*n*Code: Limiting Harmful Writes to Emerging Mobile NVRAM through Code Swapping

Kan Zhong, Duo Liu*, Linbo Long, Xiao Zhu, Weichen Liu, Qingfeng Zhuge, Edwin H.-M. Sha
Key Lab. of Dependable Service Computing in Cyber Physical Society (Chongqing Univ.),
Ministry of Education, China
College of Computer Science, Chongqing University, Chongqing, China
{kzhong1991, liuduo}@cqu.edu.cn

Abstract— Mobile applications are becoming more and more powerful but also dependent on large main memories, which consume a large portion of system energy. Swapping to byte-addressable, non-volatile memory (NVRAM) is a promising solution to this problem. However, most NVRAMs have limited write endurance. 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. 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.

I. Introduction

Mobile devices like smartphones are always short for main memory as applications get more and more powerful and resource demanding. Having larger main memory, which is DRAM based, is effective but also drains the limited battery 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 [1]. 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.

With emerging byte-addressable non-volatile memory (NVRAM) such as phase change memory (PCM) [2], spin-transfer torque RAM (STT-RAM) [3] and memristor [4], swapping is no more a deal-breaker for smartphone performance and user experience. Swapping to NVRAM [5] 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 [6].

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. Any writes to NVRAM that are not necessary are considered harmful. When swapping to NVRAM, one must avoid as many harmful writes as possible and proactively reduce writes on NVRAM.

The conventional wisdom is hot/cold data identification. Identifying "cold" data—those that are not frequently modified—is both lightweight and proactive as in this way, we can control which data can be placed in NVRAM and DRAM main memory. Since cold data are not accessed frequently, after being swapped out, they will not be swapped in in a short time. Therefore, swapping out cold data can preserve the lifetime of NVRAM. After cold data are identified, they are given a higher priority to be swapped out to NVRAM. With application-specific knowledge, hot and cold data can be identified and treated differently in an almost perfect manner [7]. But this is a non-trivial process for smartphone applications: different applications have vastly different data access patterns. Obtaining proper hot/cold data identification for smartphones without application-specific information and make it adaptive to different workloads are difficult, if not impossible. It also brings significant run-time overheads to detect and adapt to data access patterns.

Though identifying data access patterns and adapting applications to them are difficult, identifying *code pages* is very straightforward—they are the executables stored in smartphone storage. Moreover, code pages are read-only, making them a perfect candidate for swapping to write-limited NVRAM. With these insights, we propose to build a system called "*n*Code" to reduce harmful 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 [5], [6] to allow execute-in-place of code pages on the swap space. Since NVRAM is byte-

*Corresponding author: Duo Liu, College of Computer Science, Chongqing University, Chongqing, China. E-mail: liuduo@cqu.edu.cn.

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 traditional flash based and NVRAM based swapping systems, furthering improving the endurance of NVRAM.

We make the following contributions:

- 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 II outlines background on swapping and emerging NVRAM in mobile devices. Section III details the design of *n*Code. Evaluation results are presented and analyzed in Section IV. Finally, we discuss related work and conclude in Sections V and VI, respectively.

II. Background

In this section, we first introduce NVRAM and its endurance problem, and then give the background on swapping in mobile devices, especially in smartphones.

A. NVRAM

Byte-addressable non-volatile memory (NVRAM) such as PCM [2], STT-RAM [3] and memristor [4], 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 [8], [9]. 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 I, PCM exhibits much lower power consumption and similar read latency but longer write latency when compared to DRAM. More importantly, it is write-limited, i.e., each PCM cell can only be programmed for a limited 10^8–10^9 times [10], [11], 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 [7], [12], [13], [14]. Write reduction tries reduce the number of writes or the number of bit flips to memory cells [15], [16]. Alone with PCM, memristor is another promising NVRAM and has the potential of providing at least DRAM performance. Recently, many researches are based on memristor, thus as memristor based approximation computing [17] and artificial neural network [18]. In this paper, we don't target at any specific NVRAM products, instead, we focus on the software aspects as *n*Code works on the OS level.

TABLE I: Comparison of PCM and DRAM [19], [20].

Attributes	DRAM	PCM
Read Latency	∼50ns	50 - 100ns
Write Latency	∼20 - 50ns	∼1µs
Power	∼W/GB	100→500mW/die
Idle power	∼W/GB	≪0.1W
Endurance (write cycles)	10^{15}	$10^8 - 10^9$

B. Swapping

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. A natural solution to this problem is swapping, which is useful for extending memory capacity without adding more DRAM. When the system is under memory pressure, a page frame reclaim routine will write inactive pages to the swap area. In smartphones, flash memory is usually used as the swap area due to their low cost and high capacity. However, swapping memory pages in and out of the flash based swap area must go through all the I/O stack. Therefore, the performance of swapping is mainly decided by the performance of the underlying storage medium.

