Accelerating Application Start-up with Nonvolatile Memory in Android Systems

Application launch time in mobile systems can adversely affect user experience. Android employs several software techniques to reduce application launch time, but not much research has been done from a hardware perspective. In this article, the authors analyze memory usage patterns of Android applications, suggest several hardware optimization techniques, and demonstrate how using nonvolatile memory can accelerate start-up time.

Hyojong Kim
Georgia Institute of Technology

Hongyeol Lim
Sejong University

Dilan Manatunga

Hyesoon Kim
Georgia Institute of Technology

Gi-Ho Park
Sejong University

Reducing application launch time has received little attention because the possible benefit is relatively small compared to application running time. However, users tend to have immensely diverse phone usage patterns in mobile systems, and some users have a large number of relatively short sessions (less than 10 seconds).[1] For these users, a long application start-up time can hinder their experience significantly.

Android has employed several techniques to reduce application launch time and memory usage. For example, the Zygote process lets applications share memory space in an efficient manner by reducing the need for repeated loading of shared libraries.[2] Android also attempts to keep an application in memory, even after the application is terminated, to reduce start-up time for future use.[3] In addition, Google provides several software optimization guidelines.[4] However, not much study has been done by hardware designers on reducing application launch time to enhance overall user experience (see the "Related Work on Improving Application Launch Time" sidebar).

In this article, we analyze Android applications' memory access behaviors during application launch time, and we propose a hardware solution to improve application launch time using nonvolatile memory (NVM)—specifically, using phase-change memory (PCM). We envision that future mobile processors will have NVM in the memory system to reduce energy consumption and improve performance. (Note: We use the term *NVM* to indicate NVM used as a memory, and we use the term *Flash* to indicate the storage.) Our proposal reserves some portion of PCM to store frequently used applications and shared libraries used by the applications. Our simulation results show a reduction of application launch

0272-1732/15/$31.00 © 2015 IEEE Published by the IEEE Computer Society

Related Work on Improving Application Launch Time

Prefetching has been a widely used technique to improve application launch time. For Android-specific optimizations, Yan et al. proposed a fast application launching mechanism called FALCON for slow applications.[1] FALCON uses user contexts such as user locations and behaviors to predict application launches, and it prelaunches an application to reduce the application launch time. Yan et al. showed that user behavior is habitual, which makes their prediction mechanism work well.

Researchers have proposed using nonvolatile memory (NVM) to speed up application launch time or to resume running an application for several domains. For example, Dong et al. proposed to create a checkpoint in NVM and use the checkpointed image to restart an application to support a fault-tolerant system.[2] Joo et al. proposed an application prefetching method called FAST, which specifically targeted solid-state drives (SSDs) to improve application launch time. They maintained a set of maps for each application that stores the application launch time file accesses and use that information to generate prefetching requests. FAST attempts to overlap the computation time with the SSD access time during each application launch time.[3] Our mechanism stores file images to NVM as well, but we also store common libraries and data to increase the coverage of applications while using a small storage area. Furthermore, our work is the first work to be demonstrated for Android applications.

References

1. T. Yan et al., "Fast App Launching for Mobile Devices Using Predictive User Context," *Proc. 10th Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 12), 2012, pp. 113–126.
2. X. Dong et al., "Hybrid Checkpointing Using Emerging Nonvolatile Memories for Future Exascale Systems," *ACM Trans. Architecture and Code Optimization* (TACO), vol. 8, no. 2, 2011, doi:10.1145/1970386.1970387.
3. Y. Joo et al., "FAST: Quick Application Launch on Solid-State Drives," *Proc. 9th USENIX Conf. File and Storage Technologies* (FAST 11), 2011, pp. 259–272.

time for 14 Android applications on several mobile platforms.

Background

We briefly summarize key optimizations applied to the Android application launch process and memory management system. Please refer to Android's developer documents for further details.[3]

Android's Zygote system

One of the key optimizations to reduce application launch time and memory consumption in Android systems is the introduction of the Zygote process. Zygote is a daemon process whose main job is to launch applications. This means that Zygote is the parent of all application processes. When an application is launched, Zygote forks a new process by cloning the entire virtual machine. However, because Android uses the copy-on-write policy, no memory data is actually copied until there is a write. So, libraries or data can be easily shared among applications.

Android's memory management systems

Similar to other embedded systems, Android does not offer swap space for memory. All memory addresses used by applications stay in memory until the process is completely terminated. Only garbage collectors can deallocate memory space.

When a user leaves an application, Android does not kill the application to allow fast resuming. Instead, it keeps the application in memory, so that if the user switches back to the application, the system can return to the prior state quickly. Only under low-memory conditions will Android evict previously used applications, starting with the least recently used (LRU) one.

Backgrounds on NVM

Emerging NVM technology has received much attention because its energy efficiency makes it an attractive solution for future computing systems, especially for low-power, high-performance mobile systems. Table 1 summarizes the performance and power characteristics of various memory technologies.[5]

PCM, one of the most promising emerging memory technologies, has fast read operations and low power consumption. However, as Table 1 shows, PCM suffers from exceptionally slow and power-hungry write operations and limited write endurance.

Table 1. Comparison of memory technologies.

