Building NVRAM-Aware Swapping Through Code Migration in Mobile Devices

Kan Zhong, Duo Liu, Lingbo Long, Jinting Ren, Yang Li, and Edwin Hsing-Mean Sha

Abstract—Mobile applications are becoming increasingly feature-rich and powerful, but also dependent on large main memories, which consume a large portion of system energy, especially for devices equipped with 4/6 GB DRAM. Swapping inactive DRAM pages to byte-addressable, non-volatile memory (NVRAM) is a promising solution to this problem. However, most NVRAMs have limited write endurance and the current victim pages selecting algorithm does not aware it. Therefore, to make it practical, the design of an NVRAM based swapping system must also consider endurance. In this paper, we target at prolonging the lifetime of NVRAM based swap area in mobile devices by reducing the write activities to NVRAM based swap area. Different from traditional wisdom, such as wear leveling and hot/cold data identification, we propose to build a system called *n*Code, which exploits the fact that code pages are easy to identify, read-only, and therefore a perfect candidate for swapping. Utilizing NVRAM's byte-addressability, we support execute-in-place (XIP) of the code pages in the swap area, without copying them back to DRAM based main memory. Experimental results based on the Google Nexus 5 smartphone show that *n*Code can effectively prolong the lifetime of NVRAM under various workloads.

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

1 Introduction

Mobile devices like smartphones are always short for main memory as applications are getting more and more powerful and resource demanding. Having larger main memory, which is DRAM based can effectively meet the applications' demands for memory space. A shown in Fig. 1, the memory capacity of flagship smartphones has increased 24 times over the past seven years. To meet the memory demands, current mainstream phones have equipped 4 GB or even 6 GB main memory, such as Samsung Galaxy Note 7 and OnePlus 3, which respectively equipped with 4 GB and 6 GB DRAM. However, more DRAM installed means more energy consumption and drains the battery more faster. Swapping is an effective way to extend memory capacity without adding more DRAM for big servers. However, it has been disabled in most smartphones and other mobile systems due to the sub-optimal random access performance of flash memory [2]. To still satisfy memory allocation requests under this setting when the system is under memory pressure, existing running applications have to be constantly terminated. This way avoids the performance degradation caused by swapping to flash, however, it makes application switch and load time longer, and worsens user experience.

Emerging byte-addressable non-volatile memory (NVRAM) technologies such as phase change memory (PCM) [3], spin-transfer torque RAM (STT-RAM) [4], memristor [5] and Intel and Micron's 3D XPoint [6] have nearly DRAM performance and much lower standby and read power than DRAM. Several approaches have been proposed to integrate NVRAM into computer systems [7], [8], [9], [10]. With byte-addressable NVRAM, swapping is no more a deal-breaker for smartphone performance and user experience. Swapping to NVRAM [11] avoids excessive application terminations and can preserve the most performance as NVRAM is orders of magnitude faster than flash. Since NVRAM is byte-addressable, the heavy storage layer which traditional swapping is built on can be replaced by a lightweight memory layer, further improving performance and simplifying system design. Re-adopting swapping using NVRAM also significantly reduces energy consumption of smartphones as NVRAM eliminates a considerable portion of the system's standby power [12], [13], [14].

Despite the benefits brought by NVRAM based swapping, a must-solve problem that is similar to swapping to flash is the endurance of NVRAM—most NVRAMs have limited write endurance ("write-limited"). For example, the way PCM works determines a cell can only be re-written for 10^8–10^9 times before worn-out. To have a practical NVRAM based swapping system in smartphones, the endurance of NVRAM must be prolonged. Although a heap-based wear-leveling algorithm is already proposed to evenly distribute writes to all NVRAM pages in NVRAM based swap area [11], the total writes to NVRAM based swap area is not reduced. Therefore, to prolong the lifetime of NVRAM based swap area, we argue that when swapping to NVRAM, one must reduce writes to NVRAM as many as possible.

- *K. Zhong, D. Liu, J. Ren, Y. Li, and E. H.-M. Sha 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: {kzhong1991, liuduo}@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.

Manuscript received 31 Oct. 2016; revised 30 Apr. 2017; accepted 24 May 2017. Date of publication 8 June 2017; date of current version 11 Oct. 2017.
(Corresponding author: Duo Liu.)
Recommended for acceptance by M. Kandemir.
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/TPDS.2017.2713780

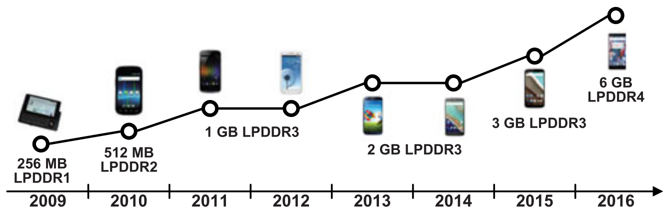


Fig. 1. Memory evolution of flagship smartphones from 2009 to 2016.

When selecting victim DRAM pages to swap out, the conventional wisdom is active/inactive pages identification. Identifying inactive pages—those that are not frequently modified—is both lightweight and proactive as in this way, we can control which pages can be placed in NVRAM and DRAM main memory. Since inactive pages are not accessed frequently, they are given a higher priority to be swapped out to NVRAM. However, different applications have vastly different data access patterns, obtaining proper active/inactive pages identification for smartphones without application-specific information and make it adaptive to different workloads are difficult. Moreover, due to intensive user interactions—applications are consecutively switched between foreground and background, page thrashing, which exists in most inactive/active data identification based page replacement algorithms will be exacerbated. Therefore, pages may consecutively moving between DRAM and NVRAM swap area, makes the current victim pages selecting algorithm does not aware the limited endurance of NVRAM.

Instead of selecting inactive pages, we find that read-only pages are good candidates for swapping to write-limited NVRAM as read-only pages can be read directly in the NVRAM. However, identifying read-only pages involves inspecting every pages in the process address space, which would introduce a great overhead. We observe that *code pages* are all read-only and identifying code pages is very straightforward with the segment information provided by the OS. With these insights, we propose to build a system called "*nCode*" to reduce writes on NVRAM based swap space in smartphones. *n*Code gives code pages higher priorities when finding swapping candidates. In particular, *n*Code utilizes the unique *direct read* ability provided by NVRAM based swapping [11], [12] to allow execute-in-place of code pages on the swap space. Since NVRAM is byte-addressable, all the swapped out pages can be accessed directly through `load` and `store` instructions. Consequently, one does not have to swap in pages for read requests. Therefore, an NVRAM based swapping system can simply set up page table mappings from virtual addresses to physical NVRAM addresses to allow direct read. Code pages are read and executed directly in the swap area, without being swapped back to main memory.