Due to the sub-optimal performance of flash memory, swapping is usually disabled in smartphones. A recent solution is using NVRAM as the swap area [5]. Since NVRAM has near-DRAM performance, swapping will not degrade system performance. But to make this practical, a must-solve problem is the endurance of NVRAM. 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. It is well-known that without application semantics, it is difficult for the OS to identify whether a page is active or not. However, we found that page type is usually available to the OS, and it is easy for the OS to differentiate code and data pages. Therefore, we argue that a better solution is to swap out code pages to the swap area and execute the code in-place, without copying them back to main memory. We illustrate the design and evaluation of this architecture in the rest of this paper.

III. *n*Code Design

In this section, we first give an overview of *n*Code. We then discuss the key techniques behind *n*Code: (1) identifying code pages in the process address space at runtime, and (2) providing support for XIP of code pages in the swap area.

A. Overview

We illustrates the system architecture of *n*Code in Fig. 1. In an NVRAM based swapping system, the swap area is directly attached to the memory bus and accessed through

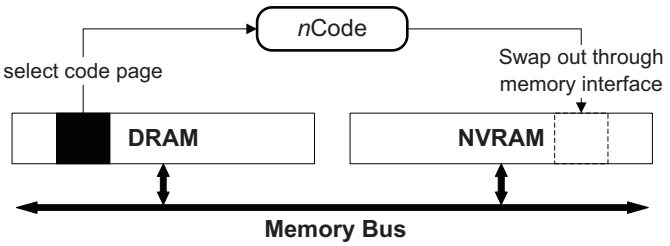


Fig. 1: System architecture. Swap area is backed by NVRAM and memory interface is used to swap out victim code pages selected by *n*Code.

`load` and `store` instructions, instead of block I/O. Different types of pages are swapped out of and into the swap area. They have very diverse access characteristics. In particular, code pages contain program binary code or mapped shared libraries. 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 the system is under memory pressure, i.e., the amount of free memory is below a predefined threshold, *n*Code will start to scan the virtual address space of all running processes and select their code pages as candidates to be swapped out. When these pages are accessed again, we execute the code in-place on the swap area without copying these code pages back to main memory by utilizing the byte-addressability of NVRAM.

B. Identifying Code Pages

The identification of code pages (i.e., selection of code pages as swapping candidates) is accomplished through the page frame reclamation algorithm (PFRA), which is invoked when the system is under memory pressure, e.g., when the amount of free memory is below a predefined threshold. A traditional PFRA maintains an active and inactive list during normal processing. When the system is under memory pressure, pages in the inactive list are selected to be swapped out. 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 [6]. 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. In particular, our implementation reuses Android's low memory killer (LMK), which tends to choose a low priority process with high memory usage. The LMK was originally for terminating applications to make room for incoming memory allocation requests. Instead of maintaining active and inactive lists of pages found in traditional PFRAs, *n*Code utilizes LMK's victim selection mechanism to determine which process' code pages to swap out. After a process is selected, we scan its virtual address space to find out all code pages (i.e., those that are marked `executable`). We then migrate all the resulted pages to the NVRAM swap area.

Algorithm III.1 shows the details of migrate code pages of a certain process to the NVRAM swap area. Usually the memory space belonging to a process consists of multiple

Algorithm III.1 $MigrateCodePages(proc)$

Input: $proc$: the process selected to be migrated.
Output: $null$

```
1:  for each virtual memory range v in proc do
2:    if v is executable then
3:      for each page in v do
4:        newpage ← allocate new page in NVRAM.
5:        update all the PTEs for page via reverse mapping.
6:        memcpy(newpage, page).
7:        free(page).
8:      end for
9:    end if
10: end for
```

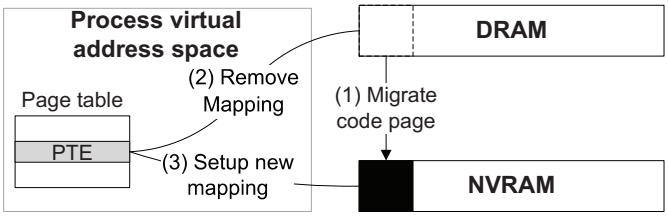


Fig. 2: XIP of code pages. (1) The code page is swapped out from DRAM to NVRAM; (2) Remove the mapping for the code pages and free them; (3) Set up new page table mappings for the code pages in NVRAM, to allow the code to be executed in-place on NVRAM.

