown, DRAM-backed swapping only achieves slightly better application relaunching delay than NVM-Swap. For some applications, the relaunching delay of NVM-Swap and DRAM-bakced swapping are almost the same, such as the Fancy in Fig. 13c.

4.6 Application Execution Time

Fig. 14 compares the normalized application execution time under different swapping schemes. Identical to our application relaunching delay results, DRAM-backed swapping achieves the shortest execution time while NAND flash-backed swapping achieves the longest execution time. Compared to NAND flash-backed swapping, application execution in NVM-Swap is around 14-45 percent faster. In the context of runtime execution, faster memory allocation can definitely shorten the execution time. Since the evaluation is conducted when the memory is under pressure, page swapping will be triggered by memory allocation, and thus make page swapping an important factor in application execution time. Therefore, NAND flash-backed swapping always exhibits the longest execution time.

Compared to DRAM-backed swapping, application execution in NVM-Swap is slightly slower. As we analyzed in Section 4.5, although DRAM is faster than NVM in both read and write, the software infrastructure of `ramdisk` slows down the page swapping. Therefore, DRAM-backed swapping only outperforms NVM-Swap by around 4 percent. We believe that with faster NVM, NVN-Swap can achieve higher performance.

4.7 Performance and Cost Analysis

DRAM-backed swapping achieves the best performance with high energy consumption, while NAND flash-backed swapping achieves worst performance with low energy consumption. To show which swapping scheme is more suitable for mobile devices, we compare the *performance per dollar* (performance-cost ratio) [30] of each swapping scheme. To evaluate the application performance under different swapping schemes, we use the sum of average application relaunching speedup and average execution speedup over NAND flash-backed swapping as the performance metric. Therefore, the performance of NVM-Swap and DRAM-backed swapping can be respectively expressed as

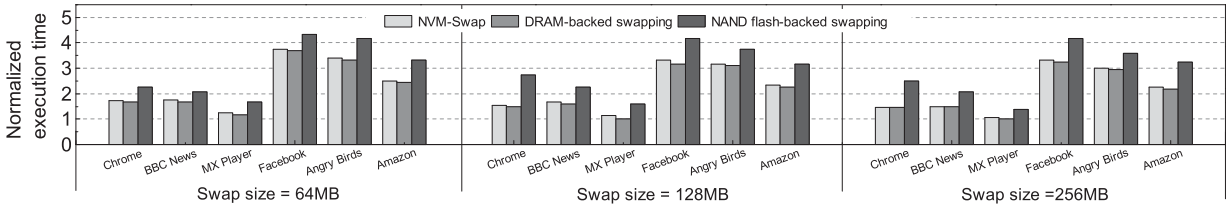


Fig. 14. Comparison of application execution time under different swapping schemes with different swap size.

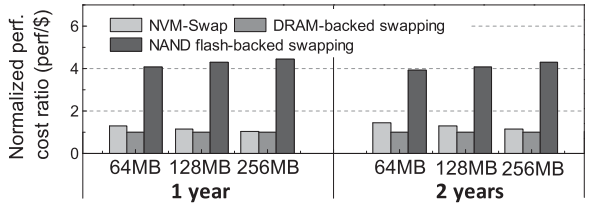


Fig. 15. Comparison of normalized performance-cost ratio (performance/$) between different swapping schemes.

$$P_{NVM} = \frac{\sum_{i=1}^{N} T_{NAND}^{launch}(i) / T_{NVM}^{launch}(i)}{N} + \frac{\sum_{i=1}^{M} T_{NAND}^{exec}(i) / T_{NVM}^{exec}(i)}{M}, \quad (1)$$

$$P_{DRAM} = \frac{\sum_{i=1}^{N} T_{NAND}^{launch}(i) / T_{DRAM}^{launch}(i)}{N} + \frac{\sum_{i=1}^{M} T_{NAND}^{exec}(i) / T_{DRAM}^{exec}(i)}{M}, \quad (2)$$

where $T_{NAND}^{launch}(i)$, $T_{NVM}^{launch}(i)$ and $T_{DRAM}^{launch}(i)$ denote the application relaunching time of ith application in NAND flash-backed swapping, NVM-Swap and DRAM-backed swapping. $T_{NAND}^{exec}(i)$, $T_{NVM}^{exec}(i)$ and $T_{DRAM}^{exec}(i)$ denote the application execution time of ith application.