We have implemented *n*Code in Google Android and its Linux kernel. To evaluate its effectiveness, we conduct experiments using various real applications on a Nexus 5 smartphone. Experimental results show that *n*Code can reduce significant number of writes to NVRAM when compared to swapping schemes that use traditional page frame reclaim algorithm, furthering improving the endurance of NVRAM. The contributions of this work are summarized as follows:

TABLE 1
Comparison of PCM and DRAM [7], [24], [25]

Attributes	DRAM	PCM
Read Latency	∼50ns	50 - 100ns
Set/Reset Latency	∼20 - 50ns	∼100 - 700ns
Power	∼W/GB	100→500mW/die
Idle power	∼W/GB	≪0.1W
Endurance (write cycles)	10^{15}	$10^8 - 10^9$

- We propose *n*Code, a discriminative NVRAM based swapping system that prioritizes the swapping of read-only code pages, to reduce harmful writes to write-limited NVRAM.
- Utilizing the byte-addressability of NVRAM, we provide XIP support of code pages in the NVRAM swap area, without copying the code pages back to DRAM.
- We implement *n*Code into Android's Linux kernel, and evaluate *n*Code with real applications based on Google Nexus 5.

The remainder of this paper is organized as follows. Section 2 outlines background on swapping and emerging NVRAM in mobile devices, as well as the motivation. Section 3 details the design of *n*Code. Evaluation results are presented and analyzed in Section 4. Finally, we discuss related work and conclude in Sections 5 and 6, respectively.

2 Background and Motivation

In this section, we first introduce the background on NVRAM and swapping in mobile devices, especially in smartphones, then we present the motivation.

2.1 Emerging Non-Volatile Memory

Byte-addressable non-volatile memory such as PCM [3], STT-RAM [4], memristor [5] and Intel and Micron's 3D XPoint [6] are future candidates for replacing DRAM (as main memory), SRAM (as on-chip cache), and even flash (as storage) because of its nice features such as low power consumption, high density and non-volatility [15], [16]. Compared to DRAM, NVRAM does not need constant voltage to maintain data because it keeps data by changing the physical state of its underlying material, such as resistance level. PCM is one of the most promising candidates. It uses the state changing between amorphous and crystalline of phase-change materials (e.g., GST) to record logical zeros and ones. Therefore, as shown in Table 1, PCM exhibits much lower power consumption and similar read latency but longer write latency when compared to DRAM.

Despite the advantages of NVRAM, most NVRAMs are write-limited, i.e., each PCM cell can only be programmed for a limited 10^8–10^9 times [17], [18], while the number for DRAM is at least 10^{15}. Besides PCM, other NVRAMs also have similar limited write endurance. Two major ways to overcome the endurance problem of NVRAM are wear leveling and write reduction. Wear leveling evenly distributes the writes over all memory cells to prevent some cells wearing out faster [19], [20], [21]. Write reduction tries reduce the number of writes or the number of bit flips to memory cells [22], [23]. In this paper, we try to prolong the lifetime of NVRAM by reducing the total writes to NVRAM based swap area. Moreover, this paper do not target at any specific NVRAM technologies as the proposed technique works in the OS level.

2.2 Swapping and Victim Pages Selecting

Main memory has never been enough for smartphone applications, hence the trend of having big main memories in tiny smartphones. However, more memory also means more energy consumption since the memory subsystem is a major contributor to the overall energy consumption. Traditionally, an effective solution to this problem is swapping. Instead of adding more DRAM, swapping extends the memory capacity by borrowing space from secondary storages. When there is no sufficient space to meet the memory allocation request, a page frame reclaim routine will start and write inactive pages to swap area. In most smartphones, flash memory is adopted as the storage device due to its low cost and high capacity, and therefore is usually used as the swap area when swapping is enabled. However, flash memory based swapping is usually disabled in smartphones for two main reasons. First, due to the limited bandwidth and performance of the on-board flash memory (i.e., eMMC[1]), swapping in and out pages can lead to I/O contentions, and thus potentially slowing down the normal I/O operations. Second, frequent page reads/writes will exacerbate the wear condition of flash memory and increase its garbage collection overhead.

A recent solution to enable swapping without performance degradation is using the in-memory paging architecture [11], [12], [13]. In this architecture, NVRAM is attached to the memory bus, side by side with DRAM, thus it can be accessed directly by using the `load` and `store` instructions. Different from the hybrid memories, which treat NVRAM as part of main memory, the in-memory paging architecture dedicates NVRAM as the swap area to store inactive DRAM pages when there is no sufficient DRAM space. When selecting victim pages, a key component is the page frame reclamation algorithm (PFRA), which is used to select victim pages which needs to be swapped out. Usually two page lists are maintained: the *active list* and *inactive list*. Anonymous pages in the inactive list are scanned and selected as candidates to be swapped out. However, smartphones usually exhibit intensive user interactions, making applications consecutively switching between foreground and background, and thus pages may alternate between active and inactive states and consecutively moving between DRAM and NVRAM swap area. Therefore, to make NVRAM-based swapping practical, a must-solve problem is the endurance of NVRAM. Although a head-based ware-leveling algorithm has been proposed to evenly distribute writes to all NVRAM pages [11], the total writes to NVRAM is not reduced, which can shorten the lifetime of NVRAM swap area.

In this paper, we found that page type is usually available to the OS, and it is easy for the OS to differentiate code and data pages. The read-only code pages making them a perfect candidate for swapping to write-limited NVRAM.

1. Embedded Multi Media Card (eMMC), comprised by a flash chip and a controller.

TABLE 2
The Number of Code Pages of Popular Applications

Applications	# of total pages	# of code pages	Ratio
Firefox	44,199	7,338	16.60%
Chrome	28,408	5,896	20.75%
Ebay	21,621	4,577	21.17%
Google+	22,162	5,030	22.70%
Facebook	29,244	5,522	18.88%
Pinterest	23,941	4,198	17.53%
Instagram	13,318	2,557	19.20%
Flipboard	22,358	4,008	17.93%

Therefore, we argue that a better solution is to swap out code pages and execute the code in-place of NVRAM swap area, without copying them back to main memory. Through this, the consecutively page moving between DRAM and NVRAM can be avoid and thus the writes to NVRAM swap area can be reduced.

2.3 Motivation

In NVRAM-based swapping, traditional page reclaim algorithm does not aware the limited endurance of NVRAM and may consecutively move pages between DRAM and NVRAM, thus shorten the lifetime of NVRAM based swap area.

