Mismatched Memory Management of Android Smartphones

Yu Liang
Department of Computer Science,
City University of Hong Kong

Qiao Li
Department of Computer Science,
City University of Hong Kong

Chun Jason Xue
Department of Computer Science,
City University of Hong Kong

Abstract

Current Linux memory management algorithms have been applied for many years. Android inherits Linux kernel, and thus the memory management algorithms of Linux are transplanted to Android smartphones. To evaluate the efficiency of the memory management algorithms of Android, page re-fault is applied as the target metric in this paper. Through carefully designed experiments, this paper shows that current memory management algorithms are not working well on Android smartphones. For example, page re-fault is up to 37% when running a set of popular apps, which means a large proportion of pages evicted by the existing memory management algorithms are accessed again in the near future. Furthermore, the causes of the high page re-fault ratio are analyzed. Based on the analysis, a tradeoff between the reclaim size and the overall performance is uncovered. By exploiting this tradeoff, a preliminary idea is proposed to improve the performance of Android smartphones.

1 Introduction

With many optimizations [2, 3, 5, 7], existing Linux memory management algorithms have been applied for many years. Android smartphones have seen remarkable growth in recent years. Android inherits Linux kernel. Thus the memory management algorithms of Linux are transplanted to Android smartphones.

To evaluate the efficiency of memory management algorithms on Android smartphones, page re-fault ratio is applied. Page re-fault represents the case that a page fault happens on a previously evicted page. Page re-fault ratio represents the proportion of re-faulted pages on all the evicted pages. Through carefully designed experiments, this paper shows that current memory management algorithms do not match the characteristics of apps running on Android smartphones. They induce high page re-fault, up to 37% when running popular apps. In the I/O stack of Android smartphones, page fault is the root cause of long read latency. As page re-fault is one type of page faults, high page re-fault ratio will significantly degrade the system performance.

Prior research focused on reducing the number of page faults by optimizing eviction algorithms [13, 15]. Eviction algorithms decide how to select the victim pages which will be evicted out of memory. The optimized LRU is known as a good eviction algorithm and is applied in Android. However, the experimental results show that page re-fault ratio is surprisingly high on Android smartphones.

This paper shows that the main cause of high page re-fault ratio is that the reclaiming scheme of memory mismatches the characteristics of apps running on Android smartphones. The mismatches include two aspects: 1. The reclaim size [1] is often too large for the requests on smartphones; 2. The limited reclaim scope [2] aggravates the punishment. To solve this problem, this paper proposes to exploit the tradeoff between the reclaim size and the overall performance of smartphones.

This paper reveals several interesting observations:

- Page re-fault is unexpectedly high (up to 37%) on Android smartphones when running popular apps;
- Even launching one app after restarting smartphones could induce page re-fault;
- The maximum allocation size of buddy system [3] is often too large for the requests of apps running on Android smartphones.

In the following sections, a brief introduction of Android I/O stack and the motivation of this work are presented in Section 2. Section 3 presents the observations of page re-fault on Android smartphones and their causes are analyzed in Section 4. In Section 5, a preliminary idea is proposed to exploit the tradeoff between the reclaim size and the performance. Section 6 introduces related works. This paper is concluded in Section 7.

[1] Reclaim size represents the number of pages freed by each reclaim operation.

[2] Reclaim scope represents the region of pages freed by each reclaim operation, such as the pages in the inactive_file_lru list.

[3] Buddy system is used to manage memory, and it divides memory blocks into partitions to service a memory request as fit as possible.

2 Background and Motivation

Android is a lightweight operating system maintained by Google, based on the Linux kernel and designed primarily for mobile devices. Figure 1 presents the architecture of Android I/O stack, mainly including userspace, Linux kernel, and device.

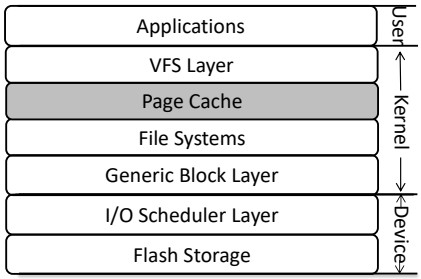

Figure 1: An overview of Android I/O stack.

Application read requests are serviced from the kernel page cache. If a requested page is not in the page cache, a page fault will be generated. File system layer will be accessed to find the logical address of the requested page. After that, a read request will go through generic block and I/O scheduler layers to fetch the requested page from flash storage as an I/O operation. Each layer contributes some factors, which could prolong the read latency, such as fragmentation in file system, I/O scheduler scheme, GC of flash storage. The latency of accessing all of these layers is on microsecond scale, while the latency of accessing memory is on nanosecond scale. Hence page fault is often the root cause of long read latency.

