Building High-Performance Smartphones via Non-Volatile Memory: The Swap Approach

Kan Zhong Tianzheng Wang[§] Xiao Zhu Linbo Long
Duo Liu[*] Weichen Liu Zili Shao[†] Edwin H.-M. Sha

College of Computer Science, Chongqing University, liuduo@cqu.edu.cn
Key Lab. of Dependable Service Computing in Cyber Physical Society (Chongqing Univ.), Ministry of Education
[§]Department of Computer Science, University of Toronto, tzwang@cs.toronto.edu
[†]Department of Computing, The Hong Kong Polytechnic University, cszlshao@comp.polyu.edu.hk

Website: http://nvm-swap.bitbucket.org

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 revisit 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. Based on NVM's high performance and byte-addressability, we show that a copy-on-write swap-in scheme can achieve even better performance by avoiding unnecessary memory copy operations. To avoid fast worn-out of certain NVMs, we also propose Heap-Wear, a wear leveling algorithm that more evenly distributes writes in NVM. Evaluation results based on the Google Nexus 5 smartphone show that our solution can effectively enhance smartphone performance and give better wear-leveling of NVM.

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. Nevertheless, the NAND flash based storage—an important but usually ignored component—has been evolving very slowly and failing to catch up with mobile processors and large main memories [9, 13]. This discrepancy leads to a unique phenomenon that is only happening in smartphones: processes constantly get terminated ("killed") when memory is under pressure because swapping to flash is usually disabled to avoid sacrificing performance [6]. To show the effectiveness of swapping, in Figure 1 we plot the percentage of reduction on process termination when running a mixture of applications for 30 minutes in a Google Nexus 5 smartphone[1] with swap enabled. With different memory capacities on the X-axis, swapping can help reduce around 66% to 91% of process terminations, greatly lowering the chance of an application being terminated when the system is short for memory. However, the slow performance of NAND flash prohibits adopting swapping directly, and the dominant solution today is still to simply terminate processes when the system is under pressure.

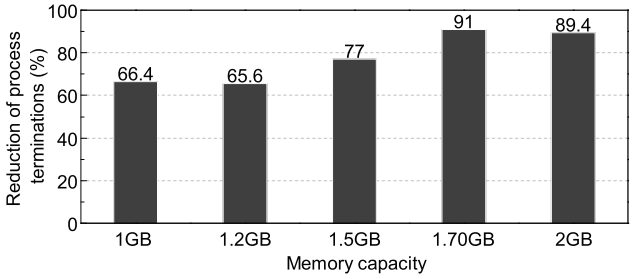


Figure 1: Percentage of reduced process terminations with swapping when running various applications in a Google Nexus 5 smartphone (Android 4.4) for 30 minutes. Swapping can reduce around 66% to 91% of process terminations.

Emerging byte-addressable, non-volatile memory (NVM) technologies such as phase change memory (PCM) [26], spin-transfer torque RAM (STT-RAM) [7] and memristor [23] are

[*]Duo Liu is the corresponding author.

EMSOFT'14, October 12 - 17, 2014, New Delhi, India.

http://dx.doi.org/10.1145/2656045.2656049.

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

Table 1: Comparing PCM, DRAM and NAND flash [4, 22].

Attributes	DRAM	PCM	Flash
Non-volatility	No	Yes	Yes
Write bandwidth	∼GB/s	50–100MB/s	5–40MB/s
Write latency	20–50ns	∼1us	∼500us
Erase cycles	∞	$10^8 - 10^9$	$10^4 - 10^5$

changing this situation. Compared to NAND flash, these NVM products offer not only faster (near-DRAM) performance, but also larger erase cycles. As shown in Table 1, PCM exhibits much better write latency and endurance when compared to flash memory. A plethora amount of work have been proposed to further achieve near-DRAM performance and better endurance for PCM [1, 2, 5, 10, 15, 18, 19, 30]. Other NVM technologies such as STT-RAM could promise even faster performance than DRAM [11]. 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 [6]. To further reduce unnecessary overheads, we propose copy-on-write swap-in (COWS), for read requests to directly get data from the swap area with zero memory copy. When a swapped-out page is accessed again, COWS sets up the page table mapping and returns the page in NVM directly, without first copying the page out of the swap area. When a write happens to that page, we do the actual swap-in by copying the page from NVM to DRAM (thus the name "copy-on-write swap-in"). COWS reduces the consumption of DRAM, allowing even more applications to be retained in DRAM, and NVM's high performance can ensure that the user experience is not degraded because of swapping.