For fair comparison, we define the cost of different swapping schemes as the sum of energy consumption cost during the lifetime of mobile devices and the swap device cost. Table 4 shows the detailed energy parameters. Note that the prices of DRAM and flash are obtained from Amazon.com, and the swap-ins and swap-outs are estimated based on the number of memory copy operations shown in Section 4.3, the actual numbers may vary with user behavior and swap size. Therefore, the total cost of swapping can be expressed as

$$Cost = (E_{rd} \times N_{rd} + E_{wr} \times N_{wr} + P_{stby} + P_{ref}) \times T \times C_E + C_{price} \times Size_{swap\_area}, \quad (3)$$

where T denotes the endurance of mobile devices.

Fig. 15 shows the normalized performance-cost ratio. Compared to DRAM, although NVM has higher price, it exhibits much lower standby power and zero refresh power, making NVM-Swap outperforms DRAM-backed swapping by around 22 percent. Because of the high idle power (i.e., standby and refresh power) of DRAM, the energy-efficiency of DRAM-backed swapping is much lower than that of both NVM-Swap and NAND flash-backed swapping. Thus, DRAM-backed swapping exhibits the lowest performance-cost ratio. Due to the low price, NAND-flash backed swapping achieves the highest performance-cost ratio. However, the poor read/write performance of NAND flash slows down the application relaunching and execution, makes that the flash-backed swapping is always disabled in mainstream smartphones. Therefore, we argue that NVM-Swap is a better swapping solution for mobile devices.

5 Related Work

There are various of previous work related to NVM-Swap. Most of the related work lies in the following three areas: flash based swapping system, NVM wear-leveling and hybrid NVM/PCM main memory. In this section, we discuss these related work separately.

TABLE 4
Energy and Price Parameters

	DRAM	PCM	Flash
Read (E_{rd})	$4.7 \times 10^{-8} J$/4KB	$1.7 \times 10^{-8} J$/4KB	$2.6 \times 10^{-5} J$/4KB
Write (E_{wr})	$4.7 \times 10^{-8} J$/4KB	$2.1 \times 10^{-7} J$/4KB	$2.9 \times 10^{-5} J$/4KB
Standby (P_{stby})	5.3mW/GB	2.5mW/GB	1.2mW/GB
Refresh (P_{ref})	12.4mW/GB	0	0
Price (C_{price})	~7$/GB	8$/GB [29]	~0.6$/GB
Swap-outs(N_{wr})	~20000 per 15 min, thus ~22.22 pages/sec		
Swap-ins (N_{rd})	~10000 per 15 min, thus ~11.11 pages/sec		
Electricity price (C_E)	~0.12$/KW		

The energy parameters are obtained using the power model in [14].

Flash Based Swapping System. There are several newly proposed flash based swapping systems. MARS [31] is flash-aware page swapping system and aims to speed-up the application relaunching. Jung et al. design and implement FASS [32], which is a raw flash memory based swapping system without using a flash translation layer. FlashVM [18] is another flash backed swapping, which integrates flash memory with virtual memory and provides better garbage collection by batching writes. SSDAlloc [33] is an SSD/DRAM hybrid system which extends DRAM with SSD and allows programmers to treat SSD as DRAM. Recently, Kim et al. [20] evaluate the impact of sub-optimal NAND flash based storage in smartphones and report that storage plays a significant role in application performance. To eliminate the performance gap between main memory and flash based swap area, we replace flash memory with emerging byte-addressable NVM and adopt it as swap area, swap in/out is through memory interface.