To quantitatively show the influence of page fault on Android smartphones, the latencies of launching Twitter and Facebook apps in three situations are measured on a real Android smartphone (Huawei P9 mounted with fourth Extended file system (Ext4) and Flash-Friendly File System (F2FS)), as shown in Figure 2.

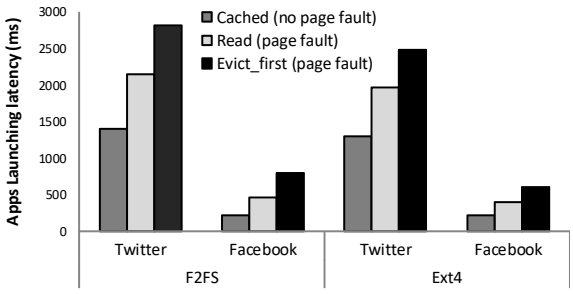

Figure 2: Influence of page fault on app launching latencies on Android smartphones.

In the "Cached" case of Figure 2, the requested data pages are in the page cache and can be directly accessed. This case is implemented by re-launching the app right after it is closed, and thus its data is still in memory [4]. In this case, only few page faults may happen. In the "Read" case, page fault may happen on all the requested pages, and the page cache has enough free space to launch the app immediately. This case is implemented by launching the app after cleaning the page cache. In the "Evict_first" case, page fault may happen on all the requested pages, and the page cache is full. To launch the app, some pages have to be evicted from the page cache to release space first. This case is implemented by launching the app after sequentially launching twenty other apps. The results show that the latency of launching an app is the shortest in the "Cached" case. Compared to "Cached" case, "Read" and "Evict_first" cases cause longer launching latency for the listed two apps. The additional latency is mainly caused by page faults.

Page fault happens in three cases: 1. Reading a page for the first time; 2. Reading an evicted page; 3. Reading a wrong address. When reading a page for the first time (case 1), page fault will happen because the physical memory is not allocated for this page. When reading a wrong address (case 3), the process will be killed. Compared to the other two cases, the case 2 could be avoided. Reading an evicted page is called page re-fault in this paper. It means the requested page had been in memory but was evicted by reclaiming scheme. Page re-fault ratio represents the proportion of re-faulted pages on all evicted pages. Compared to the traditional metric, page cache hit ratio, page re-fault ratio is more suitable for evaluating the efficiency of memory reclaiming scheme.

3 Page Re-fault on Android Smartphone

This section presents results and analysis of measuring page re-faults on Android smartphones when running popular apps.

3.1 Experimental Setup

All experiments are performed on a Huawei P9 smartphone with ARM's Cortex-A72 CPU, 32GB internal memory and 3GB RAM, running Android 7.0 on Linux 4.1.18 kernel. There is no external SD card and all the I/O happens on the internal eMMC flash storage (/data partition) of Android. We instrument kernel source code to collect information about memory allocation and reclaiming. The information includes the number of re-fault pages, the number of evicted pages, the size of each allocation, and the size of each reclaiming. The **adb** (Android Debug Bridge) tool [4] is used to obtain this information from the smartphone. To avoid bias, all experiments are conducted ten times and the average is shown.

Page re-fault ratio depends on the status of memory (empty or full) and workloads (light or heavy). Several popular Android applications, including Facebook, Twitter, Chrome, Google Earth, Google Map, Angrybird, Youtube (i.e. social apps, browser, map, game, and multimedia) are used in the experiments. Both app launching and execution are evaluated for different cases as shown in Table 1.

[4] Cache status is checked by the command **dumpsys meminfo**.

Table 1: Application combinations used in experiments.

Application	Memory	Workloads
Launching one app	empty	light
Using two apps	empty	light
Launching five apps and using two apps	full	moderate
Launching ten apps	full	moderate
Launching twenty apps and using three apps	full	heavy

Launching one app and using two apps each for one minute are evaluated for empty memory and light workloads. Launching five apps and using two apps each for one minute as well as launching ten apps are evaluated for full memory and moderate workloads. Finally, launching twenty apps and using three apps each for one minute are evaluated for full memory and heavy workloads. All these cases are designed based on our survey of Android smartphone users' usage behaviors.

3.2 Page Re-fault

Through carefully designed experiments, results show that page re-fault ratio on Android smartphones when running popular apps is unexpectedly high. Two aspects of page re-fault are evaluated: severity and reproducibility.