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 [26]). 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 we have mentioned earlier, a lot of wear leveling techniques have been proposed to mitigate the endurance problem of NVM [1, 2, 10, 15, 30]. However, different from those existing work, which usually depends on data comparison write (DCW) [2, 28] or segment switching [19], 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. Though our experimental platform is 32-bit and the 4 GB address space is very limited, note that the size of memory is increasing continuously. However, larger size of memory will lead to more energy consumption. To alleviate this issue, an NVM-based swapping area is still needed to reduce energy consumption. In this paper, we only focus on performance and endurance issues. We have discussed energy related issues in previous work [29]. Experimental results show that NVM-Swap can avoid terminating most processes and reduce application launch time by more than 20% when compared to the no-swap case and a flash-based swap. In addition, with Heap-Wear, swap slots are more uniformly written, giving us a durable NVM-Swap. We keep our design general enough to ensure that NVM-Swap can be easily adopted by platforms other than Android. Our system is available at:

`nvm-swap.bitbucket.org`

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 copy-on-write swap-in (COWS) to avoid unnecessary memory copy induced by read-only requests, 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 background on swapping and NVM in smartphones. 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

In this section, we first give background on NVM. We then discuss swapping in smartphones.

2.1 Emerging Byte-Addressable NVM

Non-volatile memory (NVM) 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 (PCM) [26], 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 (20ns – 50ns) and also limited write endurance (around 10^8 – 10^9 programming cycles) [15, 30]. To overcome these problems, various wear leveling and caching schemes have been proposed [1, 2, 10, 15, 30]. 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. Thus we also do not regard performance as a major problem in this paper. Moreover, the swap area is usually cleared after use and prepared as a clean slate before being used again. Therefore, endurance is not a major concern, especially in the context of swapping.

Other promising NVM technologies include spin-transfer torque RAM (STT-RAM) [7] and memristor [23]. 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 [20]. Because of its low latency (10ns – 25ns [7]) and non-volatility, STT-RAM can be used to build both on-chip cache and main memory [11, 27]. These newer NVMs also have better endurance behavior than 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 pressure and finds itself difficult to satisfy memory allocation requests, the OS will choose to write ("swap") some 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 [21]. However, the situation in mobile devices is different, despite they can also use NAND flash as the swap area.

Different from enterprise-level flash-based solid state drives, mobile NAND flash storage performs notoriously worse [9, 13]. 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. The latest Google Android 4.4 allows "swap to ZRAM" (a.k.a "Compcache") [6], 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 1000mAh – 2000mAh 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, we argue that the ultimate solution to extending memory is to enable a "real" swap area that is backed by emerging fast NVM, instead of a RAM disk that absorbs even more energy and computation power.

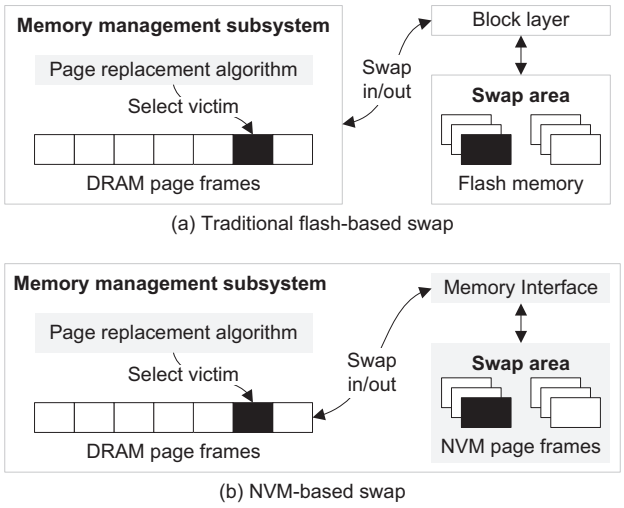


Figure 2: Comparison between traditional flash-based swapping and NVM-Swap. (a) A flash-based swap area requires swapping requests go through the whole storage stack. (b) NVM-Swap uses memory interface to access the swap area, which is backed by NVM (attached to the memory interface). Note that in NVM-Swap, the swap area becomes part of the memory management subsystem and requires no I/O operations.

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) COWS which avoids unnecessary memo-

ry copy 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 COWS and Heap-Wear in detail.

3.1 Architecture

In our previous work [29], 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. 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 Figure 2. In Figure 2(a) 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 Figure 2(b), 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 Figure 2(b) 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 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 [24] do not exist in NVM-Swap, since swap area is discarded after use, and will be re-initialized when it is setup.

3.2 Copy-on-Write Swap-In

Though avoiding the storage stack in swapping and the high-performance nature of NVM eliminate most overhead, we find that swap-ins in NVM are actually inducing unnecessary extra memory copy: victim pages will be copied first to the swap area and then copied back to main memory (i.e., DRAM) when these pages are requested again by applications. The kernel handles such requests through the page fault handler, which reads the swap area to fetch the requested page frame, set up new page table mappings and return to the user application. The whole operation will involve at least one NVM page read, one DRAM page write and one page table entry (PTE) write. However, a large number of such accesses are read access and will not modify the page. The whole swap-in process is actually inducing extra memory copy operations. NVM's byte-addressablity provides a great opportunity to reduce such unnecessary memory copy operations for read requests, since the requested memory page already resides in memory—the NVM—though in a different region (the swap area).