In this paper, we give the highest priority to code pages for swapping out. In Table 2, we compare the number of total pages and the number of code pages of several popular applications. As shown, code pages, including processes' text section and shared library make a large proportion (i.e., around 20 percent) of the application's total pages. Therefore, selecting code pages for swapping out shows a high potential in reclaiming memory space. Moreover, code pages usually exhibit read-only access permission. This motivate us to design a novel page reclaim technique, which selects code pages to swap out and executes code pages in place on the NVRAM-based swap area. Unlike the traditional page selecting policy, in which pages are copied back to DRAM upon requested and thus pages may consecutively moving between DRAM and swap area, the proposed technique executes the code pages in place on the NVRAM and thus avoid page thrashing, which finally reduce the writes to NVRAM swap area. We illustrate the design and evaluation of this architecture in the rest of this paper.

3 *n*CODE DESIGN

In this section, we first give an overview of *n*Code. We then present the key techniques of *n*Code in details, including runtime code pages identification, supporting for code pages XIP of swap area and NVRAM swap area management.

3.1 Overview

We illustrates the system architecture of *n*Code in Fig. 2. In an NVRAM based swapping system, the swap area is directly attached to the memory bus and accessed through `load` and `store` instructions, instead of block I/O. The proposed *n*Code contains three major components: victim page selection, XIP engine and NVRAM swap area management. The victim page selection module responses for

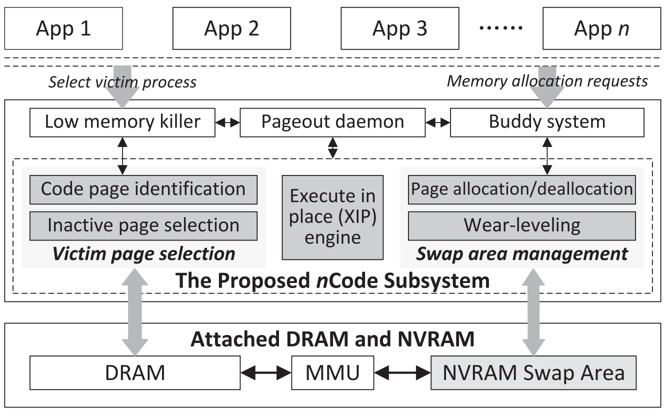


Fig. 2. System architecture. Swap area is backed by NVRAM and memory interface is used to swap out victim code pages selected by *n*Code.

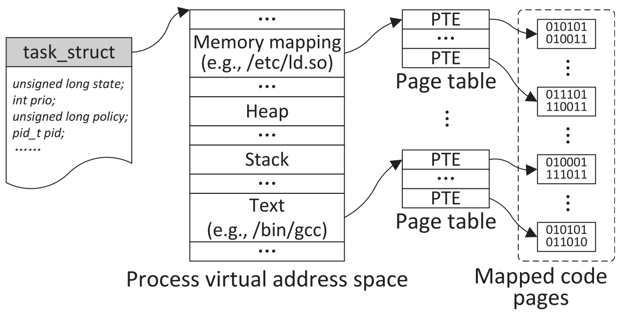


Fig. 3. Identifying code pages of an victim process. Code pages are scanned trough the process virtual address space using the information provide in kernel data structures.

selecting victim pages to swap out and then the XIP engine provides support for XIP of code pages in swap area. NVRAM swap area management module manages the allocation/deallocation of NVRAM pages and integrates a heap-based wear-leveling algorithm, which has been proposed and evaluated in our previous work [11]. When the system's memory is under pressure, the pageout daemon (i.e., `kswapd`) starts to reclaim memory space by swapping selected victim pages to NVRAM swap area and writing back cached pages to secondary storage.

Since code pages contain program binary code or mapped shared libraries, they can be easily recognized using the information provide by the OS. Different from normal data pages that can be read and written, code pages are usually read-only and marked as `executable`. Exploiting this property, when selecting victim pages to reclaim memory space, *n*Code will start to scan the virtual address space of all running processes to identify code pages and and select those pages as candidates to be swapped out. Then with the support of XIP engine, we execute the code in-place of the swap area without copying these code pages back to main memory by utilizing the byte-addressability of NVRAM. However, when there is no enough code pages to select for swapping out, the traditional page selecting algorithm is used to reclaim inactive pages.

Since the NVRAM pages are allocated/deallocated in unit of single page, we currently employ a simple doubly linked list to manage all the free pages in NVRAM swap area. Our previously proposed heap-based wear-leveling algorithm, namely Heap-Wear is also adopted to evenly distribute writes across the whole swap area. Heap-Wear records the age information of each page in NVRAM and always chooses a "young" NVRAM page to store the page swapped out from DRAM, guarantees "old" page will not be worn out quickly. In this paper, we will focus on the code pages identification and in-place execution, more details about NVRAM swap space management can be found in our previous work [11].

3.2 Page Reclaiming in *n*Code

When the system's memory is under pressure, i.e., the amount of free memory is below a predefined threshold, *n*Code will start to select victim pages to reclaim memory space. In our design, the code pages have the highest priority to be swapped out and when there is no more code page to reclaim, inactive pages will be selected. In this section, we will primarily show how to identify code pages by scanning the process virtual address space at runtime.

3.2.1 Identifying Code Pages

The identification of code pages (i.e., selection of code pages as swapping candidates) is accomplished with the pageout daemon, which is responses for monitoring memory usage and reclaiming memory space by invoking memory shrink components, such as *n*Code. Therefore, when there is no sufficient memory left, the pageout daemon will notify *n*Code to start memory reclaiming process. In *n*Code, instead of selecting inactive pages, we select code pages as swapping candidates, utilizing the information provided by various data structures that maintain process states at runtime. After code pages are migrated to the NVRAM based swap area, they are mapped to the processes' virtual address space via page tables to allow direct read and XIP [12]. Swapping in upon read is therefore avoid. The code pages in NVRAM are freed when the process referencing them exit.

To obtain candidate code pages, we first select victim processes which owns code pages. As shown in Fig. 2, *n*Code uses the mechanism provide by Android Low Memory Killer (LMK) to select victim process. In Android OS, LMK was originally for terminating applications to make room for incoming memory allocation requests. When selecting victim process, LMK tends to choose a process with the lowest priority, which means the process currently is not a foreground process and not used by smartphone users, it is running or cached in the background. Therefore, those kind of processes are perfect candidates. Note that if multiple processes have the same priority, the one that with the highest memory usage will be selected. After a process is selected, we scan its virtual address space to find out all code pages (i.e., those that are marked `executable`).