virtual memory ranges, which are organized using a linked list. Each range has multiple physical page frames mapped through page tables. Each range 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). Our algorithm scans the given process' memory ranges and select the executable ranges. For each page in the selected ranges, we allocate a new page in NVRAM (line 4), redirect the virtual addresses that were pointing to the DRAM page to the NVRAM page (line 5), and copy the contents to it (line 6). The DRAM page is then freed to the OS memory manager for future memory allocations (line 7).

C. 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.

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. 2, 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. Furthermore, as the code pages in the swap area will not be erased until process termination, writes to the swap area are limited, furthering prolonging the lifetime of the NVRAM based swap area.

IV. EVALUATION

We have implemented *n*Code into Android's Linux kernel for the Google Nexus 5 smartphones. 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.

A. Experimental Setup

We run all the experiments on a Google Nexus 5 smartphone. TABLE II gives the details of experiment setup. We connect the Google Nexus 5 smartphone to a desktop PC and use the Android debug bridge (`adb`) provided in the Android SDK to communicate with it.

Because NVRAM products are not yet widely available on the market, in our experiments, we implement the NVRAM based swap area using 128MB DRAM. We use the default swapping scheme implemented in the Linux kernel as a baseline, and compare it to the proposed technique. We use the default setting in Linux kernel for selecting victim process by LMK in *n*Code and the default swappiness value in baseline swapping scheme. For fair comparison, instead of flash memory, we use a RAM-disk of the same size as the swap area for the baseline scheme. In this paper, we focus on the endurance issue of NVRAM and we hope that future NVRAM products will provide near-DRAM performance. In TABLE II we list six types of popular applications used in our experiments. Each category includes five applications, including browser, gaming, news, etc. In our experiments, we disable all the radio communication functionality except Wi-Fi as most applications need Internet connection to work properly. Before each test, we reboot the phone and wait for a few seconds to ensure each round of experiment starts with roughly the same state.

B. Metrics and Methodology

We collect the results for the metrics listed in this section. The methodologies are detailed as we discuss the corresponding metrics. For each metric, each application in the same category is run for four times.

Number of swap-outs. This metric measures the effect of write reduction using *n*Code. For each test, we run all the applications in each category for 20 minutes and count the number of swap-outs. We compare the number of swap-outs in *n*Code and the baseline scheme.

Application switching delay. Application switching delay is an important metric for smartphone users. We use the UI/Application Exerciser Monkey in Android to automate the switching between foreground applications and use the Linux `time` command to obtain the time consumed to finish each switching. When an application is switched to foreground, we also use Monkey to generate a stream of 100 user events such as touches and clicks to that application. We start the recording of application switching delay when the switching is triggered, and end when the 100 user events are applied to that application. The results are compared to a swap disabled scheme to show the switching delay reduction of *n*Code.

Number of page fault. Page faults in applications will wake up the kernel and make kernel spend some time to handle the faults. It's important to evaluate the number of page faults in *n*Code and baseline scheme. We again use the UI/Application Exerciser Monkey to generate 5000 events to each application and count the total number of page faults in each category.

TABLE II: Experimental setup.