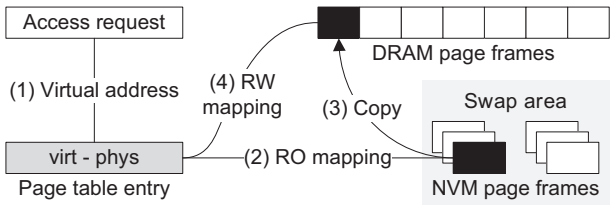
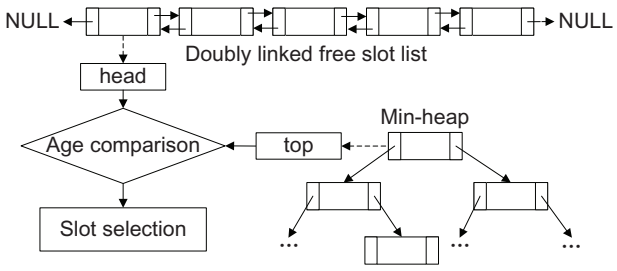
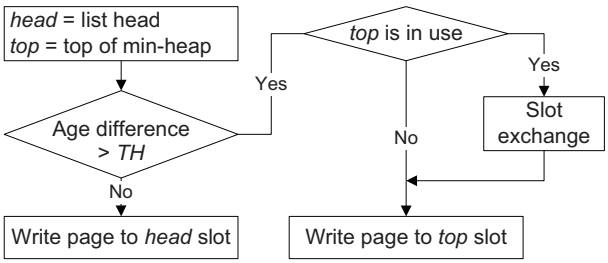


Figure 3: Copy-on-write swap-in (COWS). (1) The CPU uses virtual address to access a page that was swapped out; (2) The page fault handler sets up read-only (RO) permission for the requested page in NVM; (3–4) Any attempt to write the page will trigger a page fault, which will copy the page from NVM to DRAM and grant it read-write (RW) permission. Application can read the data directly in NVM and no data are transferred between DRAM and NVM for read requests, avoiding extra memory copy operations.

We adopt the concept of copy-on-write for swap-ins to remove unnecessary memory copy operations between NVM and DRAM. As shown in Figure 3, copy-on-write swap-in (COWS) directly sets up page table mappings for the application when handling page fault incurred by accessing a page that was swapped out. The memory pages are then become available directly to the user space. In this way, we avoid first reading and then copying the page from NVM to DRAM. No memory copy operation is required between DRAM and NVM. COWS naturally utilizes the fast (at least near-DRAM) read performance of most NVM technologies. We do not allow direct write to the NVM page if it was swapped in by COWS by marking the page as "read-only" in its page table entry. If the application needs to modify the page, we do the actual swap in—copy the requested page from NVM to DRAM and set up new page table mappings accordingly. Again we rely on the page fault handler to copy the page from NVM to DRAM and setup the page table entry (recall that COWS first maps the page as read-only, and a write attempt to the page will then trigger a page fault). Compared to the traditional approach, we speed up the swap-in process for read-only requests by avoiding transferring a whole page between DRAM and NVM. The only overhead left is one write for the page table entry, which only involves writing a 32-bit entry in most today's ARM-based smartphones. It is true that most NVMs are slower than DRAM, however, in general it is expected that their read performance is very close to DRAM. The NVM pages will also be cached by the processor caches, just like caching DRAM pages, which has been shown to be effective [3].



(a) The free slot list and min-heap data structure.



(b) Illustration of slot selection in Heap-Wear.

Figure 4: Design of Heap-Wear.

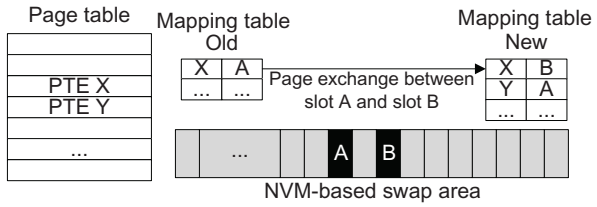


Figure 5: Example of Heap-Wear.

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, by default pages that are being swapped out could hit arbitrary swap slots in an unbalanced manner 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. Figure 4(a) 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. As shown in Figure 4(b), once a free slot in the head is selected for 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 3.1.

Algorithm 3.1 Heap-Wear algorithm

Input: $list$: free slot list; $heap$: slot min-heap; TH: slot exchange threshold.
Output: $offset$: the slot used to hold an inactive page.

```
1: Let head = list.head.
2: Let top = heap.top.
3: if head.age − top.age > TH then
4:    if top is used then
5:       /* Perform a slot exchange. */
         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:      end if
13:   else
14:      Remove top form list.
15:   end if
16:   /* Put the victim page to the top slot. */
      offset = top.
17: else
18:   /* Put the victim page to the head slot. */
      Remove head form list.
19:   offset = head.
20: end if
21: return offset.
```

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. Figure 5 shows an example of exchanging pages between two slots. Initially, slot A is on top of the min-heap and holds a swapped-out page, which corresponds to PTE X, that means slot A is not referenced via page table. Then to allocate a slot for the victim page, slot B is selected from free slot list. Comparing the ages of slot A and slot B finds the slot exchange condition is satisfied. Therefore, pages in slots A and B are exchanged, with corresponding PTEs re-mapped. Otherwise, if slot A is referenced via page table, we not only perform slot exchanging but also update all the PTEs that are pointing to slot A. We make all new PTEs reference slot B through reverse mapping—the corresponding PTEs of all processes that use a particular page are mapped to that 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 128MB swap area only needs no more than 1MB main memory space to store the data structure. 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

Table 2: Workload applications.