Fig. 3 illustrates how to identify code pages through process virtual address space. As shown, the virtual memory space belonging to a process consists of multiple virtual memory areas (a.k.a `vma`), which are organized using a linked list. Each memory area has multiple physical page frames mapped with it through page tables, and its corresponding data structure has several state flags to denote its purpose, such as `read` and `executable`, indicating that the physical pages are used to store shared libraries (e.g., `/etc/ld.so`) or the text section (i.e., the process' code). In *n*Code, we scans the given process' memory areas and select

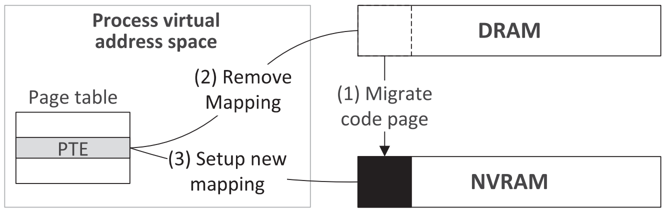


Fig. 4. XIP of code pages. (1) The code page is swapped out from DRAM to NVRAM; (2) Remove the mapping for the code pages and free them; (3) Set up new page table mappings for the code pages in NVRAM, to allow the code to be executed in-place on NVRAM.

the executable areas. For each page in the selected areas, we mark it as code page and migrate it to NVRAM swap area to save DRAM space.

3.2.2 The Secondary Choice: Inactive Pages

In *n*Code, when there is no more code page to swap out, which means that all the processes' code pages have been migrated to NVRAM swap area, the traditional page frame reclaim algorithm will be adopted to select inactive pages for swapping out. Therefore, inactive pages are considered as secondary swap candidates in *n*Code. A traditional PFRA maintains all the userspace pages in an active and inactive list, which are both kept in least recently used order, and pages can be moving between active list and inactive based on their access patterns. In *n*Code, when the system is under memory pressure and all the code pages have been migrated to NVRAM swap area, pages in the inactive list are selected to be swapped out.

Algorithm 1 shows the detailed steps to select victim pages in *n*Code. As shown, we first use LMK to choose an victim process (line 3). Then, our algorithm scans its virtual address space and selects the executable virtual memory area. For each page in the selected memory area, we allocate a new page in NVRAM (line 8), redirect the virtual addresses that were pointing to the DRAM page to the NVRAM page using the XIP engine (line 9), and copy the contents of DRAM page to NVRAM page using memory copy (line 10). The DRAM page is then freed to the OS memory manager for future memory allocations (line 11). However, when all processes' code pages have been migrated to NVRAM swap area (i.e., $victim\_list = null$), *n*Code will use traditional PFRA to swap out inactive pages (line 13). Moreover, if the NVRAM swap area if full, LMK has to choose a process from all the userspace process and then terminates it to reclaim memory space (line 15-16). As shown in the algorithm, the code pages identification involves scanning a process' all virtual memory areas. Let m denotes the number of virtual memory areas and n denotes the number of pages in a virtual memory area, the time complexity of the algorithm is $\mathcal{O}(mn)$.

3.3 XIP of Code Pages

Utilizing the byte-addressability of NVRAM, *n*Code provides XIP of code pages. Code pages in NVRAM are not swapped in upon read access. In a naive implementation of NVRAM based swap space, victim pages will be copied to swap area first and then copied back to main memory when these pages are accessed again. The kernel handles such requests through the page fault handler, which fetches the requested pages from swap area, sets up new page table mappings and then returns to the user process. The whole operation will involve at least one page read in the swap area, one memory page write and one page table entry (PTE) write. The whole swap-in process is actually inducing extra memory copy operation. With the byte-addressability of NVRAM, unnecessary memory copy can be avoided as the requested page already resides in memory—the NVRAM swap area.

Algorithm 1. Page Reclaiming in *n*Code

```
Input: $victim\_list$: reclaimable process list, $proc\_list$: list of all
       userspace processes;
Output: $null$
 1: if NVRAM swap area has enough free space then
 2:   if $victim\_list \neq null$ then
 3:     $proc \leftarrow LMK\_SELECT\_VICTIM(victim\_list)$
 4:     delete $proc$ from $victim\_list$
 5:     for each virtual memory area $vma$ in $proc$ do
 6:       if $vma$ is executable then
 7:         for each page $p$ in $vma$ do
 8:           $s \leftarrow$ allocate a page from NVRAM
 9:           $XIP\_ENGINE(p, s)$
10:           $MEMCPY(p, s)$
11:           $FREE(p)$
12:   else
        /* No more code page.                          */
13:     Use traditional PFRA to swap out inactive pages;
14: else
15:   $proc \leftarrow LMK\_SELECT\_VICTIM(proc\_list)$
      /* Kill process to reclaim memory.               */
16:   $LMK\_KILL\_PROCESS(proc)$;
17: return;
```

Specifically for code pages, we adopt the concept of XIP to reduce these unnecessary memory copy operations between main memory and the swap area. As shown in Fig. 4, using the reversing mapping, we directly set up the page table mappings for the process when its code pages are stored in NVRAM. The code pages will still be available to user space processes, i.e., the binary code in the pages can be executed in-place on the swap area. In this way, we do not need to copy the pages in swap area back to main memory since the code pages are always read-only. Since code pages executed on NVRAM can only be read, we do not perform explicit swap in operations. In this work, pages will be freed when the process is killed by LMK or exit naturally, and thus code pages in the swap area will not be erased until process termination. Thus, writes to the swap area are limited, furthering prolonging the lifetime of the NVRAM based swap area.

4 EVALUATION

We implement a prototype of *n*Code in Google Android 4.4 with the kernel version ARM Linux 3.4 for the Google Nexus 5 smartphone. In the Android Linux kennel, we introduce a new memory zone and serve it as the NVRAM swap area for allocating NVRAM page frames. Cooperating with the traditional PFRA, the code pages identification technique has been integrated within the memory reclaiming routine to select victim pages when the system's

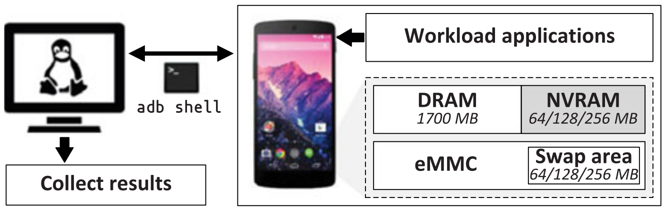


Fig. 5. The experimental setup. We conduct the experiments on Google Nexus 5 and use adb tool to communicate with it.

memory is under pressure. In our experiment, we do not target at any specific type of NVRAM or emulate its latency values. Though different NVRAMs have different performance parameters, we believe that future mobile NVRAM products will generally provide near-DRAM performance. In this section, we first give the experimental setup, and then discuss the experiment metrics and methodology. Finally, we present and analyze the experimental results.

4.1 Experimental Setup

Fig. 5 illustrates the experimental setup. We run all experiments in a Google Nexus 5 smartphone. It features a Qualcomm Snapdragon 800 processor clocked at 2.26 GHz and 2 GB DRAM as main memory. The phone is connected to a desktop PC and the Android debug bridge (adb) is used to communicate with it. Before each experiment, the phone is rebooted and full charged to make sure it is working in its full performance capability.