NVM Wear-Leveling. Most NVMs have limted enduracne and are vulnerable to unbalance writes. To address this problem, Chen et al. [6] introduce an age-based wear-leveling scheme with near-zero searching cost. Start-Gap [12] and segment swapping [9] are two representative wear-leveling algorithm which aim evenly distribute writes among all PCM cells. Qureshi et al. [34] propose a set of techniques such as lazy write and line level writeback to reduce writes. [35] and [36] aim to prolong the lifetime of PCM-based main memory in embedded systems. Zhang et al. [37] enhance the lifetime of PRAM while considering the process variations. Jiang et al. [38] propose LLS, which is a line-level mapping and salvaging scheme that integrates state-of-art wear leveling techniques. Ferreira et al. [39] increase PCM lifetime by swapping pages on page cache writebacks. Different from these approaches, which target at the wear-leveling of main memory, Heap-Wear uses counters to trace age information of NVM pages and provides page-level wear-leveling with low overhead for NVM swap area.

Hybrid NVM/DRAM Main Memory. Due to their low standby power, high density and byte addressability, NVMs, such as PCM, are considered as promising DRAM alternatives [9], [40]. In [41], [42], a hybrid PCM/DRAM main memory system is proposed where pages can be transferred between PCM and DRAM for saving energy and improving PCM lifetime. Qureshi et al. [34] propose a hybrid main memory organization with on-chip DRAM cache and PCM main memory, in which DRAM is not

visible to OS and managed by the dedicated memory controller. Different from the architecture proposed in [34], in our architecture, DRAM is served as the main memory and NVM is used as the swap area, and both DRAM and NVM are visible to OS. *n*Code [43] is similar to our approach but it aims to reduce the write to NVM by storing code pages in NVM. Though we have the same or similar hardware architecture compared to these hybrid memories, the software or the management strategy is totally different. In the hybrid approaches, NVM is treated as main memory and the space is managed by the OS memory management component for allocation and deallocation. In our architecture, we adopt NVM as the swap area and the space is managed by the swap subsystem, which in charge of allocation swap space to store inactive pages swapped out from DRAM.

6 Conclusion

In this paper, we have revisited swapping in smartphones and proposed NVM-Swap to build high-performance smartphones. We replace part of the DRAM with NVM, and use it as a swap area. Compared to flash-based swap solutions in smartphones, NVM-Swap maintains good user experience (much fewer process terminations) without degrading performance. To reduce the memory copy operations, we proposed Lazy Swap-in, which gives pages a second chance to stay in NVM swap area and supports read pages in NVM directly without copying it back to DRAM. To overcome the drawback that most NVMs have limited write endurance, we designed Heap-Wear, a space-efficient wear-leveling algorithm for NVM-Swap. Experimental results show that Lazy Swap-in can reduce the memory copy operations by around 10-30 percent. Heap-Wear can evenly distribute writes across the whole NVM swap area, greatly improving the lifetime of NVM. Finally, application relaunching delay and application execution time are respectively reduced by around 12-46 percent and 14-45 percent when compared to NAND flash-backed swapping. Moreover, NVM-Swap does *not* target at any specific NVM products as they are still in constant change, our system is kept general enough for quick adoption when NVM hardware becomes available.

Acknowledgments

We would like to thank all the anonymous reviewers for their valuable feedback and improvements to this paper. This work is partially supported by grants from the National Natural Science Foundation of China (61672116, 61601067 and 61472052), National 863 Program 2015AA015304, Chongqing High-Tech Research Program cstc2016jcyjA0332 and cstc2014yykfB40007, the Science and Technology Research Program of Chongqing Municipal Education Commission (KJ1704085), and the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 152138/14E, GRF 15222315/15E and GRF 152736/16E). A preliminary version of this paper was presented at the ACM/IEEE 2014 International Conference on Embedded Software (EMSOFT) [1].

References

[1] K. Zhong, et al., "Building high-performance smartphones via non-volatile memory: The swap approach," in *Proc. 14th Int. Conf. Embedded Softw.*, 2014, pp. 30:1–30:10.

[2] H. S. P. Wong, et al., "Phase change memory," *Proc. IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.