Category	Application
Browser	Android Browser, Firfox Browser for Android, Google Chrome, Opera
Social networking	Facebook, Google+, Pinterest, QQ, Sina Weibo, Skype, Twitter, WhatsApp
Multimedia	Google Play Music, MX Player, TTpod Player, Youtube
Office	Evernote, Gmail, Google Drive, Google Maps, Office Mobile
Gaming	Angry Brid, Asphalt 8 , Temple Run 2
Online shopping	Amazon, Ebay, Fancy, Google Play, TaoBao
News	BBC News, Engadget, Flipboard, Google Newsstand, NBC News, NetEase News, Netflix, TED, Zaker

is activated, the age counters are loaded to main memory, and only synchronized periodically to avoid wearing out the underlying NVM cells. With only one extra NVM page copy when exchanging two NVM swap slots, Heap-Wear can avoid zombie slots efficiently.

4. EVALUATION

We have implemented NVM-Swap in Google Android 4.4 for the Google Nexus 5 smartphone. In the Linux kennel, we introduced a new memory zone for allocating NVM page frames. The swap subsystem is modified to use memory from this NVM zone. We also modified the page fault handler to realize COWS and used functions in Heap-Wear to manage swap slots. In our experiment, we do not target at any specific type of NVM or emulate its latency values. Though different NVM products have different performance parameters, we believe that future mature NVM products will generally provide near-DRAM performance and reasonable lifetimes. Moreover, our system does not rely on any specific type of NVM and can be easily deployed in different NVM-based systems. In the rest of this section, we first describe experimental setup, metrics and methodology. Finally, we discuss experimental results.

4.1 Experimental Setup

We run all experiments with a Google Nexus 5 smartphone. It features Qualcomm Snapdragon 800 processor clocked at 2.26GHz and 2GB DRAM as main memory. Our Nexus 5 model has 16GB internal flash storage. The Linux kernel we use for Android is 3.4. 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.

We test three different swap variants: (1) flash-based, (2) DRAM-based, and (3) NVM-Swap. For flash-based swap, we use a 128MB file in the smartphone's internal NAND flash memory as the swap area[2]. The DRAM-based variant

[2]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.

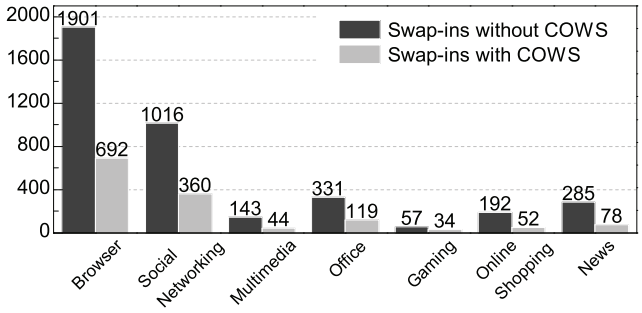


Figure 6: Memory copy reduction results. For each application category, COWS is able to reduce a lot of memory copy operations, especially for browser and social networking applications.

is essentially a 128MB RAM disk and for showing the overhead induced by the storage stack alone. For NVM-Swap, we replace part of DRAM with NVM. We use DRAM to simulate NVM swap area by reserving 256MB memory from main memory as NVM products are not yet widely available. When the hardware becomes available, our system can be quickly adopted. In addition to the 128MB capacity which is the same as the other variants, we also test NVM-Swap with 64MB and 256MB capacities to evaluate our system in detail. For each experimental variant, we run seven types of popular applications, as listed in Table 2. The applications range from browser, social networking, to gaming, etc.

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.

Number of memory copy operations. We use this metric to measure the effectiveness of COWS. We run all the applications in each category shown in Table 2 for 15 minutes. All the applications are run in the variant with COWS and without COWS, respectively. We count the total number swap-ins of each category and compare the results between the two configurations. For more accuracy, we run each category in both configurations for five times and calculate the average value.

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 128GB data in total and the reader repeats until all data are read out. We find that the amount of data (128GB) is large enough to make sure all swap slots are used during our experiment.

Application launch time. Application launch time is an very important performance metric. However, it is not straightforward to test application launch time and obtain accurate results. We therefore use customized applications

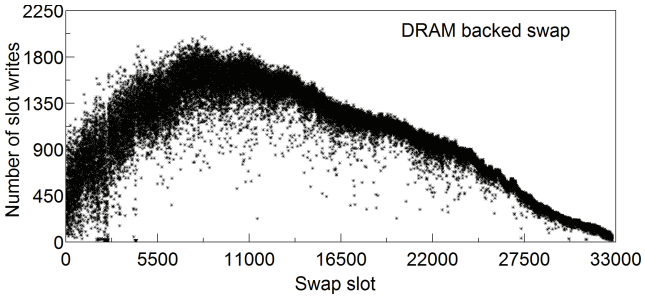


Figure 7: Write distribution of a 128MB DRAM-based swap. Due to the lack of NVM awareness, writes are concentrated to certain swap slots, instead of being evenly distributed. Such an unbalanced write pattern greatly reduces the lifetime of NVM with limited endurance.