Since our system works in the OS level, we do not target at any specific type of NVRAM. Though different NVRAM technologies have different performance parameters, some NVRAM technologies, such as STT-RAM and memristor can meet or surpass DRAM's latency. Therefore, in this paper, we do not regard the performance of NVRAM as a major problem, the NVRAM is simulated by using DRAM and the latency difference between DRAM and NVRAM is not concerned. In the experiments, we reserve 256 MB DRAM from the Google Nexus 5's main memory and use it to simulate NVRAM. Therefore, after reserving, we configure the amount of available DRAM for OS to 1700 MB.

We test three different swapping shcemes: (1) flash-based, (2) NVM-Swap, and (3) *n*Code. For flash-based swapping, we use a file in the smartphone's internal NAND flash memory as the swap area[2]. The NVM-Swap is built on NVRAM and use memory interface to swap in and out. It also supports Copy-on-Write swap-in (CoWs) and enables NVRAM wear-leveling. The difference between NVM-Swap and *n*Code is that NVM-Swap uses traditional PFRA, which chooses inactive pages to swap out while *n*Code prefers code pages when selecting victim pages. Moreover, *n*Code supports XIP of code pages in the swap area. We compare *n*Code to NVM-Swap to see the performance and write reduction achieved by *n*Code. For all these swapping scheme, we configure three different swap sizes—64 MB, 128 MB and 256 MB.

Table 3 illustrates the workload applications we used to evaluate *n*Code. As different kinds of applications may exhibit different memory access patterns, we prepared six categories of applications, including browser, social networking, multimedia, gaming, online shopping and news. To represent the realistic scenarios, two mixed categories, namely mix1 and mix2 have been added by combining the applications selected from the above six categories. For each test, we run applications in the same category in round-robin order.

TABLE 3
Workload Applications

Category	Applications
Browser	Firefox, Chrome, Opera, UC Browser, Next Browser
Social networking	Google+, Pinterest, Twitter, Facebook, Instagram
Multimedia	Google Play Music, MX Player, TTpod Player, Youtube, KMPlayer
Gaming	Angry Birds, Ingress, Temple Run, Crazy Snowboard, Hill Climb Racing
Online shopping	Amazon, Ebay, Fancy, Google Play, TaoBao
News	BBC News, Flipboard, NetEase News, TED, Zaker
Mix1	Chrome, Pinterest, Youtube, Ingress, NetEase News
Mix2	Firefox, Facebook, Angry Birds, Amazon, BBC News

4.2 Metrics and Methodology

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

1) Number of swap-outs: To reduce the writes to NVRAM swap area, *n*Code selects code pages to swap out and supports XIP of code pages in swap area. We use this metric to evaluate the effect of write reduction of *n*Code. To collect this metric, we run each application of a certain category in foreground for 1 minute, and all the applications in that category are run in round robin order for four times. Thus, 20 minutes are required for each application category to accomplish the evaluation. We compare the number of swap-outs between *n*Code and NVM-Swap.

2) Application relaunching delay: Application relaunching delay is an important performance metric for smartphone users. Application relaunching may cause victim 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. We use this metric to evaluate the performance of *n*Code, and compare the results between different swapping schemes. To measure the application relaunching delay, we use Swap-Bench [26] to perform applications auto switching and results collecting.

2) Number of page fault: Page faults in applications will interrupt the process execution, wake up the kernel and make it to handle the faults. We therefore use it as a performance metric. To collect this metric, We use the UI/Application Exerciser Monkey to generate 5,000 events to each application and add a counter in the Android Linux kernel to count the total number of page faults in each application category.

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.

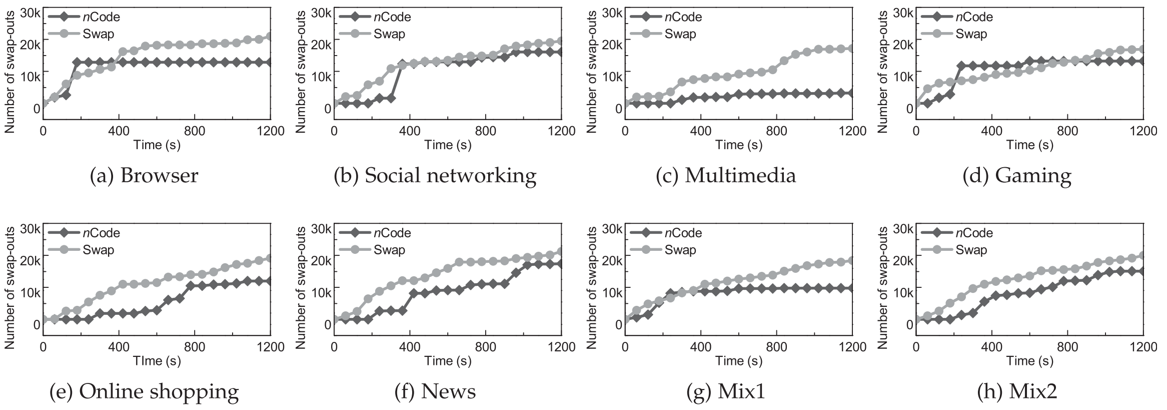


Fig. 6. Comparison of the number of swap-outs between *n*Code and NVM-Swap for each application category when swap size = 64 MB.

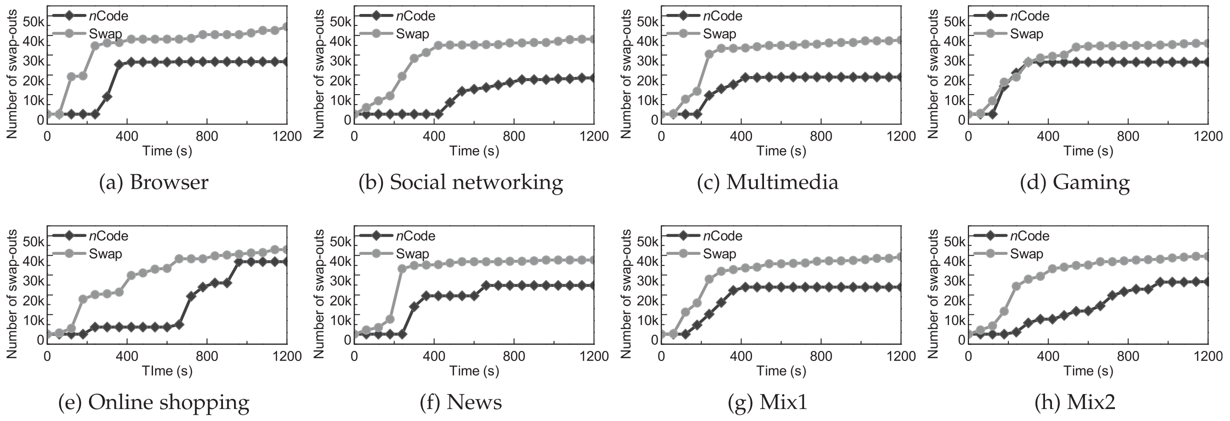


Fig. 7. Comparison of the number of swap-outs between *n*Code and NVM-Swap for each application category when swap size = 128 MB.