For severity: the ratio and the number of page re-fault when running popular apps are shown in Figure 3. These experiments are designed based on our survey of eighty Android smartphone users. According to the survey, more than 70% of users will not close apps after they finish using them. Thus, many apps stay in the background, but only a few of them are used frequently. In order to reproduce this scenario, twenty apps are launched and stay in the background and only three apps are used iteratively (each for one minute) in these experiments. Thirty five combinations of seven apps (Facebook, Earth, YouTube, Map, Twitter, Chrome) are evaluated in the experiments. In the experimental results, we use the acronyms as an abbreviation of application combinations. For example, FEY represents Facebook, Earth, and Youtube.

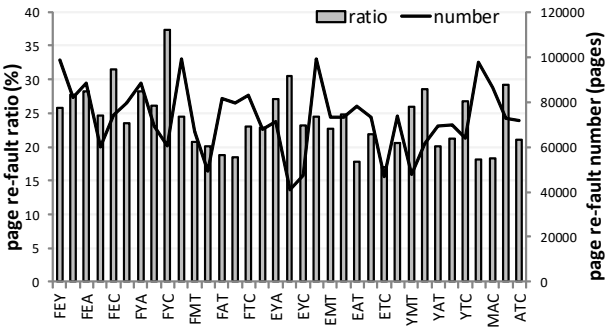

Figure 3: Ratio and number of page re-fault when using different application combinations for three minutes.

In Figure 3, the results show that page re-fault could be up to 37% when running popular apps. This means the existing Linux memory management algorithm often reclaims the pages, which will be used in the near future. The number of page re-faults could be up to 99358 in three minutes, which would degrade the performance according to the analysis in Section 2.

For reproducibility: the page re-fault ratio is also collected in other different cases and shown in Figure 4. Since page re-fault ratio depends on the memory status, the experiments are conducted in two cases: after restart and after cleaning cache. This is because some data could be pre-loaded into memory after restart, while the page cache will be empty after cleaning cache. The results show that after restart, page re-fault could happen even when only launching one app.

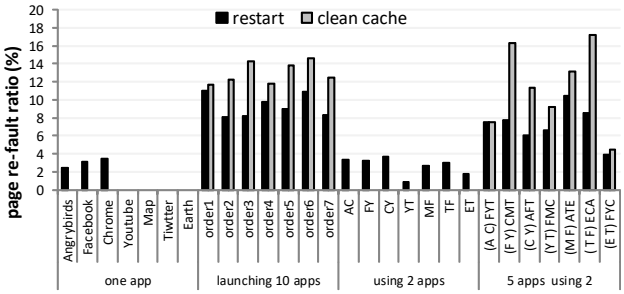

Figure 4: Page re-fault ratio of light workloads where there are some free spaces in memory. In "launching 10 apps" case, 10 apps are launched in different orders. In "5 apps using 2" case, (AC) FYT represents launching Angrybirds, Chrome, Facebook, Youtube, and Twitter, but only using Angrybirds and Chrome.

In summary, the above results show that page re-fault is prevalent when running popular apps on Android smartphones. The latency of a request with page fault could be 1000 times of the latency of a request with page cache hit. Thus, the performance could be degraded 10 times by 1% of page faults.

4 Page Re-fault Analysis

To find the causes of the high page re-fault ratio on Android smartphones, the allocation and reclaiming procedures are investigated. The investigation results show that the main cause is that the reclaiming scheme of buddy system mismatches the characteristics of apps running on Android smartphones. The mismatches include two aspects: 1. The reclaim size is often too large for the requests on Android smartphones; 2. The limited reclaim scope aggravates the punishment.

Cause 1: Compared to the allocation size, the reclaim size is often too large.

In buddy system, every memory block has an order, where the order is an integer ranging from 0 to 11. The size of a block of order n is 2^n. The distribution of allocation order when running popular apps is shown in Figure 5. These results are collected from the allocation function *_alloc_pages_nodemask()*.

The results show that on the Android smartphone, 99% of allocation orders are 0 (1 page), and more than 99.9% of orders are smaller than 4 (16 pages). This is because the requests on Android smartphones are mostly in small size. One of the main reasons is that most Android applications use SQLite as database. SQLite and its temporary files are mostly accessed in 4KB (1 page) units [9, 11].