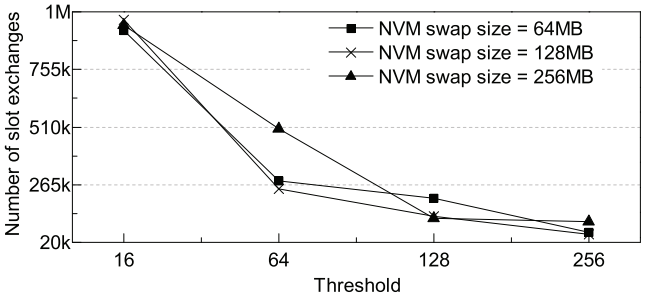


Figure 8: Number of slot exchanges in Heap-Wear with different swap sizes and thresholds. In general, the larger the threshold, the fewer slot exchanges.

to collect the results of this metric. We designed five simple test applications, each of which loads a file of certain size from NAND flash to main memory. This simulates the application launch process, which essentially just loads executables, configuration files and shared libraries into main memory. Before running these test applications, we first run another simple application that allocates almost all the available memory, making sure that the launch of new applications will trigger swapping. We run each test application with the DRAM-based variant and NVM-Swap, and record the time spent on loading the files. As a baseline, we also run these experiments with swap disabled.

The experimental results based on the above metrics are discussed in Sections 4.3 - 4.5.

4.3 Number of memory copy operations

Figure 6 shows the result for memory copy reduction. As shown, COWS can help reduce around 40% – 75% of swap-ins, which means a great number of memory copy operations were reduced. According to the figure, we observe a lot of reductions of memory copy operations for all categories. In particular, browser and social networking applications exhibit the highest number of memory copy reduction.

With COWS, we reduce the number of memory copy operations by around 60% for browser, social networking, office and online shopping applications. Moreover, for multimedia and news applications, COWS can reduce more than 70% of memory copy operations. The only overhead of COWS is updating the page table entries (PTEs) when handling page faults. Compared to actually swapping in NVM pages, which needs one page copy and one PTE update, the overhead of COWS is negligible.

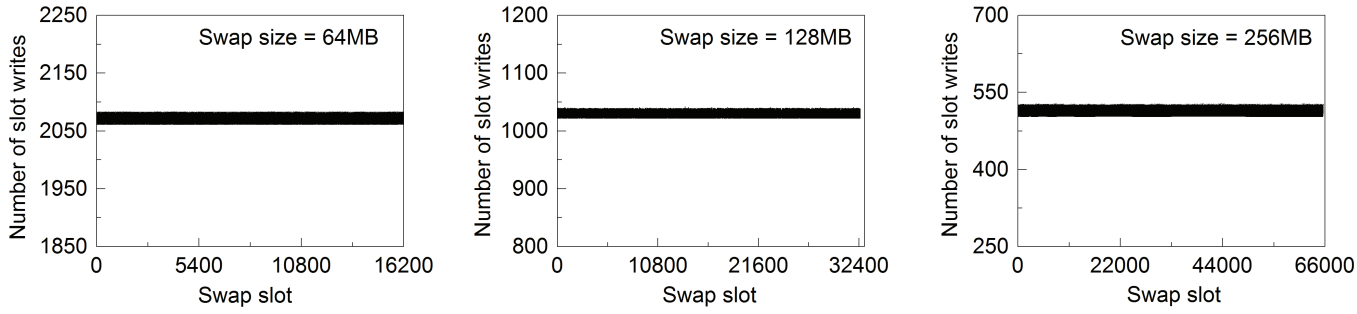
4.4 Wear Leveling

Figure 7 illustrates the write distribution of a traditional DRAM-based swap. The X-axis lists all the swap slots, and the Y-axis plots the number that the slot on the X-axis is written/used. As shown, in a DRAM-based swap, 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.

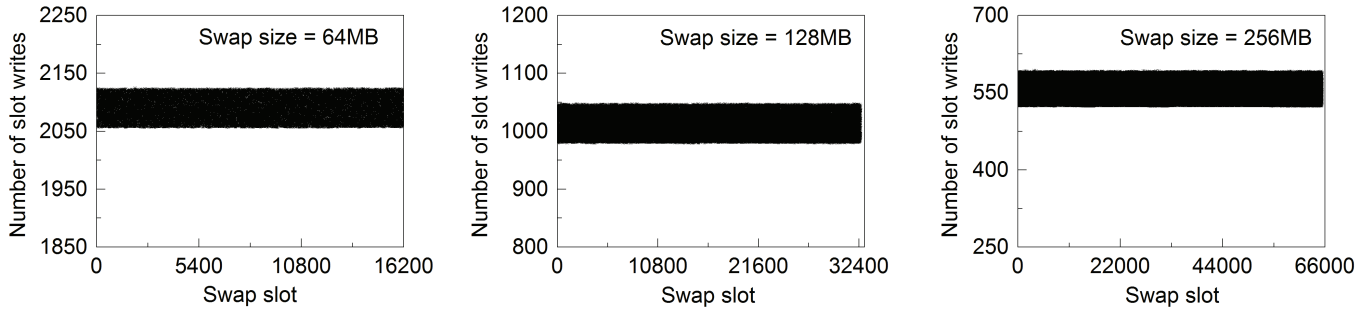
Figure 9 shows the same metric of NVM-Swap using Heap-Wear with different swap area sizes and thresholds. Compared to DRAM-based swap, NVM-Swap distributes writes much more evenly across the whole swap area. The threshold determines if a slot exchange should happen, which in turn determines the degree 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 Figure 9, in general a larger threshold leads to more variations (larger difference between the maximum and minimum numbers of writes). In Figure 8, 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 Figure 10 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, 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.