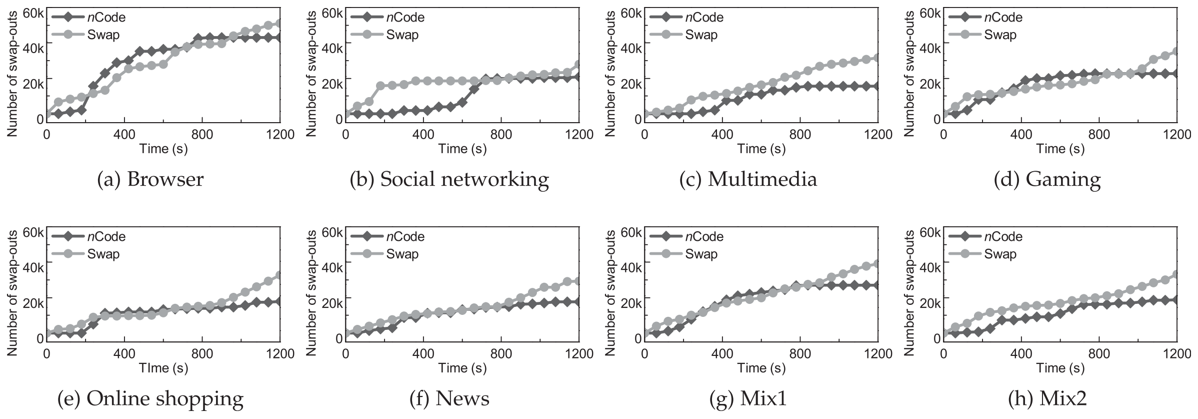
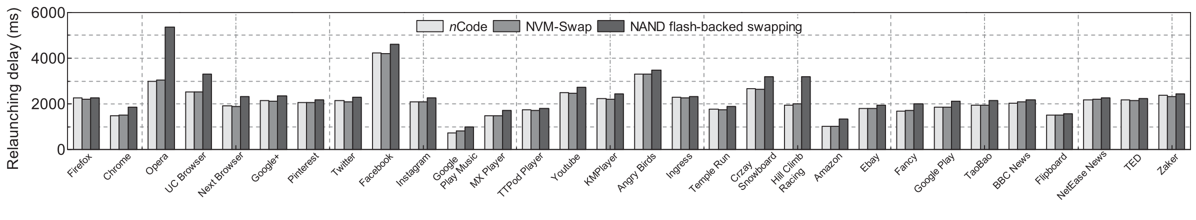


Fig. 8. Comparison of the number of swap-outs between *n*Code and NVM-Swap for each application category when swap size = 256 MB.

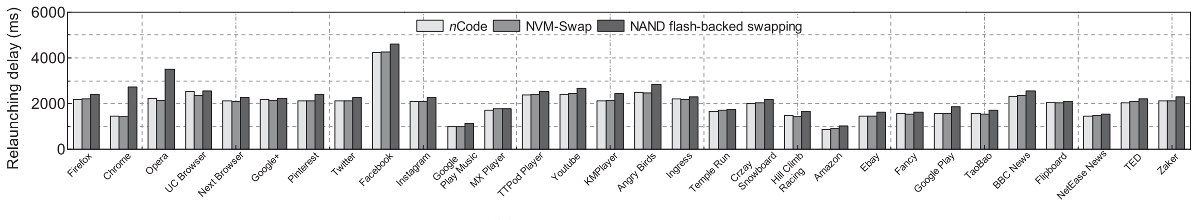
4.3 Number of Swap-Outs

Figs. 6, 7, and 8 compare the number of swap-outs between *n*Code and NVM-Swap under different swap capacities. As shown in the figures, compared to NVM-Swap, *n*Code can reduce around 14–51 percent writes to the NVRAM swap area. In *n*Code, by exploiting the read-only property and NVRAM's byte-addressability, we migrate application's code pages to the swap area when the system's memory is under pressure, and execute the code pages in-place on the swap area. Therefore, swap-ins are avoided, and this further avoids page thrashing, which will increase the writes to swap area.

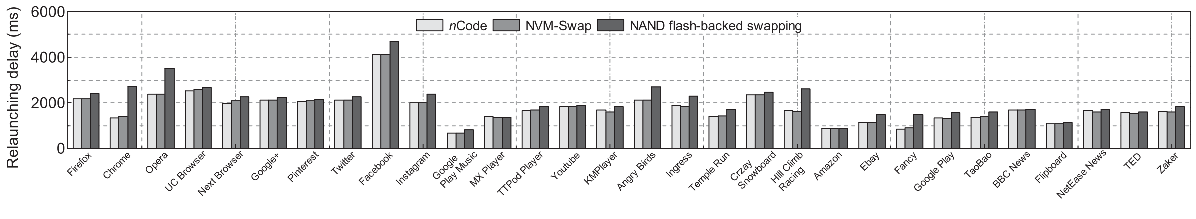
As shown in the figure, along with the time increasing on the *x*-axis, the number of swap-outs in *n*Code gradually reaches a stable state—no more writes to swap area or the writes are increased slowly. In Figs. 6, 7, and 8, we note that the number of swap-outs almost stayed the same after a certain period of time, especially for browser and gaming applications in Fig. 6, and browser, multimedia, gaming and news applications in Fig. 7. On the contrary, the number of swap-outs in NVM-Swap keeps growing, though the growth rate has slowed down after certain point. The major reason behind is that when swapped out pages are written to, we first need to allocate new pages and then copy them



(a) Swap size = 64MB



(b) Swap size = 128MB



(c) Swap size = 256MB

Fig. 9. Comparison of relaunching delay between different swapping schemes under different capacities.

back to DRAM. However, the requests for new pages may cause other pages to be swapped out. Thus, the number of swap-outs in NVM-Swap will keep growing over time.