[3] M. Hosomi, et al., "A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM," in *Proc. IEEE Int. Conf. Electron Devices Meeting*, Dec. 2005, pp. 459–462.

[4] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, 2008.

[5] Google, "Running android with low RAM," *Android Developers Documentation*, Google, Mountain View, CA, USA, 2014.

[6] C.-H. Chen, P.-C. Hsiu, T.-W. Kuo, C.-L. Yang, and C.-Y. M. Wang, "Age-based PCM wear leveling with nearly zero search cost," in *Proc. 49th Annu. Des. Autom. Conf.*, 2012, pp. 453–458.

[7] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2009, pp. 347–357.

[8] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. Childers, "Improving write operations in MLC phase change memory," in *Proc. IEEE 18th Int. Symp. High Performance Comput. Archit.*, Feb. 2012, pp. 1–10.

[9] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proc. 36th Int. Symp. Comput. Archit.*, Jun. 2009, vol. 37, pp. 2–13.

[10] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 14–23.

[11] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu, "A low power phase-change random access memory using a data-comparison write scheme," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2007, pp. 3014–3017.

[12] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with Start-gap wear leveling," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2009, pp. 14–23.

[13] K. Zhong, et al., "DR. Swap: Energy-efficient paging for smarthpones," in *Proc. ACM Int. Synp. Low Power Electron. Des.*, Aug. 2014, pp. 81–86.

[14] K. Zhong, et al., "Energy-efficient in-memory paging for smartphones," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1577–1590, Oct. 2016.

[15] H. Saadeldeen, et al., "Memristors for neural branch prediction: A case study in strict latency and write endurance challenges," in *Proc. ACM Int. Conf. Comput. Frontiers*, 2013, pp. 26:1–26:10.

[16] A. Jog, et al., "Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs," in *Proc. 49th Des. Autom. Conf.*, Jun. 2012, pp. 243–252.

[17] W. Mauerer, *Professional Linux Kernel Architecture*. Birmingham, U.K.: Wrox Press Ltd., 2008.

[18] M. Saxena and M. M. Swift, "FlashVM: Revisiting the virtual memory hierarchy," in *Proc. 12th Conf. Hot Topics Operating Syst.*, 2009, p. 13.

[19] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 309–320.

[20] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, p. 17.

[21] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc. ACM Int. Conf. Archit. Support Program. Languages Operating Syst.*, Mar. 2011, vol. 47, pp. 91–104.

[22] S. Eilert, M. Leinwander, and G. Crisenza, "Phase change memory: A new memory enables new memory usage models," in *Proc. IEEE Int. Conf. Memory Workshop*, May 2009, pp. 1–2.

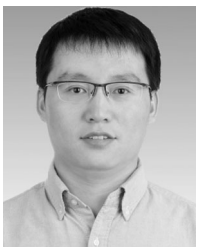
[23] J. Condit, et al., "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 133–146.

[24] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. 9th USENIX Conf. File Stroage Technol.*, 2011, pp. 61–76.

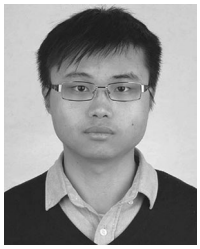
[25] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 672–685.

[26] J. Coburn, et al., "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proc. 16th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 105–118.