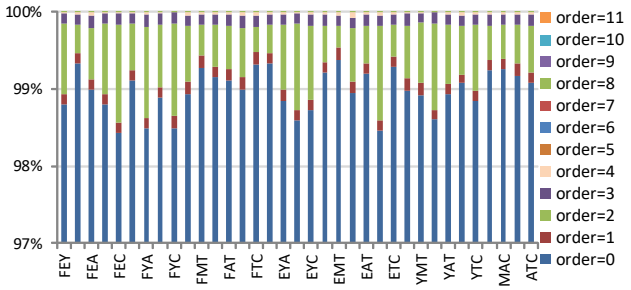

Figure 5: The distribution of allocation orders. The corresponding allocation size equals to 2^{order}.

The distribution of reclaim sizes is shown in Figure 6. The results show that in most of the cases, the reclaim size is much larger than the allocation size. 80% of reclaim sizes are larger than 32 pages (order=5), but 99% of allocation orders are 0 as shown in Figure 5.

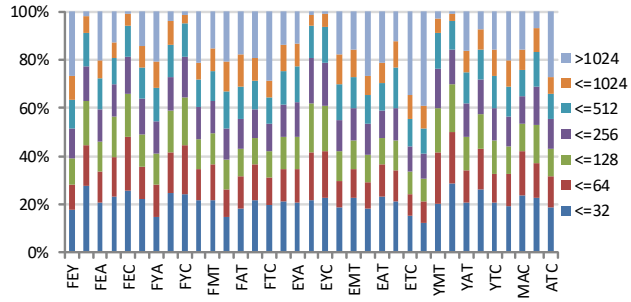

Figure 6: The distribution of reclaim sizes. These results include reclaimings from both LRU lists and slab, and they are collected in the functions *shrik_lruvec()* and *shrik_slab()*, respectively.

There are mainly three types of reclaimings of buddy system: fast reclaiming, direct reclaiming and ***kswapd***. ***kswapd*** is a thread used to reclaim free space when the number of free pages is lower than a threshold. The other reclaiming methods are called by the allocation procedure. Direct reclaiming is a heavy-weight reclaiming method, it will be triggered when the free space is not enough for the current allocation. During direct reclaiming, the allocation procedure needs to wait until enough free pages are reclaimed. Moreover, direct reclaiming could trigger flush operations. Compared to direct reclaiming, fast reclaiming is faster. It reclaims free pages by using *zone_claim*, and it does not reclaim mapped pages or trigger flush operations. The minimum reclaim size equals to twice the size of allocation size. The maximum reclaim size could be a few thousand of pages, and these large-size reclaimings are mostly produced by ***kswapd***. This design is used to avoid too many heavy-weight direct reclaimings. For servers, this design is suitable because the request size is usually large and the workloads are often heavy. However, for Android smartphones, the request size is often relatively small and the workloads are often light. Thus, a large-size reclaiming scheme could induce more page re-faults than necessary. An example of this case is shown in Figure 7.

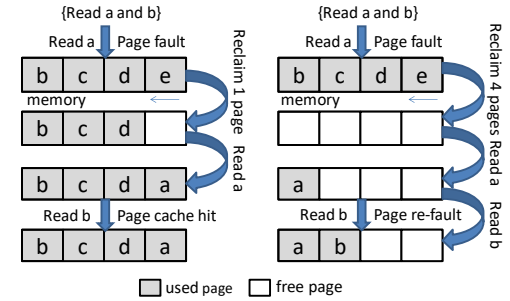

Figure 7: Large-size reclaiming induces more page re-faults.

In this example, there are 4 pages in the memory. Two requests need to be processed: read page **a** and read page **b**. When read page **a**, page fault happens. Since memory is full, some pages need to be evicted to reclaim free space. If the reclaim size equals 1 (left), only page **e** is evicted from memory. When read page **b**, page cache hit happens because page **b** is in memory. If reclaim size equals 4 (right), all four pages are evicted from memory. When read page **b**, page re-fault happens because page **b** has been evicted from memory. In summary, the large-size reclaiming scheme could induce more page re-faults.

Cause 2: Even if some apps have not been used for a long time, their pages are not evicted from memory. Instead, useful pages of foreground apps are often evicted.

All the pages are in one of five LRU lists: *Active_anonymous*, *inactive_anonymous*, *active_file*, *inactive_file*, and *unevictable*. The pages in the *unevictable* list will not be evicted. Since anonymous pages contain the heap information associated with a process, they are more important than file pages to the process. The pages in *active_anonymous* list will usually not be evicted, even if they belong to a background process. Thus, some anonymous pages of background processes could be stay in memory, while the file pages of foreground processes evicted. These file pages of foreground apps evicted may be accessed again in the near future, which leads to a high page re-fault ratio.