Performance and power characteristics	Static RAM (SRAM)	DRAM	NAND Flash	Phase-change memory (PCM)
Read time	10 ns	10 ns	5 to 50 μs	10 to 100 ns
Write time	10 ns	10 ns	2 to 3 ms	100 to 1,000 ns
Standby power	Cell leakage	Refresh power	Zero	Zero
Read power	Low	Low	High	Low
Write power	Low	Low	High	High
Endurance	10^{18}	10^{15}	10^{5}	10^{8} to 10^{12}
Nonvolatility	No	No	Yes	Yes

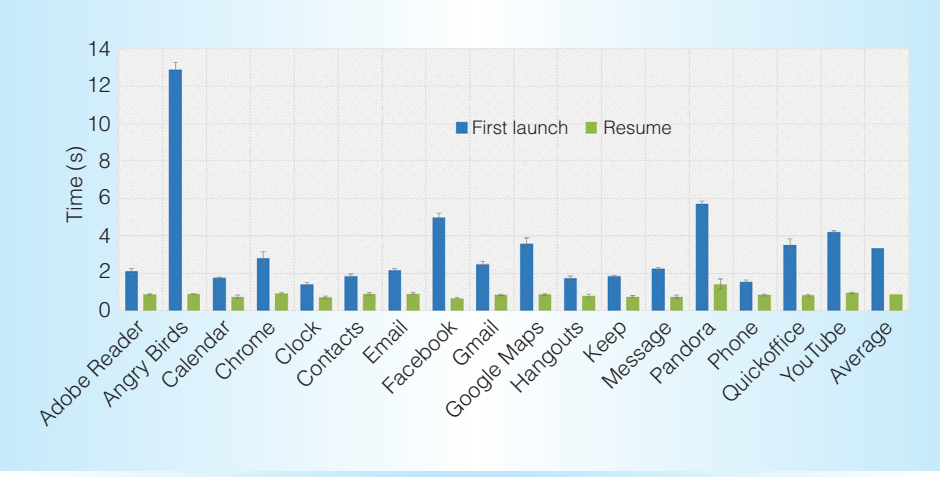


Figure 1. Application launch and resume time measurements of 14 widely used applications on a first-generation Nexus 7. It shows the performance overhead of first-time application launching.

Hence, a DRAM-PCM hybrid memory system attempts to exploit the advantage of these two memory technologies, and we propose to use PCM to improve application launch time.

Motivation

Figure 1 shows the application launch time of 14 widely used applications on a first-generation Nexus 7. We define an application launch time as the time it takes for an application to reach a ready state where user input is accepted. The results show two types of launch time: `first_launch` and `resume`.

`First_launch` measures the time it takes for an application to be launched for the first time after the device is turned on. The result shows that first-time launching takes 2.76 seconds on average and as much as 12.87 seconds for the case of Angry Birds.

`Resume` shows the time to resume an application after having switched to another application. Because Android keeps previously used applications in memory to minimize performance degradation caused by frequent switching between applications, `resume` allows restarting an application from the previous state, which often only requires the time to redraw a screen from memory. This explains the very short resume time.

Memory behaviors on devices

Here, we present memory usage behavior of Android applications.

Table 2. Evaluated devices and their memory and storage performance results.

Device	CPU	GPU	Memory	Flash
Google Nexus 7 (2012) 4.4.3	Nvidia Tegra 3 T30L (ARM Cortex-A9) Quad-core 1.3 GHz	Nvidia GF 12-core 416 MHz	L1 cache: 32 Kbytes, 3.34 ns, 32-byte line size L2 cache: 1 Mbyte, 25.63 ns, 64-byte line size DRAM: DDR3L-1333 1 Gbyte, 10.6 Gbytes per second (GBps), 148.70 ns	16 Gbytes R: 33.78 Mbytes per second (MBps) W: 12.60 MBps
Motorola RAZR i (2012) 4.1.2	Intel Z2460 Atom Single-core 2 GHz	PowerVR SGX 540 400 MHz	L1 cache: 24 Kbytes, 1.49 ns, 128-byte line size L2 cache: 512 Kbytes, 8.02 ns, 128-byte line size DRAM: LPDDR2-800 1 Gbyte, 12.8 GBps, 98.97 ns	8 Gbytes R: 37.25 MBps W: 12.08 MBps
LG Optimus G (2012) 4.1.2	Qualcomm Krait Quad-core 1.5 GHz	Qualcomm Adreno 320 400 MHz	L1 cache: 16 Kbytes, 2.93 ns, 64-byte line size L2 cache: 2 Mbytes, 39.13 ns, 128-byte line size DRAM: LPDDR2-1066 2 Gbytes, 17.0 GBps, 182.08 ns	32 Gbytes R: 43.09 MBps W: 5.98 MBps
Google Nexus 5 (2013) 4.4.3	Qualcomm Snapdragon 800 2.26 GHz	Qualcomm Adreno 330 450 MHz	L1 cache: 16 Kbytes, 1.33 ns, 64-byte line size L2 cache: 2 Mbytes, 16.02 ns, 128-byte line size DRAM: LPDDR3-1600 2 Gbytes, 25.6 GBps, 105.57 ns	16 Gbytes R: 86.57 MBps W: 46.23 MBps

Memory and storage performance

We measure memory system performance with LMBench,[6] and storage performance with AndroBench,[7] on four Android devices from different vendors. Table 2 shows the devices' specification and the measured performance of their memory systems and internal storage. The results show that L2 cache size is becoming larger (up to 2 Mbytes) and that Flash's bandwidth has significantly increased.