In *n*Code, a small swap area makes applications be killed frequently and thus code pages will be continuously written to NVRAM when the space in swap area is reclaimed due to process kills. On the contrary, with a large NVRAM swap area, there would be excessive NVRAM space since the code pages cannot use up all the NVRAM. In this situation, inactive pages will be swapping out when the memory is under pressure, which will bring page thrashing and consequently introduces more writes. Since each application's code pages take around 20 percent of its corresponding DRAM pages, in this work, we believe that the size of NVRAM swap area should depend on the size of DRAM. In our experiments, the memory capacity of Nexus 5 is set to 1,700 MB, in which the Android OS occupies around 500-600 MB (in our evaluation). Therefore, the available memory for applications is around 1,100-1,200 MB and a 256 MB NVRAM swap area is sufficient to store all application's code pages.

4.4 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 or swap area if part of the application's data are swapped out. Therefore, different swapping schemes will lead to different relaunching delay. Fig. 9 shows the application relaunching delay under different swapping schemes. As shown in the figure, *n*Code achieves almost the same relaunching delay when compared with NVM-Swap, indicating that *n*Code can achieve the same performance as NVM-Swap achieved. While compared to NAND flash-based swapping, *n*Code reduces the relaunching delay by around 10–40 percent.

In NAND flash-backed swapping, 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 NVRAM, making NAND flash-backed swapping exhibits the highest relaunching delay. On the contrary, both *n*Code and NVM-Swap use pure memory copy operations to write/read pages from/to swap area, resulting in a much faster application relaunching. In particular, the relaunching of Chrome and Opera in *n*Code and NVM-Swap are around 40 percent faster than that in NAND flash-based swapping, which means a great improvement on user experience.

4.5 Number of Page Faults

Table 4 compares the number of page faults between *n*Code and NVM-Swap. As shown, *n*Code reduces around 20 percent number of page faults on average compared to NVM-Swap, further reducing the time cost by page fault handler and improving application performance. In *n*Code, code pages in NVRAM based swap area are accessed directly with XIP, thus no page faults occur when accessing pages in NVRAM. On the contrary, even through NVM-Swap features CoWs, page faults still will be triggered when access swapped out pages, make the kernel to setup the page table entries for the requested pages or copy the requested pages from swap space back to DRAM and then update the mapping.

5 Related Work

There are various of previous work related to *n*Code. Most of the related work lies in the following three areas: swapping, NVRAM endurance prolonging and hybrid NVRAM/

TABLE 4
Comparison of Number of Page Faults Between *n*Code and NVM-Swap

Category	Swap size = 128MB			Swap size = 256MB			Swap size = 512MB		
	NVM-Swap	*n*Code	↓ (%)	NVM-Swap	*n*Code	↓ (%)	NVM-Swap	*n*Code	↓ (%)
Browser	3,943,227	3,323,854	15.71	3,002,445	2,525,647	15.88	2,512,004	2,060,030	17.99
Social networking	3,113,747	2,211,451	28.98	2,283,142	1,622,312	28.94	1,912,176	1,530,295	19.97
Multimedia	3,533,948	2,635,189	25.43	2,662,082	2,011,507	24.44	2,202,907	1,689,455	23.31
Gaming	3,419,200	2,743,878	19.75	2,666,061	2,093,847	21.46	2,096,170	1,726,954	17.61
Online Shopping	3,369,403	2,989,436	11.28	2,476,821	2,340,482	5.50	2,220,993	1,819,195	18.09
News	3,154,903	2,344,854	25.68	2,465,836	1,844,587	25.19	2,121,985	1,552,561	26.83
Mix1	3,503,370	3,017,570	13.87	2,578,311	2,210,281	14.27	2,014,215	1,752,430	13.00
Mix2	3,267,316	2,677,574	18.05	2,551,028	2,013,314	21.08	2,102,842	1,680,917	20.06

DRAM main memory. In this section, we discuss these related work separately.

Swapping. To extend memory capacity, many swapping system have been proposed for both mobile devices and servers. NVM-Swap [11], DR. Swap [12], [13] and Refinery Swap [27] re-adopt swapping using NVRAM for better performance and energy behavior of smartphones. MARS [28] speed-up the application relaunching by exploiting a flash-aware page swapping system. FASS [29] implements a raw flash memory based swapping system without using a flash translation layer. FlashVM [30] is another flash backed swapping that integrates flash memory with virtual memory and provides better garbage collection by batching writes. SSDAlloc [31] is an SSD/DRAM hybrid system which extends DRAM with SSD and allows programmers to treat SSD as DRAM. To eliminate the performance gap between main memory and flash based swap area, in this paper we adopt NVRAM as swap area and use memory interface to swap in and out. Different traditional wisdom, which chooses inactive pages to swap out when the memory is under pressure, *n*Code selects code pages and executes code pages in place on the NVRAM swap area.

NVRAM endurance prolonging. Many researches have been conducted to solve the endurance issue of NVRAM. Chen et al. [20] introduced an age-based wear leveling scheme which is compatible with existing virtual memory design. Start-Gap [21] and segment swapping [18] are two representative wear leveling algorithms that evenly distribute writes among all cells of PCM-based main memory. NVM-Swap [11] features Heap-Wear, a heap-based wear leveling scheme for NVRAM-based swapping. Hu et al. [32] focus on reducing writes on NVRAM for embedded CMPs. Qureshi et al. [33] proposed a set of techniques such as lazy write and line level writeback to reduce writes. Khouzani et al. [34] presented a segment-aware and wear-resistant page allocation method to prolong the PCM lifetime. Flip-N-Write [22] and DCW [23] try to reduce the number of bit flips in NVRAM cells. [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. Ferreira et al. [38] increase PCM lifetime by swapping pages on page cache writebacks. In [39], Khouzani et al. proposed a hierarchical hybrid DRAM/PCM memory architecture, where DRAM is served as the cache of PCM. To reduce the writes to PCM, a proactive page allocation algorithm is proposed to distribute heavily written pages across different DRAM sets. Different from this approach, *n*Code is a parallel organization and DRAM is adopted as the main memory while NVRAM is used as the swap area. Moreover, in this paper, we aim to build a NVRAM-aware swapping, and we reduce the writes to NVRAM by swapping out code pages.

Hybrid NVRAM/DRAM main memory. Due to its low standby power, high density and byte addressability, NVRAM, such as PCM, are considered as promising DRAM alternative [7], [17]. In [8], [40], a hybrid PCM/DRAM main memory system is proposed, in which pages can be transferred between PCM and DRAM for saving energy and improving PCM endurance. Qureshi et. al [33] design a hybrid DRAM/PCM main memory where DRAM is invisible and used as on-chip cache while PCM is served as main memory. Similarly, Chang et al. [41] use DRAM as a cache for the PCM-based main memory but the DRAM is software-controlled. Hassan et al. [42] proposed a software-only approach to reduce the energy consumption in hybrid DRAM/NVRAM memory. To improve the memory bandwidth and reduce the system energy consumption, Zhao et al. [43] proposed a hybrid GPU memory architecture by combining different memory technologies (DRAM, STT-RAM, and RRAM). Compared to these hybrid approaches, trough *n*Code has the same or similar hardware architecture, the management strategy is totally different. In hybrid approaches, NVRAM is treated as part of the main memory while in *n*Code, DRAM is served as the main memory and NVRAM is dedicated as the swap area.