In summary, the large-size reclaiming on Android smartphones could induce a high page re-fault ratio, especially when the reclaiming scheme keeps *active_anonymous* pages of background apps in memory.

5 Preliminary Idea

Since *active_anonymous* pages are very important to the process and should not be reclaimed, a preliminary idea is to

reduce the reclaim size according to the characteristics of Android smartphones. From the analysis in Section 4, most large-size reclaimings come from ***kswapd***. A preliminary idea could be implemented by reducing the number and the reclaim size of ***kswapd*** by tuning its thresholds. There is a tradeoff between the reclaim size and the overall performance:

- If the reclaim size and the number of ***kswapd*** are too small, the free pages will be consumed quickly. Thus, when an allocation failed because of insufficient free pages, the heavy-weight direct reclaiming will be triggered and thus degrades the performance;
- On the other hand, if the reclaim size and the number of ***kswapd*** are too large, the ratio of page re-fault will be high and thus degrades the performance.

To show the tradeoff on Android smartphones, the statistics of page re-fault and direct reclaiming when ***kswapd*** is turned on or turned off is collected and presented in Table 2.

Table 2: Influence of ***kswapd*** on performance.

Metrics	With *kswapd*	Without *kswapd*	For performance
Reclaim size	650	30.6	-
Page re-fault	28.19%	20.06%	negative
Direct reclaiming	1.12%	40.39%	negative

These results show that when ***kswapd*** is turned off, the average reclaim size is greatly reduced. The page re-fault ratio is decreased, and thus the performance could be improved. However, the ratio of heavy-weight direct reclaiming is increased, and thus the performance could be degraded. There is a tradeoff between the reclaim size and the performance.

To further illustrate the tradeoff, the latency of launching seven apps are tracked when ***kswapd*** is turned on or turned off. The results are shown in Figure 8.

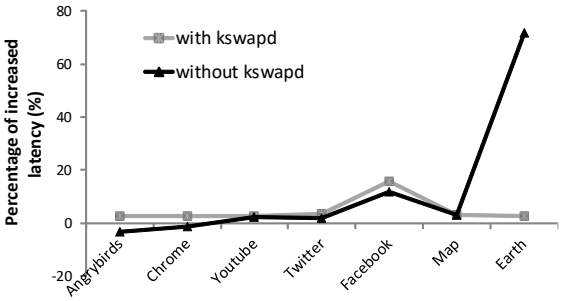

Figure 8: The percentage of increased latency of launching seven apps with and without ***kswapd***.

The results show that at the beginning, the latency when ***kswapd*** is turned on is similar to when it is turned off. This is because free pages are sufficient at the beginning. As the number of launching operations increases, the free pages in memory will be consumed. Thus, during this period, the performance when ***kswapd*** is turned off is better than that when ***kswapd*** is turned on. This is because when ***kswapd*** is turned on, reclaim operations are triggered and the page re-fault ratio becomes higher than that when ***kswapd*** is turned off. When the free pages are used up, without ***kswapd***, direct reclaiming will be triggered to reclaim free pages and thus the performance will be worse than with ***kswapd***.

The above experimental results show that there is a tradeoff between the reclaim size and the overall performance. The preliminary idea is to exploit the tradeoff to find the optimal reclaim size for Android smartphones.

6 Related Work

Buddy system has been used to manage memory for many years. Many previous works were focusing on the design of buddy system for managing memory. Burton [2] proposed a generalized buddy system. By using the Fibonacci numbers as block size, Knuth [7] proposed the Fibonacci buddy system. Moreover, this idea was complemented by Hirschberg [6], and was optimized by Hinds [5], Cranston and Thomas [3] to locate buddies in time similar to the binary buddy system. Shen and Peterson [14] proposed the weighted buddy system. Page and Hagins [12] proposed the dual buddy system, an improvement to the weighted buddy system, to reduce the amount of fragmentation to that of the binary buddy system. A buddy system designed for disk-file layout with high storage utilization was proposed by Koch [8]. Brodal et al. [1] improved buddy system for fast allocation and deallocation. Marotta et al. [10] proposed non-blocking buddy system for scalable memory allocation on multi-core machines. This work shows that the existing buddy system is not working well for Android smartphones.