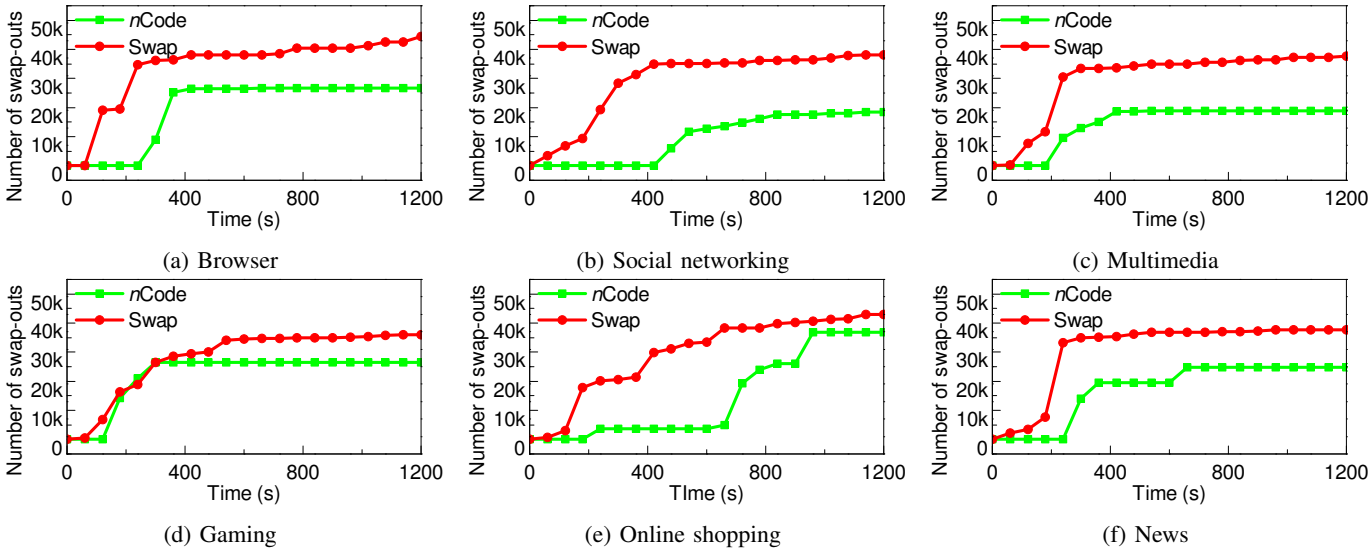
Nexus 5	CPU	Snapgragon 8974 @2.26GHz
	DRAM	2GB LPDDR3
	Storage	16GB eMMC NAND flash
	OS	Android 4.4 with kernel 3.4
Workload applications	**Category**	**Applications**
	Browser	Firfox, Chrome, Opera, UC Browser, Next Browser
	Social networking	Google+, Pinterest, QQ, Sina Weibo, Instagram
	Multimedia	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

TABLE III: Comparison of total number of swap-outs between *n*Code and baseline scheme.

Category	**Number of swap-out**		**Reduction (%)**
	Swap	*n*Code	
Browser	44403	26591	40.11
Social networking	38077	18481	51.46
Multimedia	37695	18945	49.74
Gaming	36031	26482	26.50
Online shopping	43034	36787	14.41
News	37714	24823	34.18

C. Results

Fig. 3 shows the number of swap-outs for both *n*Code and the baseline scheme. As we can see, compared to baseline scheme, *n*Code reduces a large amount of writes to NVRAM. TABLE III gives the total number of swap-outs under both configurations. *n*Code can reduce around 14%–51% writes to the NVRAM swap area when compared to the baseline scheme. In *n*Code, we migrate code pages to the swap area. Exploiting the read-only property and NVRAM's byte-addressability, we execute the code in-place on the swap area. Thus, we have avoided swap-ins and also limited the writes to NVRAM. As time goes by on the x-axis in Fig. 3, *n*Code gradually reaches a stable state—no more swap-outs. In Fig. 3, we note that the number of swap-outs almost stayed the same after a certain period of time, especially for browser, multimedia, gaming and news applications. On the contrary, the number of swap-outs in the baseline scheme kept growing, though the growth rate has slowed down after certain point. The major reason behind is that when swapped out pages are accessed, we first need to allocate new pages and then copy them back to DRAM. However, the requests for new pages



(a) Browser (b) Social networking (c) Multimedia

(d) Gaming (e) Online shopping (f) News

Fig. 3: Comparison of the number of swap-outs between *n*Code and baseline scheme in each application category.

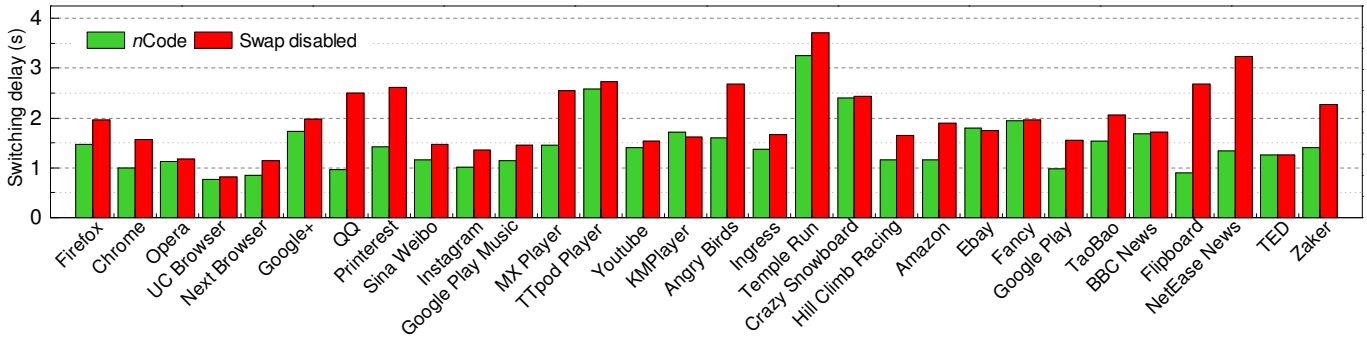


Fig. 4: Comparison of average switching delay between *n*Code and swap disabled scheme for each application.