Memory usage growth measurements

We measure each application's memory usage at 5-second intervals starting at launch. Applications fall into two categories:

- *Stable.* Memory usage increases with time for the first 10 seconds and then stabilizes. Alarm, Multimedia Messaging Service (MMS), and Dialer fall into this category.
- *Unstable.* Memory usage increases with time and never stabilizes. Facebook and Chrome fall into this category.

One of the most prominent examples of the unstable category is Chrome, which ends up consuming about 250 Mbytes of memory if the user continues opening new tabs without closing previously used ones. We project that in such cases, Chrome will keep increasing its memory usage until Android can no longer tolerate it (Chrome's memory usage linearly increases as the number of open tabs increases). Hence, even though the average memory usage can be low, depending on a user's usage patterns, unstable applications can use up all the memory space. In such a scenario, all other applications residing in memory would have a higher chance of being evicted from the memory. As a result, an evicted application must be brought back from the storage, which will take much more time, and users will experience launch-time performance degradation.

Hardware optimizations

Now, we describe our solution to reduce the first-time application launching time overhead.

Key observations

From our previous experiments on devices, we make three observations. First, application launch time is typically more than 2 seconds, and for some applications it can be

higher. Even with Android's application caching mechanism, some applications' launch times are still not reduced (see Figure 1).

Second, although most applications do not use the entire memory space (except for a few exceptions), certain applications can use up almost all the memory space over time. Examples include Chrome and Facebook (see Figure 2).

Finally, even with Flash storage, I/O access time is an order of 500× to 1,000× slower, and I/O access time is a bottleneck for some applications' launch times (see Tables 2 and 3).

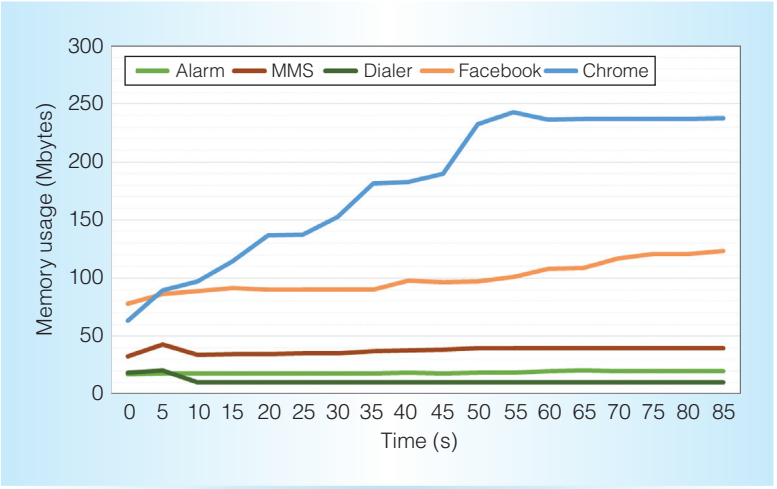


Figure 2. Memory usage over time for each application. (For visual clarity, we plot only selected applications from each group.)

Possible optimizations

With these observations, we suggest two hardware optimizations. First, most of the time, users do not use the entire DRAM memory space, so dynamically adjusting the DRAM memory size at runtime can reduce power consumption of DRAM. Second, instead of LRU, other memory management policies can improve user experience. Potential policies include keeping start-up-time-unfriendly applications inside the memory and limiting the amount of memory usage for applications.

Launch time optimization with NVM as memory

We evaluate hardware optimization techniques that employ NVM in the memory system. We envision that NVM will be used as a secondary memory between DRAM and Flash, as Figures 3b and 3c show. (NVM is a secondary memory to DRAM in the sense that in the event of data eviction from DRAM, the data is migrated to NVM according to our mechanism.) To enhance application launch time, we propose using dedicated regions (DRs) of the NVM to store frequently used applications and shared libraries accessed by the applications. Figures 3a through 3c illustrate an overview of the proposed memory system, the current mobile system, and a future memory system with NVM, respectively.

Figures 3d and 3e show the memory management mechanism with NVM. Let us first look at the insertion policy shown in Figure 3d. When the application source code is fetched from Flash, it will be stored in NVM instead of DRAM (Step 1), and in turn, directly sent to the L2 cache, bypassing DRAM (Steps 2 through 4). (Our experimental results show that for all 14 applications, performance degradation due to relatively slow PCM read latency is only about 0.28 percent on average and 0.54 percent at worst.)

On the one hand, when there is a data miss, data will be brought to DRAM instead of NVM (Step 1), and sent through the cache hierarchy to the core (Steps 2 through 4). In other words, DRAM and PCM sit at the same level with respect to the insertion policy, one serving instructions and the other serving data.

On the other hand, when there is a data eviction (Figure 3e), NVM will serve as a secondary memory to DRAM because the data is migrated onto NVM from DRAM (Step 3). However, when there is an instruction eviction from the last-level cache, L2 in this example, it will be simply invalidated without any write-back requests to the lower memory hierarchy (Step 2). Note that caches use the noninclusive policy, and both NVM and DRAM are part of the memory space.