6 Conclusions

In this paper, we have proposed *n*Code, an NVRAM based swapping system that aims to improve the system's performance and reduce writes to NVRAM. Different from traditional page frame reclaim algorithm, which selects inactive pages and introduces page thrashing, *n*Code priorities code pages as swapping candidates in smartphones since code pages are read-only—perfect candidates to be swapped out to write-limited NVRAM. To avoid page thrashing and further reduce writes to NVRAM swap area, we support XIP of code pages on the swap space without copying the code pages back to DRAM by utilizing the byte-addressability of NVRAM. Currently, a prototype of *n*Code has been implemented in Google Nexus 5 smartohone. Experimental results with various real applications show that when compared to NVM-Swap, *n*Code can respectively reduce the swap-outs and page faults by 30 and 20 percent on average.

The application relaunching delay has also been reduce by around 10–40 percent when compared to NAND flash-based swapping.

Acknowledgements

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). A preliminary version of this paper was presented at the ACM/IEEE 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE) [1].

References

[1] 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.

[2] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *ACM Trans. Storage*, vol. 8, no. 4, pp. 1–25, 2012.

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

[4] M. Osomi, et al., "A novel Nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM," in *Proc. IEEE Int. Electron Devices Meeting Tech. Dig.*, 2005, vol. 459, pp. 459–462.

[5] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[6] Intel and micron produce breakthrough memory technology, 2015. [Online.] Available: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/

[7] 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*, 2012, pp. 398–403.

[8] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid pram and dram main memory system," in *Proc. 46th ACM/IEEE Design Autom. Conf.*, 2009, pp. 664–669.

[9] 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 .

[10] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proc. 17th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 401–410.

[11] K. Zhong, et al., "Building high-performance smartphones via non-volatile memory: The swap approach," in *Proc. Int. Conf. Embedded Softw.*, 2014, Art. no. 30.

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

[13] 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.

[14] K. Yan, X. Zhang, and X. Fu, "Characterizing, modeling, and improving the GoE of mobile devices with low battery level," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 713–724.

[15] C. Xue, G. Sun, Y. Zhang, J. Yang, Y. Chen, and H. Li, "Emerging non-volatile memories: Opportunities and challenges," in *Proc. 9th Int. Conf. Hardware/Software Codesign Syst. Synthesis*, 2011, pp. 325–334.

[16] Y. Wang, et al., "A compression-based area-efficient recovery architecture for nonvolatile processors," in *Proc. Des., Autom. Test Europe Conf. Exhib.*, 2012, pp. 1519–1524.

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

[18] 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.

[19] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha, "Curling-PCM: Application-specific wear leveling for phase change memory based embedded systems," in *Proc. 18th Asia South Pacific Des. Autom. Conf.*, 2013, pp. 279–284.

[20] 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.

[21] 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.

[22] 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.*, 2009, pp. 347–357.

[23] 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.

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

[25] Y. Choi, et al., "A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2012, pp. 46–48.

[26] 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.

[27] X. Chen, et al., "The design of an efficient swap mechanism for hybrid DRAM-NVM systems," in *Proc. 13th Int. Conf. Embedded Softw.*, 2016, pp. 22:1–22:10.

[28] 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.

[29] 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.

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

[31] 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.

[32] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Write activity reduction on non-volatile main memories for embedded chip multiprocessors," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 3, pp. 1–27, 2013.

[33] 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.

[34] H. A. Khouzani, Y. Xue, C. Yang, and A. Pandurangi, "Prolonging PCM lifetime through energy-efficient, segment-aware, and wear-resistant page allocation," in *Proc. Int. Symp. Low Power Electron. Des.*, 2014, pp. 327–330.

[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 Eur. 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] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Moss, "Increasing PCM main memory lifetime," in *Proc. Des., Autom. Test Europe. Conf. Exhib.*, 2010, pp. 914–919.

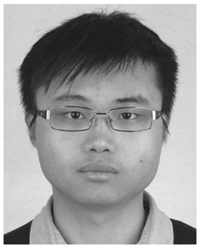
[39] H. A. Khouzani, C. Yang, and J. Hu, "Improving performance and lifetime of DRAM-PCM hybrid main memory through a proactive page allocation strategy," in *Proc. 20th Asia South Pacific Des. Autom. Conf.*, 2015, pp. 508–513.

[40] 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.

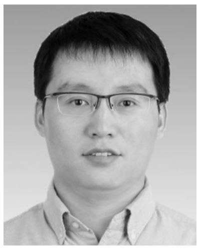
[41] H.-S. Chang, Y.-H. Chang, T.-W. Kuo, and H.-P. Li, "A light-weighted software-controlled cache for PCM-based main memory systems," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2015, pp. 22–29.

[42] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Software-managed energy-efficient hybrid DRAM/NVM main memory," in *Proc. 12th ACM Int. Conf. Comput. Frontiers.*, 2015, pp. 23:1–23:8.

[43] J. Zhao and Y. Xie, "Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration," in *Proc. Int. Conf. Comput.-Aided Des.*, 2012, pp. 81–87.

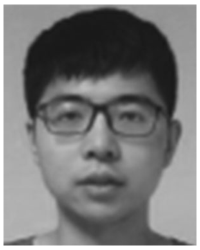


Kan Zhong received the BSc degree in computer science from Chongqing University, Chongqing, China, in 2013, where he is currently working toward the PhD degree. His current research interests include mobile computing, embedded system, and non-volatile memory.



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.

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 in the 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.



Jinting Ren received the BSc degree in computer science from Chongqing University, Chongqing, China, in 2016, where he is working toward the PhD degree. His current research interests include approximate computing and embedded system.

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.

Edwin Hsing-Mean Sha received the PhD degree from the Department of Computer Science, Princeton University, in 1992. From August 1992 to August 2000, he was in the Department of Computer Science and Engineering, University of Notre Dame. Since 2000, he has been a tenured full professor in the Department of Computer Science, University of Texas at Dallas. Since 2012, he served as the dean of College of Computer Science, Chongqing University, China. He has published more than 280 research papers in refereed conferences and journals. He has served as an editor for many journals, and as program committee and Chairs for numerous international conferences. He received Teaching Award, Microsoft Trustworthy Computing Curriculum Award, NSF CAREER Award, and NSFC Overseas Distinguished Young Scholar Award, Chang Jiang Honorary Chair Professorship and China Thousand Talents Program.