TABLE IV: Comparison of average switching delay between *n*Code and the swap disabled scheme in each category.

Category	Avg. switching delay (s)		Reduction (%)
	Swap disabled	*n*Code	
Browser	1.33	1.04	21.80
Social networking	1.98	1.26	36.36
Multimedia	1.98	1.66	16.16
Gaming	2.43	1.96	19.34
Online shopping	1.84	1.48	19.57
News	2.24	1.32	41.07

may cause other pages to be swapped out. Thus, the number of swap-outs in baseline scheme will keep growing over time.

Fig. 4 shows the switching delays for each application under different configurations. TABLE IV lists the average switching delay for each application category. *n*Code can reduce around 19%–41% switching delays on average for each application category. As shown in Fig. 4, for some applications, we have reduced the average switching delay by more than 50%, such as QQ, Flipboard and NetEase News. In the swap disabled scheme, applications may get terminated by the LMK when there is no sufficient memory. Thus, switching between those applications could take longer time as they will have to be reloaded from flash storage. In *n*Code, the main memory is extended by migrating code pages to the swap area. Moreover, code pages are executed in-place on the swap area. Thus, applications running with *n*Code will have less chance to be terminated by LMK.

Fig. 5 shows the number of page faults for both *n*Code and the baseline scheme. As shown, *n*Code reduces around 20% number of page faults on average compared to baseline scheme. 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, when applications access swapped out pages in baseline scheme, the underlying hardware will triggers page faults to make the kernel copy the pages from swap space back to DRAM and then sets up the page table entries for the applications. Since *n*Code reduces the number of page faults compared to baseline swapping scheme, furthering reducing the time cost by page fault handler. We conclude that *n*Code can help improve the performance of applications.

V. Related Work

To extend memory capacity, many designs have been proposed for both mobile devices and servers. NVM-Swap [5] and DR. Swap [6] re-adopt swapping using NVRAM for better performance and energy behavior of smartphones. FASS [21] is a raw flash based swapping system without using a flash translation layer. FlashVM [22] is a dedicated flash swapping design and provides better garbage collection by batching writes. Li et al. [23] revisits the virtual memory design with

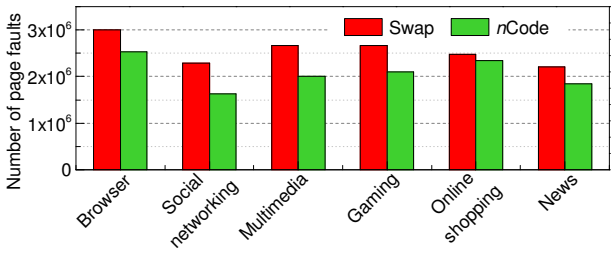


Fig. 5: Comparison of number of page faults between *n*Code and baseline scheme in each application category.

flash in embedded system and focuses on energy behavior of the system. SSDAlloc [24] is an SSD/DRAM hybrid system which treats SSD as an extension to DRAM.

Improving the endurance of NVRAM is another active area of research. Chen et al. [12] introduced an age-based wear leveling scheme which is compatible with existing virtual memory design. Start-Gap [13] and segment swapping [11] are two representative wear leveling algorithms that evenly distribute writes among all cells of PCM-based main memory. NVM-Swap [5] features Heap-Wear, a heap-based wear leveling scheme for NVRAM-based swapping. Hu et al. [25] focus on reducing writes on NVRAM for embedded CMPs. Qureshi et al. [26] proposed a set of techniques such as lazy write and line level writeback to reduce writes. Khouzani et al. [27] presented a segment-aware and wear-resistant page allocation method to prolong the PCM lifetime. Flip-N-Write [15] and DCW [16] try to reduce the number of bit flips in NVRAM cells. In this work, we reduce writes by swapping out code pages in NVRAM based swapping system.