Figure 4a illustrates a scenario in which our proposed mechanism will provide performance benefits. In the baseline, when an application needs more memory space, it will have another application migrated to NVM. Because Android moves data at application granularity, in this case, the victim application's entire data must be migrated to NVM. Imagine a situation in which Angry Birds,

Table 3. Evaluated benchmarks.

Benchmark	No. of total instructions	No. of memory instructions	Code		Data		I/O ratio (%)
			Size (Mbytes)	Coverage of DR (%)	Size (Mbytes)	Coverage of DR (%)	
Adobe Reader	1,626,551,883	46,494,602	9.15	68.28	47.91	89.25	48.63
Calendar	1,499,571,513	41,335,198	5.95	74.94	40.48	90.09	50.74
Clock	1,499,571,513	42,957,298	5.94	98.81	42.88	90.74	50.51
Contacts	1,625,554,921	45,314,164	6.00	96.03	50.46	84.40	54.28
Email	3,620,367,840	152,296,168	7.27	99.90	45.79	49.98	22.04
Facebook	3,104,993,640	83,004,592	6.82	98.06	82.60	84.13	52.03
Gmail	3,500,539,832	180,120,298	7.45	99.81	50.00	47.31	19.74
Hangouts	3,757,351,489	208,305,796	7.50	99.63	53.74	35.72	17.49
Keep	2,386,758,057	105,754,984	7.23	98.90	46.67	48.14	27.75
Messages	1,333,961,369	36,493,102	5.83	69.37	40.23	93.04	52.20
Pandora	902,906,369	226,933,466	9.90	98.96	52.36	28.12	16.81
Phone	1,516,985,275	42,521,026	5.87	95.89	42.93	91.18	48.55
Quickoffice	1,375,601,493	37,968,620	5.35	73.52	41.13	90.30	50.14
YouTube	1,509,964,322	1,137,890,524	7.39	24.03	54.30	29.66	3.91
Average	2,095,167,234	170,527,846	6.97	85.44	49.39	68.00	30.56

*"No. of total instructions" and "No. of memory instructions" are the aggregation from all threads. "Coverage of DR" is the proportion of code or data that is covered from each application's dedicated region. We use the `shared_code_data_in_pcm` scenario for this analysis. When more than one application shares data or code, the corresponding memory region can be stored in the dedicated region (DR). "I/O ratio" is the relative time spent in I/O accesses to entire execution time.

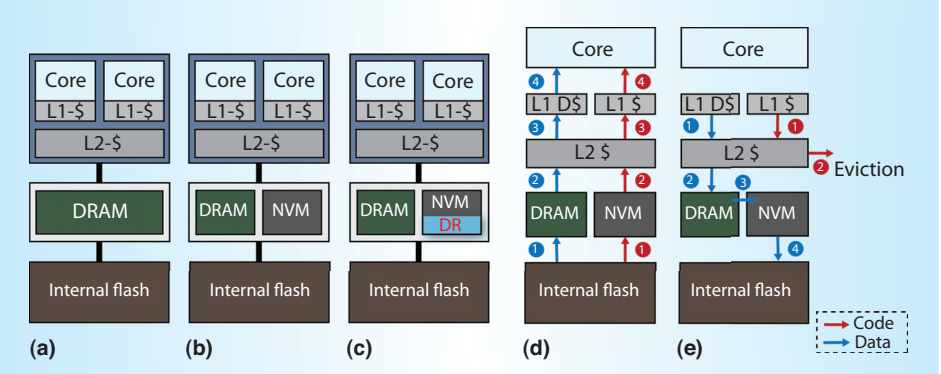


Figure 3. Memory systems with NVM: (a) the current mobile system, (b) the mobile system with NVM, (c) the proposed mechanism, (d) insertion policy, and (e) eviction policy. The figure shows current and envisioned memory systems with NVM (a through c) and the memory management mechanisms (d through e).

Gmail, and Google Maps are used, and they reside in memory. When Chrome is launched, Angry Birds will be kicked out of DRAM to NVM, according to the LRU policy. Over time, Chrome will require more memory space, as Figure 2 shows, and more applications will end up being migrated to NVM. Eventually, Chrome could become a memory hog, even using the NVM memory, because NVM is a part of the memory space.