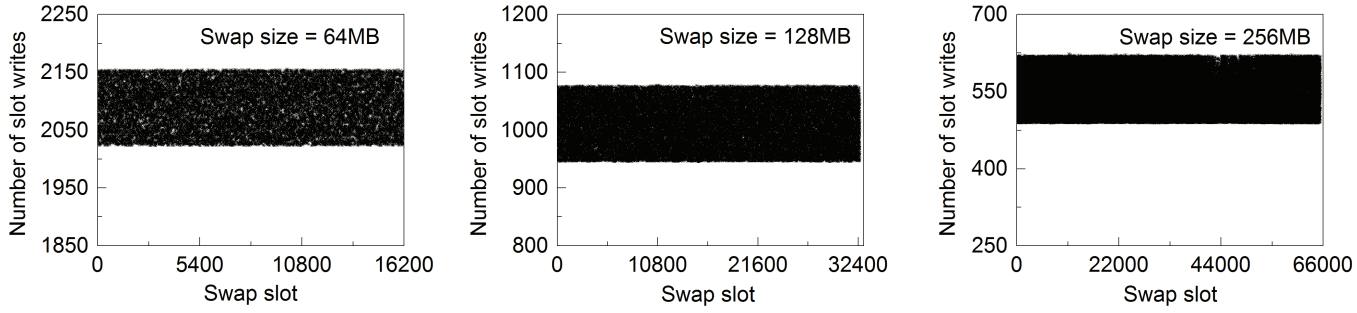
In the figure, for each swap size and threshold combination, we observe the similar trend that the wear leveling write time is roughly more than twice of a regular write, and the average write time only increases slightly. The reason is that slot exchanges only comprise a small part of the total swap-outs. 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 128MB NVM-based swap area with a threshold of 256, slot exchanges only comprise less than 0.2% of total swap-outs. The "Count" column shows the total amount of writes during regular writes and wear leveling writes, respectively. The count numbers for regular write are all similar (though slowly increasing) given increasing thresholds. However, a larger



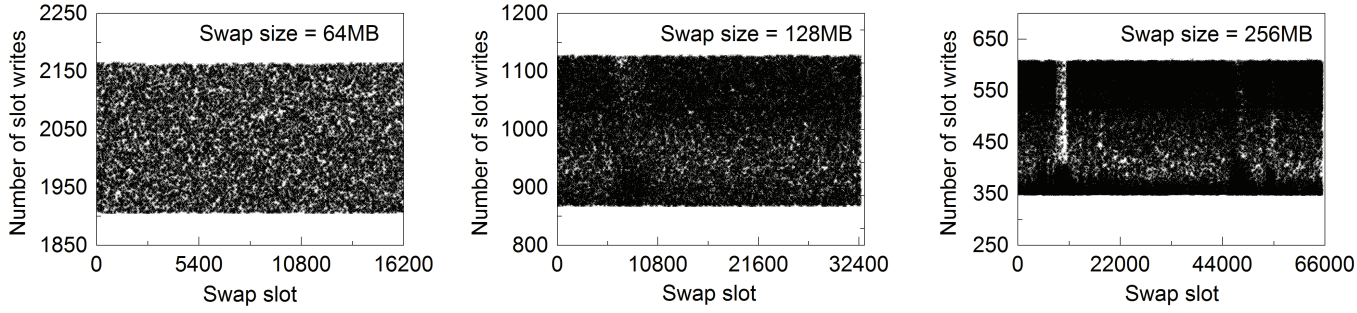
(a) Threshold = 16.



(b) Threshold = 64.



(c) Threshold = 128.



(d) Threshold = 256.

Figure 9: Write distribution of NVM-Swap with different swap area sizes and thresholds.

threshold significantly helps reduce the amount of writes and access time for wear leveling writes.