VI. Conclusions

In this paper, we have proposed *n*Code, an NVRAM based swapping system that aims to reduce harmful writes to NVRAM. *n*Code priorities code pages as swapping candidates in smartphones. *n*Code selects code pages as they are read-only—perfect candidates to be swapped out to write-limited NVRAM. Utilizing the byte-addressability of NVRAM, we support XIP of code pages on the swap space without copying the code pages back to DRAM. Experimental results with various real applications on the Google Nexus 5 smartphone show that *n*Code can reduce more than 30% writes to NVRAM on average when compared to the baseline swapping scheme. Application switching delay is reduced by more than 20% on average with *n*Code enabled. Number of page faults are also reduced around 20% on average compared to the baseline swapping scheme.

Acknowledgment

We would like to thank 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 (6130900461472052, and 61173014), National 863 Program (2013AA013202), Research Fund for the Doctoral Program of Higher Education of China (20130191120030), Chongqing cstc2012ggC40005 and cstc2013jcyjA40025, Fundamental Research Funds for the Central Universities (CDJZR14185501).

References

[1] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *ACM Transactions on Storage*, vol. 8, no. 4, pp. 1–25, 2012.
[2] 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*, vol. 98, no. 12, pp. 2201–2227, 2010.
[3] 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 *IEDM*, 2005, pp. 459–462.
[4] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.
[5] K. Zhong, T. Wang, X. Zhu, L. Long, D. Liu, W. Liu, Z. Shao, and E. Sha, "Building high-performance smartphones via non-volatile memory: The swap approach," in *EMSOFT*, 2014.
[6] 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 *ISLPED*, 2014, pp. 81–86.
[7] 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 *ASP-DAC*, 2013, pp. 279–284.
[8] C. Xue, G. Sun, Y. Zhang, J. Yang, Y. Chen, and H. Li, "Emerging non-volatile memories: Opportunities and challenges," in *CODES+ISSS*, 2011, pp. 325–334.
[9] Y. Wang, Y. Liu, Y. Liu, D. Zhang, S. Li, B. Sai, M.-F. Chiang, and H. Yang, "A compression-based area-efficient recovery architecture for nonvolatile processors," in *DATE*, 2012, pp. 1519–1524.
[10] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *ISCA*, 2009, pp. 2–13.
[11] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA*, 2009, pp. 14–23.
[12] 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 *DAC*, 2012, pp. 453–458.
[13] 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 *MICRO*, 2009, pp. 14–23.
[14] M. Zhao, L. Jiang, Y. Zhang, and C. J. Xue, "SLC-enabled wear leveling for MLC PCM considering process variation," in *DAC*, 2014, pp. 1–6.
[15] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *MICRO*, 2009, pp. 347–357.
[16] 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 *ISCAS*, 2007, pp. 3014–3017.
[17] B. Li, Y. Shan, M. Hu, Y. Wang, Y. Chen, and H. Yang, "Memristor-based approximated computation," in *ISLPED*, 2013, pp. 242–247.
[18] B. Li, Y. Wang, Y. Wang, Y. Chen, and H. Yang, "Training itself: Mixed-signal training acceleration for memristor-based neural network," in *ASP-DAC*, 2014, pp. 361–366.
[19] S. Eilert, M. Leinwander, and G. Crisenza, "Phase change memory: A new memory enables new memory usage models," in *IMW*, 2009, pp. 1–2.
[20] Z. Shao, Y. Liu, Y. Chen, and T. Li, "Utilizing PCM for energy optimization in embedded systems," in *ISVLSI*, 2012, pp. 398–403.
[21] D. Jung, J. soo Kim, S. yeong Park, J. uk Kang, and J. Lee, "FASS: A flash-aware swap system," in *IWSSPS*, 2005.
[22] M. Saxena and M. M. Swift, "FlashVM: Revisiting the virtual memory hierarchy," in *HotOS*, 2009, pp. 13–13.
[23] H.-L. Li, C.-L. Yang, and H.-W. Tseng, "Energy-aware flash memory management in virtual memory system," *IEEE VLSI*, vol. 16, no. 8, pp. 952–964, 2008.
[24] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM memory management made easy," in *USENIX NSDI*, 2011, pp. 211–224.
[25] 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 TECS*, vol. 12, no. 3, pp. 1–27, 2013.
[26] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA*, 2009, pp. 24–33.
[27] H. Aghaei Khouzani, Y. Xue, C. Yang, and A. Pandurangi, "Prolonging PCM lifetime through energy-efficient, segment-aware, and wear-resistant page allocation," in *ISLPED*, 2014, pp. 327–330.