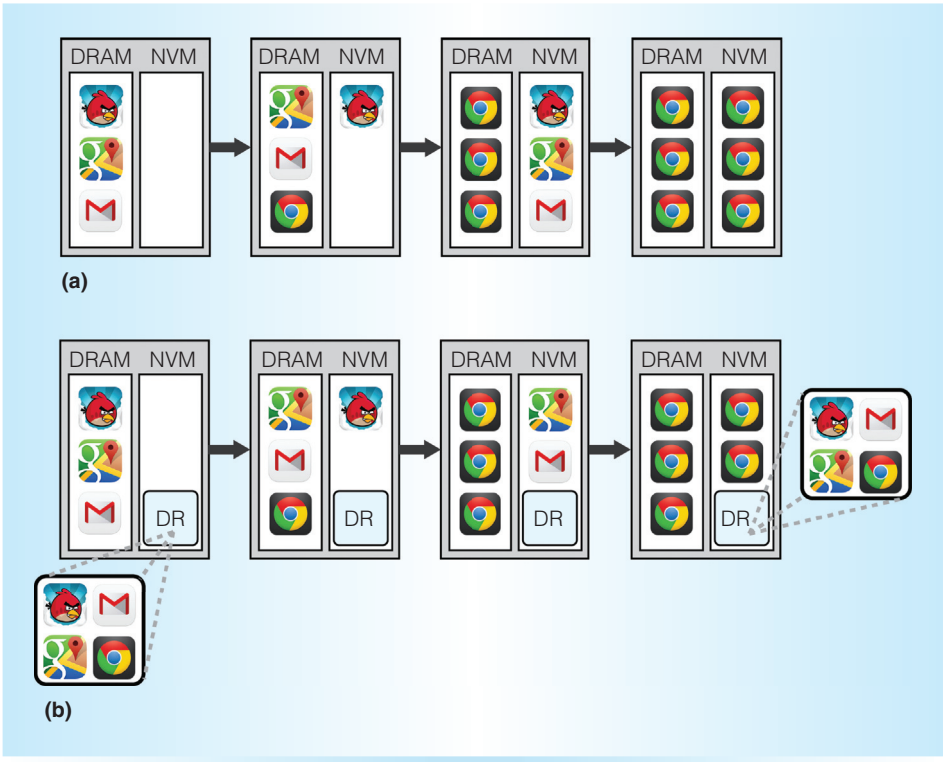


Figure 4. Replacement scenario with a high-memory-usage application running: (a) baseline vs. (b) start-up-friendly management. In this example, Chrome becomes a memory hog.

As a result, when the user switches to another application—for example, Angry Birds—the application must be brought back from Flash, which is very slow.

In our mechanism, because some portion of NVM is always reserved for start-up data, Chrome cannot evict all other previously running applications from NVM (see Figure 4b). This will directly translate into performance benefits or reduction in application launch time. From Chrome's perspective, the available memory space is less than the baseline memory system. However, because the dedicated regions are only 128.73 Mbytes (6.39 percent of NVM, assuming the size of entire memory (DRAM and NVM) is 2 Gbytes), the loss of memory space should be negligent. Please note that this calculation is based on our evaluation, which we will discuss later.

Experimental results

We evaluate the proposed mechanism's benefit on 14 Android applications because of our observation that Android applications share lots of code and data between them.

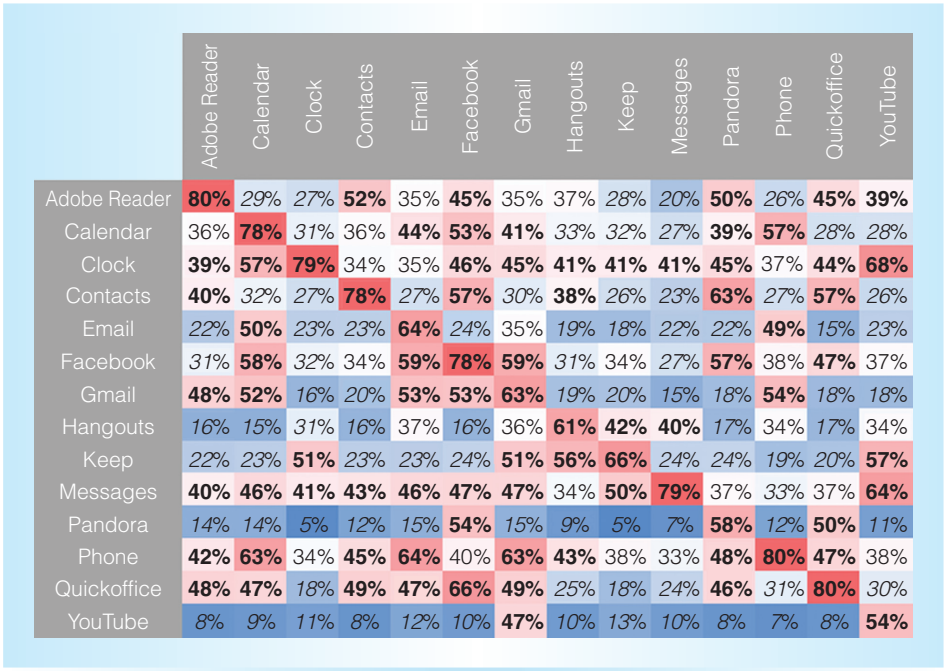
Evaluation methodology

We collected traces using Pin for Android (Pindroid).[8] We ran benchmarks on x86 platforms from the beginning of the application until they reached the ready state. Our benchmark suite (see Table 3) comprises 14 applications, each of which was chosen to represent a variety. (Owing to the limitations of using an x86 device, we replaced some applications from Figure 1 with other applications to generate traces.)

We modeled Google Nexus 7 memory timing values in Table 2 to evaluate our proposed mechanism. We chose PCM as a candidate for NVM, and the read latency of PCM is modeled to be a twofold of DRAM access latency. PCM write latency is modeled as a tenfold of read latency. The minimum unit of access to Flash is 4 Kbytes.