[27] LPDDR2-PCM Phase Change Memory 45nm Discrete, Micron Inc., Boise, ID, USA 2011,
[28] X. Zhu, D. Liu, L. Liang, K. Zhong, M. Qiu, and E. H. M. Sha, "SwapBench: The easy way to demystify swapping in mobile systems," in *Proc. IEEE 17th Int. Conf. High Performance Comput. Commun.*, 2015, pp. 497–502.
[29] Y. Zhou, R. Alagappan, A. Memaripour, A. Badam, and D. Wentzlaff, "HNVM: Hybrid NVM Enabled Datacenter Design and Optimization," Microsoft, Microsoft Research, Tech. Rep. MSR-TR-2017-8, Feb. 2017.
[30] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 33–45.
[31] W. Guo, K. Chen, H. Feng, Y. Wu, R. Zhang, and W. Zheng, "MARS : Mobile application relaunching speed-up through flash-aware page swapping," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 916–928, Mar. 2016.
[32] D. Jung, J. Soo Kim, S. Yeong Park, J. Uk Kang, and J. Lee, "FASS: A flash-aware swap system," in *Proc. Int. Workshop Softw. Support Portable Storage*, 2005.
[33] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM memory management made easy," in *Proc. Symp. Netw. Syst. Des. Implementation*, 2011, pp. 211–224.
[34] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 24–33.
[35] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. H. M. Sha, "Application-specific wear leveling for extending lifetime of phase change memory in embedded systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 33, no. 10, pp. 1450–1462, Oct. 2014.
[36] J. Hu, Q. Zhuge, C. J. Xue, W. C. Tseng, and E. H. M. Sha, "Software enabled wear-leveling for hybrid PCM main memory on embedded systems," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2013, pp. 599–602.
[37] W. Zhang and T. Li, "Characterizing and mitigating the impact of process variations on phase change based memory systems (micro)," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2009, pp. 2–13.
[38] L. Jiang, Y. Du, Y. Zhang, B. R. Childers, and J. Yang, "LLS: Cooperative integration of wear-leveling and salvaging for PCM main memory," in *Proc. IEEE/IFIP 41st Int. Conf. Depend. Syst. Netw.*, 2011, pp. 221–232.
[39] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Moss, "Increasing PCM main memory lifetime," in *Proc. Des. Autom. Test Eur Conf. Exhib.*, 2010, pp. 914–919.
[40] Z. Shao, Y. Liu, Y. Chen, and T. Li, "Utilizing PCM for energy optimization in embedded systems," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, Aug. 2012, pp. 398–403.
[41] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *Proc. 46th ACM/IEEE Des. Autom. Conf.*, Jul. 2009, pp. 664–669.
[42] W. Zhang and T. Li, "Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures," in *Proc. 18th Int. Conf. Parallel Archit. Compilation Techn.*, 2009, pp. 101–112.
[43] K. Zhong, et al., "nCode: Limiting harmful writes to emerging mobile NVRAM through code swapping," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2015, pp. 1305–1310.



Duo Liu received the BE degree in computer science from the Southwest University of Science and Technology, Sichuan, China, in 2003, the ME degree from the Department of Computer Science, University of Science and Technology of China, Hefei, China, in 2006, and the PhD degree in computer science from the Department of Computing, The Hong Kong Polytechnic University, in 2012. He is currently an assistant professor in the College of Computer Science, Chongqing University, China. His current research interests include emerging memory techniques and embedded systems. He is a member of the IEEE.



Kan Zhong received the BSc degree in computer science from Chongqing University, Chongqing, China, in 2013, where he is currently working toward the PhD degree. His current research interests include mobile computing, embedded system, and non-volatile memory.

Xiao Zhu received the BSc degree in computer science from Chongqing University, Chongqing, China, in 2014, where he is currently working toward the master's degree. His current research interests include mobile computing, embedded system, and emerging memory techniques.

Yang Li received the BSc degree in computer science from Chongqing University, Chongqing, China, in 2015, where he is currently working toward the master's degree. His current research interests include approximate computing, embedded system, and emerging memory techniques.

Linbo Long received the BSc and PhD degrees in computer science from the College of Computer Science, Chongqing University, China, in 2011 and 2016. He is currently a lecturer of College of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing, China. His current research interests include compiler optimization, emerging memory techniques, and embedded systems.

Zili Shao received the BE degree in electronic mechanics from the University of Electronic Science and Technology of China, Sichuan, China, in 1995, and the MS and PhD degrees from the Department of Computer Science, University of Texas at Dallas, in 2003 and 2005, respectively. He has been an associate professor in the Department of Computing, Hong Kong Polytechnic University, since 2010. His research interests include embedded software and systems, real-time systems, and related industrial applications.