4.5 Application Launch Time

Application launch time is an important performance metric, especially for smartphone users. Figure 11 shows the time taken to launch a certain application (i.e., loading the file from NAND flash) under different configurations. Each application loads a file of certain size upon start. The size of the file is set as 10MB, 15MB, 20MB, 25MB and 30MB for the five applications (i.e., App1-5). In the figure we plot three different schemes, including NVM-Swap, flash-based swap, and a baseline when swap is disabled. Note that before starting this experiment, we first make sure that the memory is almost out of space. As a result, NVM-Swap and flash-based swap, the page replacement algorithm will try to find memory space for the application by swapping out inactive pages. For swap disabled, the low memory killer will start

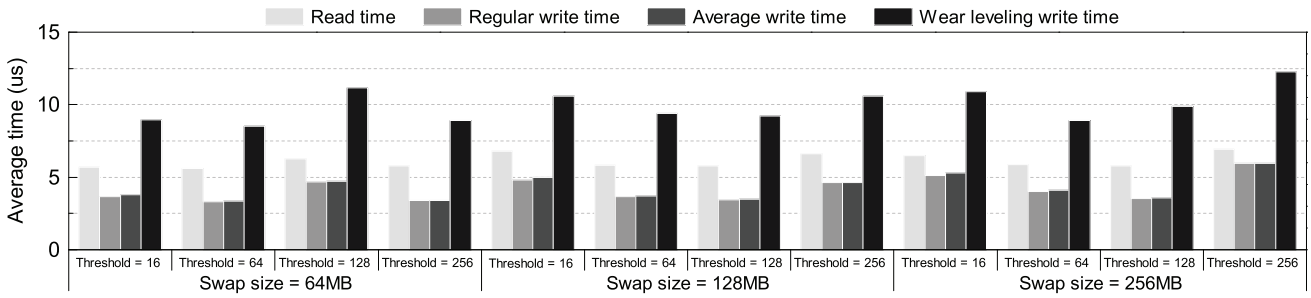


Figure 10: Access delay of NVM swap area.

Table 3: The results of writing 128GB data to NVM swap area.

Swap size	Threshold	Regular write		Wear leveling write			Average write time (μs)
		Time (μs)	Count	Time (μs)	Count	% of Total	
64MB	16	3.65	31,847,385	8.95	920,615	2.81	3.80
	64	3.30	32,485,435	8.51	282,565	0.86	3.34
	128	4.70	32,560,165	11.17	207,835	0.63	4.74
	256	3.38	32,705,784	8.90	62,216	0.19	3.39
128MB	16	4.80	31,802,620	10.58	965,380	2.95	4.97
	64	3.66	32,521,793	9.41	246,207	0.75	3.71
	128	3.45	32,638,017	9.23	129,983	0.40	3.47
	256	4.64	32,714,292	10.58	53,708	0.16	4.65
256MB	16	5.13	31,823,880	10.91	944,120	2.88	5.29
	64	4.01	32,264,143	8.91	503,857	1.54	4.08
	128	3.53	32,645,484	9.90	122,516	0.37	3.55
	256	5.94	32,592,368	12.28	107,417	0.33	5.96

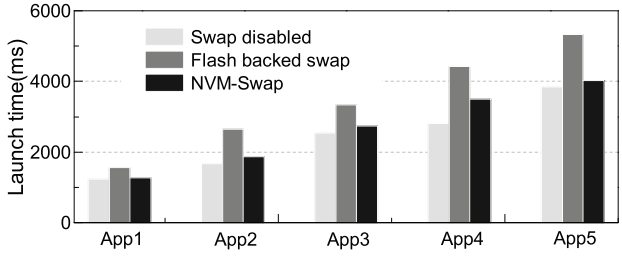


Figure 11: Application launch time under different swap implementations. App1-5 simulate application launch by loading a file of size 10MB to 30MB (5MB increment). The experiments are run when the system is under heavy memory pressure, guaranteeing to trigger swapping.

to terminate victim processes to make room for the memory requests made by the experimental application. As shown in Figure 11, using the low memory killer is the fastest way to satisfy the memory request by killing victim processes when compared to both swap schemes. However, the advantage is not significant when compared with NVM-Swap, and yet with NVM-Swap processes will not be forcedly terminated, thus maintaining good user experience. Compared to flash-based swap, NVM-Swap is more than 20% faster on average. This is mainly due to the fast performance of NVM and the removal of the storage stack in swapping related operations. Therefore, we conclude that NVM-Swap can help maintain good user experience (fewer process terminations) without degrading performance, as we have predicted in Section 1.

5. RELATED WORK

Most related work lies in areas of flash-based mobile storage and swapping. In general, NAND flash plays a key role in smartphones. Kim et al. [13] evaluated the impact of sub-optimal NAND flash based storage in smartphones and proposed several pilot solutions, including using NVM to removing the storage bottleneck. File system configuration also affect the performance of smartphones [14]. For example, utilizing special features of the eMMC interface [8] could result in faster mobile storage. To address the endurance problem of flash, Liu et al. proposed PCM-FTL [16] to co-optimize the lifetime of NAND flash and PCM in embedded systems. FTL^2 [25] is another write reduction caching design to tackle the endurance problem and performance degradation in flash memory. FlashVM [21] integrates flash memory with virtual memory to improve performance. FASS [12] is another flash-ware swap system, but targets at raw flash memory without using the flash translation layer. Although these flash-based proposals utilize characteristics of flash memory and perform well in general-purpose systems, they could not be adopted directly by smartphones due to the sub-optimal flash-based storage performance. We therefore seek better medium as swap for smartphones.