Sharing patterns of application launch time code and data

Android applications share a lot of code and data among applications. (Here, "data" indicates all the memory content used by



	Adobe Reader	Calendar	Clock	Contacts	Email	Facebook	Gmail	Hangouts	Keep	Messages	Pandora	Phone	Quickoffice	YouTube
Adobe Reader	**80%**	*29%*	*27%*	**52%**	35%	**45%**	35%	37%	*28%*	*20%*	**50%**	*26%*	**45%**	**39%**
Calendar	36%	**78%**	*31%*	36%	**44%**	**53%**	**41%**	*33%*	*32%*	*27%*	**39%**	**57%**	*28%*	*28%*
Clock	**39%**	**57%**	**79%**	34%	35%	**46%**	**45%**	**41%**	**41%**	**41%**	**45%**	37%	**44%**	**68%**
Contacts	**40%**	*32%*	*27%*	**78%**	*27%*	**57%**	*30%*	**38%**	*26%*	*23%*	**63%**	*27%*	**57%**	*26%*
Email	*22%*	**50%**	*23%*	*23%*	**64%**	*24%*	35%	*19%*	*18%*	*22%*	*22%*	**49%**	*15%*	*23%*
Facebook	*31%*	**58%**	*32%*	34%	**59%**	**78%**	**59%**	*31%*	34%	*27%*	**57%**	38%	**47%**	37%
Gmail	**48%**	**52%**	*16%*	*20%*	**53%**	**53%**	**63%**	*19%*	*20%*	*15%*	*18%*	**54%**	*18%*	*18%*
Hangouts	*16%*	*15%*	*31%*	*16%*	37%	*16%*	36%	**61%**	**42%**	**40%**	*17%*	34%	*17%*	34%
Keep	*22%*	*23%*	**51%**	*23%*	*23%*	*24%*	**51%**	**56%**	**66%**	*24%*	*24%*	*19%*	*20%*	**57%**
Messages	**40%**	**46%**	**41%**	**43%**	**46%**	**47%**	**47%**	34%	**50%**	**79%**	37%	*33%*	37%	**64%**
Pandora	*14%*	*14%*	*5%*	*12%*	*15%*	**54%**	*15%*	*9%*	*5%*	*7%*	**58%**	*12%*	**50%**	*11%*
Phone	**42%**	**63%**	34%	**45%**	**64%**	40%	**63%**	**43%**	38%	33%	**48%**	**80%**	**47%**	38%
Quickoffice	**48%**	**47%**	*18%*	**49%**	**47%**	**66%**	**49%**	*25%*	*18%*	*24%*	**46%**	*31%*	**80%**	*30%*
YouTube	*8%*	*9%*	*11%*	*8%*	*12%*	*10%*	**47%**	*10%*	*13%*	*10%*	*8%*	*7%*	*8%*	**54%**

Figure 5. A visualization of correlations across all 14 applications. Each entry at coordinates (*X*, *Y*) represents the data portion of application *X* that is shared with application *Y*. Bolded numbers indicate high correlation, and italicized numbers indicate low correlation.

memory instructions.) This is mainly because many applications are built on top of the Android-provided framework. Hence, Zygote's copy-on-write policy makes it more attractive. Before we discuss what we should store in the DR, we first examine how much code and data are shared by each application.

The heat map in Figure 5 visualizes application sharing. Each entry at coordinates (*X*, *Y*) represents the data portion of application *X* that is shared with application *Y*. Bolded numbers indicate high correlation (that is, lots of sharing), whereas italicized numbers mean low correlation. We consider only memory accesses to "actual" data, which does not include accesses to the stack frame or network buffer. Thus, the ratio is the number of actual memory accesses divided by the number of total memory accesses. This explains why we do not see 100 percent in the table.

Not surprisingly, Android "stock" applications, such as Calendar, Clock, and Messages, have relatively higher correlation with others compared with other third-party applications such as Pandora or YouTube, as indicated by the dark blue entries that appear in the last and fourth last rows.

Table 3 summarizes the amount of data that each application uses during launch time and the shared portion of that data. Here, "data" refers to all the memory regions that are brought by applications as opposed to code. On average, 56.34 Mbytes of memory is required to launch an application, about 70.19 percent of which is shared with other applications. By summing up each application's unique data size and commonly shared data size across all applications, we can estimate the amount of space required to store the data that all 14 applications use during their launch time, which is about 128.73 Mbytes. As the number of applications increases, we expect that the additional memory space to store the corresponding data will be marginal, because much of the code and data an application accesses during launch time is shared.

Impact of using DRs

Figure 6 shows application launch time variations for various scenarios. In this context, "shared" indicates memory contents that are used by more than one application. The required data size is provided in parentheses.

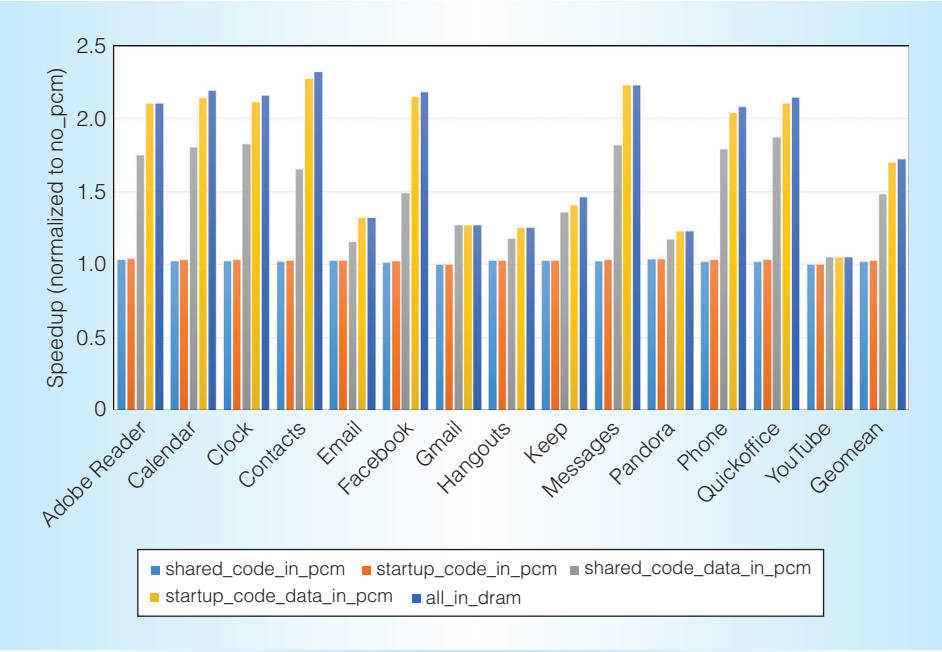


Figure 6. Application launch time speedup for various scenarios. Each bar represents the speedup normalized to the `No_pcm` case.

- In `No_pcm`, all the code and data come from Flash.
- In `Shared_code_in_pcm`, only shared code is supplied from PCM (5.59 Mbytes).
- In `Startup_code_in_pcm`, all startup code is supplied from PCM (24.93 Mbytes).
- In `Shared_code_data_in_pcm`, shared code and data are supplied from PCM (43.29 Mbytes).
- In `Start_up_code_data_in_pcm`, both instructions and data are supplied from PCM (128.73 Mbytes).
- In `All_in_dram`, the ideal scenario, all the code and data come from DRAM.

Each bar in Figure 6 represents the speedup normalized to the `No_pcm` case.

Overall, `Startup_code_data_in_pcm` achieves about a 69.98 percent performance improvement on average, and as much as a 227 percent performance improvement in the Contacts application. As the two rightmost bars in the figure indicate, this enhancement is within 2 percent of an ideal system that has all data in DRAM memory. As Table 3 implies, storing only code in PCM does not provide a good performance improvement.

Combined with I/O characteristics of each application, an application with more I/O accesses tends to get a better performance benefit from our proposed solution. For instance, Contacts shows as much as 2.27× speedup with `Startup_code_data_in_pcm`. That is because about 53.19 percent of I/O accesses is reduced and this directly translates into performance improvement. However, YouTube shows only about 1.05× speedup with `Startup_code_data_in_pcm`. That is because YouTube is computing intensive, as far as application launch time is concerned, and consumes only about 3.91 percent of time-transferring data. As compared to other, non-computing-intensive applications, performance improvement might not be considered significant, but with our mechanism we could reduce most of the I/O time, from 3.91 to 0.03 percent.

Storing all code and data in PCM requires only 128.73 Mbytes, which is only 5.36 percent of the memory. We also evaluated reducing the size of 128.73 Mbytes further by storing code and data on the basis of the number of shared applications. When we reduce the size by half (52.31 Mbytes), the performance benefit is still 44.31 percent on average.

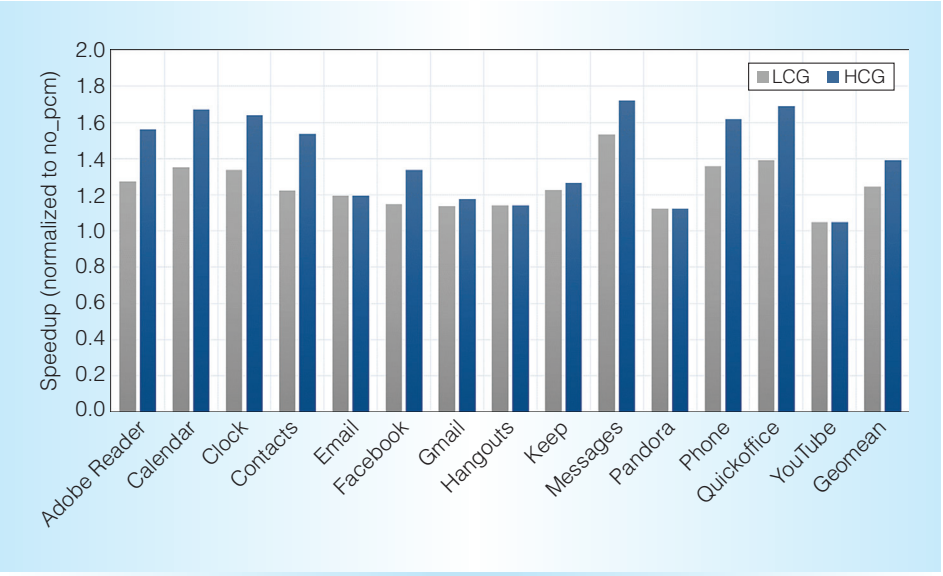


Figure 7. Application launch time after running other applications. HCG shows when highly correlated group applications are launched back to back, and LCG shows when lowly correlated group applications are launched back to back.