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 unnecessary memory copy operations, we propose copy-on-write swap-in (COWS) to guarantee zero-copy for read-only swap-in requests. To overcome the drawback that most NVMs have limited write endurance, we design Heap-Wear, a space-efficient wear-

leveling algorithm for NVM-Swap. Experimental results show that NVM-Swap with COWS can reduce more than 50% of memory copy operations on average. Heap-Wear can evenly distribute writes across the whole NVM swap area, greatly improving the lifetime of NVM. Finally, application launch time is reduced by more than 20% on average when compared to flash-based swap. Moreover, NVM-Swap does *not* target at any specific NVM products as they are still in constant change, but our system is kept general enough for quick adoption when NVM hardware becomes available. Accurately modeling NVM latency, bandwidth and endurance properties is promising future work.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our shepherd Prof. Christoph Kirsch for their valuable feedback and improvements to this paper. This work is partially supported by the National Natural Science Foundation of China (61309004), National 863 Program (2013AA013202), Research Fund for the Doctoral Program of Higher Education of China (20130191120030), Chongqing cstc2012ggC40005 and cstc2013jcyjA40025, and Fundamental Research Funds for the Central Universities (CDJZR14185501).

REFERENCES

[1] 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 *Proceedings of DAC*, pages 453–458, 2012.

[2] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *Proceedings of MICRO*, pages 347–357, 2009.

[3] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of SOSP*, pages 133–146, 2009.

[4] S. Eilert, M. Leinwander, and G. Crisenza. Phase change memory: A new memory enables new memory usage models. In *Proceedings of IMW*, pages 1–2, 2009.

[5] C. Fu, M. Zhao, C. J. Xue, and A. Orailoglu. Sleep-aware variable partitioning for energy-efficient hybrid PRAM and DRAM main memory. In *Proceedings of ISLPED*, pages 75–80, 2014.

[6] Google. Running android with low RAM. *Android Developers Documentation*, 2014.

[7] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-RAM. In *Proceedings of IEDM*, pages 459–462, 2005.

[8] Hynix. eMMC flash storage using 41nm technology. *Hynix Newsletter*, 2009.

[9] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O stack optimization for smartphones. In *Proceedings of USENIX ATC*, pages 309–320, 2013.

[10] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. Childers. Improving write operations in MLC phase change memory. In *Proceedings of HPCA*, pages 1–10, 2012.

[11] A. Jog, A. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. Das. Cache revive: Architecting volatile STT-RAM caches for enhanced performance in cmps. In *Proceedings of DAC*, pages 243–252, 2012.

[12] D. Jung, J. soo Kim, S. yeong Park, J. uk Kang, and J. Lee. FASS: A flash-aware swap system. In *Proceedings of IWSSPS*, 2005.

[13] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage*, 8(4):1–25, 2012.

[14] H. Kim and D. Shin. Optimizing storage performance of android smartphone. In *Proceedings of ICUIMC*, pages 1–7, 2013.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of ISCA*, pages 2–13, 2009.

[16] D. Liu, T. Wang, Y. Wang, Z. Qin, and Z. Shao. PCM-FTL: A write-activity-aware NAND flash memory management scheme for PCM-based embedded systems. In *Proceedings of RTSS*, pages 357–366, 2011.

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

[18] K. Qiu, Q. Li, and C. J. Xue. Write mode aware loop tiling for high performance low power volatile PCM. In *Proceedings of DAC*, pages 1–6, 2014.

[19] 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 *Proceedings of MICRO*, pages 14–23, 2009.

[20] H. Saadeldeen, D. Franklin, G. Long, C. Hill, A. Browne, D. Strukov, T. Sherwood, and F. T. Chong. Memristors for neural branch prediction: A case study in strict latency and write endurance challenges. In *Proceedings of CF*, pages 1–10, 2013.

[21] M. Saxena and M. M. Swift. FlashVM: Revisiting the virtual memory hierarchy. In *Proceedings of HotOS*, pages 13–13, 2009.

[22] Z. Shao, Y. Liu, Y. Chen, and T. Li. Utilizing PCM for energy optimization in embedded systems. In *Proceedings of ISVLSI*, pages 398–403, 2012.

[23] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[24] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of ASPLOS*, pages 91–104, 2011.

[25] T. Wang, D. Liu, Y. Wang, and Z. Shao. FTL^2: a hybrid flash translation layer with logging for write reduction in flash memory. In *Proceedings of ACM LCTES*, pages 91–100, 2013.

[26] H. S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[27] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li. Emerging non-volatile memories: Opportunities and challenges. In *Proceedings of CODES+ISSS*, pages 325–334, 2011.

[28] 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 *Proceedings of ISCAS*, pages 3014–3017, 2007.

[29] K. Zhong, X. Zhu, T. Wang, D. Zhang, X. Luo, D. Liu, W. Liu, and E. H.-M. Sha. DR. Swap: Energy-efficient paging for smarthpones. In *Proceedings of ISLPED*, pages 81–86, 2014.

[30] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of ISCA*, pages 14–23, 2009.