Launch time behavior after running other applications

As we explained earlier, Android keeps previously used applications in memory. For this reason, the applications that the user ran in the past can influence the application about to be launched, as long as those applications are still kept in memory. For example, if a user ran applications that are highly correlated with the one that is about to be launched (that is, both applications share a lot of code and data), it would take less time than the opposite case in which the user ran lowly correlated applications. Figure 7 illustrates this situation.

For this experiment, we carefully selected a set of applications (in this case, three) that are highly or lowly correlated with the given application. We call these sets of applications the highly correlated group (HCG) and the lowly correlated group (LCG), respectively. An experiment with the HCG is an attempt to mimic the scenario in which the user "luckily" used highly correlated applications with the one that is about to be launched, whereas an experiment with the LCG mimics the scenario in which the user "unfortunately" used the applications that have little to do with the given application. Even though the HCG improves performance significantly (38.87 percent on average), it is still lower than our proposed mechanism (69.98 percent on average), storing all the shared libraries. Some applications, such as Adobe Reader, Clock, and Phone, show a huge performance difference between experiments with the HCG and the LCG. Other applications, such as Email, Hangouts, Pandora, and YouTube, barely show a performance difference between those two experiments. However, even in the latter case, except for YouTube, our proposed mechanism provides 30 percent more performance than what the LCG can provide. We extrapolate that the degree of influence of previously used applications would be somewhere in the middle between the LCG and HCG cases.

Dedicating some portions of NVM shows a great performance improvement of application start-up time. However, it reduces the memory space, which could hurt the memory-intensive applications' performance, even though the proposed DR uses less than 6 percent of the total memory space. In such a scenario, it is also possible to write back the content of DRs to a storage system and fetch them back as soon as the memory-intensive application is over. In the future, we will collect user-based Android

memory-usage patterns and evaluate the performance degradations of various memory-usage scenarios and develop such kinds of dynamic policies. As the number of applications increases or as applications are updated, the size of start-up code and data can also increase. Hence, we need a more sophisticated policy to insert and replace the contents in the DR. With our Android user application usage study, we will study DR management policies. MICRO

Acknowledgments

We gratefully acknowledge the support of the National Research Foundation of Korea (NRF) grant funded by the Korean government (NRF-2011-220-D00098) and the National Science Foundation (NSF) CAREER CCF 1054830.

References

1. H. Falaki et al., "Diversity in Smartphone Usage," *Proc. 8th Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 10), 2010, doi:10.1145/1814433.1814453.
2. J.S. Rivaya, "Zygote," Oct. 2013; http://anatomyofandroid.com/2013/10/15/zygote.
3. Android Developers, "Managing Your App's Memory," Nov. 2014; http://developer.android.com/training/articles/memory.html.
4. Android Developers, "Introduction to Android," Nov. 2014; http://developer.android.com/guide/index.html.
5. X. Dong et al., "Hybrid Checkpointing Using Emerging Nonvolatile Memories for Future Exascale Systems," *ACM Trans. Architecture and Code Optimization* (TACO), vol. 8, no. 2, 2011, doi:10.1145/1970386.1970387.
6. L. McVoy and C. Staelin, "LMBench: Portable Tools for Performance Analysis," *Proc. Conf. USENIX Ann. Technical Conf.*, 1996, pp. 279–294.
7. J.-M. Kim and J.-S. Kim, *AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices*, Springer Berlin Heidelberg, 2012.
8. N. Mor-Sarid and M. Nir-Gross, "Pin—A Dynamic Binary Instrumentation Tool," June 2012; https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

Hyojong Kim is a PhD student in the School of Computer Science at the Georgia Institute of Technology. His research focuses on memory systems for near-data processing. Kim has a BS in electrical and computer engineering from Seoul National University. Contact him at hyojong.kim@gatech.edu.

Hongyeol Lim is a PhD student in the Department of Computer Science and Engineering at Sejong University. His research interests include high-performance computer architecture and memory hierarchy design with 3D-stacked memory in embedded systems. Lim has a BS in computer engineering from Sejong University. He is a member of IEEE and the ACM. Contact him at hylim@sju.ac.kr.

Dilan Manatunga is a PhD student in the School of Computer Science at the Georgia Institute of Technology. His research focuses on optimizations for mobile operating systems and architectures. Manatunga has a BS in computer science from the Georgia Institute of Technology. Contact him at dmanatunga@gatech.edu.

Hyesoon Kim is an associate professor in the School of Computer Science at the Georgia Institute of Technology. Her research interests include high-performance, energy-efficient heterogeneous architectures; interaction between programmers, compilers, and microarchitectures; and developing tools to help parallel programming. Kim has a PhD in computer engineering from the University of Texas at Austin. She is a member of IEEE and the ACM. Contact her at hyesoon@cc.gatech.edu.

Gi-Ho Park is an associate professor in the Department of Computer Science and Engineering at Sejong University. His research interests include advanced computer architectures, memory system design, system-on-chip (SoC) design, and embedded system design. Park has a PhD in computer science from Yonsei University. He is a member of IEEE and a professional member of the ACM. Contact him at ghpark@sejong.ac.kr.