7 Conclusion

Existing Linux memory management algorithms are designed for servers and PCs. Android inherits Linux kernel and thus the memory management algorithms are transplanted to smartphones. The experimental results show that these algorithms mismatch the characteristics of apps running on Android smartphones. First, the large-size reclaiming induces high page re-fault ratio on smartphones when running popular apps and thus degrades the performance. Moreover, the limited reclaim scope aggravates this punishment. Through comprehensive analysis, a tradeoff between the reclaim size and the performance is uncovered. To improve performance, a preliminary idea is proposed to exploit this tradeoff.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd Dr. Schindler for their feedbacks and guidance. This work is supported by National Science Foundation of China (NSFC) 61572411.

8 Discussion Topics

According to the previous experimental results and analysis, there are two additional preliminary ideas which could improve the performance of Android smartphones.

Idea 1: For mobile devices, background status and foreground status should be considered in the priority decision of reclaiming. For example, the reclaiming procedure could evict some *active_anonymous* pages of background processes before evicting *active_file* pages of foreground processes. This is because the pages belong to foreground processes are much more important than the pages belong to background processes for user experience on mobile devices.

Idea 2: The order used to organize free pages of buddy system should be reduced according to the characteristics of requests on mobile devices. For example, the maximum order (default 11) could be reduced to 9, because the maximum request size of mobile devices is 2^8 pages.

The order of allocation on Android smartphones is usually smaller than 4, but the maximum order used to organize free pages is 11. This will degrade the efficiency of allocation operations. According to the allocation procedure of buddy system, when the small-order free space is used up, buddy system has to breakdown the large-order free space to satisfy the allocation application and then insert the remaining part into buddy system. An example of this case is shown in Figure 9.

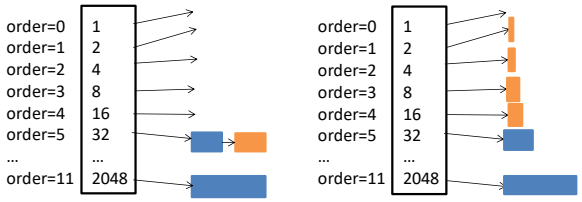

Figure 9: Allocation procedure of buddy system for 2 free pages.

In the example, when a process requests 2 free pages, the list with order=1 will be checked in buddy system first. There is no free page, then the list with order=2 will be checked and so on. Until checking the list with order=5, there are free pages, thus 32 pages will be broken down and 2 pages will be allocated to the process and remaining part will be inserted to other lists. Thus, a large maximum order could degrade the efficiency of buddy system.

References

[1] BRODAL, G. S., DEMAINE, E. D., AND MUNRO, J. I. Fast allocation and deallocation with an improved buddy system. *Acta Informatica 41*, 4 (Mar 2005), 273–291.

[2] BURTON, W. A buddy system variation for disk storage allocation. *Commun. ACM 19*, 7 (July 1976), 416–417.

[3] CRANSTON, B., AND THOMAS, R. A simplified recombination scheme for the fibonacci buddy system. *Commun. ACM 18*, 6 (June 1975), 331–332.

[4] ENGINEERS. Android debug bridge (adb) tool. https://androidmtk.com/download-minimal-adb-and-fastboot-tool, 2019.

[5] HINDS, J. A. An algorithm for locating adjacent storage blocks in the buddy system. *Commun. ACM 18*, 4 (Apr. 1975), 221–222.

[6] HIRSCHBERG, D. S. A class of dynamic memory allocation algorithms. *Commun. ACM 16*, 10 (Oct. 1973), 615–618.

[7] KNUTH, D. Dynamic storage allocation. *In: The art of computer programming 1*, 435–455.

[8] KOCH, P. D. L. Disk file allocation based on the buddy system. *ACM Trans. Comput. Syst. 5*, 4 (Oct. 1987), 352–370.

[9] LEE, K., AND WON, Y. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT)* (2012), ACM, pp. 23–32.

[10] MAROTTA, R., IANNI, M., SCARSELLI, A., PELLEGRINI, A., AND QUAGLIA, F. A non-blocking buddy system for scalable memory allocation on multi-core machines. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)* (2018), pp. 164–165.

[11] OH, G., KIM, S., LEE, S.-W., AND MOON, B. Sqlite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1454–1465.

[12] PAGE, AND HAGINS. Improving the performance of buddy systems. *IEEE Transactions on Computers C-35*, 5 (May 1986), 441–447.

[13] PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S., AND LEE, J. Cflru: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (New York, NY, USA, 2006), CASES '06, ACM, pp. 234–241.

[14] SHEN, K. K., AND PETERSON, J. L. A weighted buddy method for dynamic storage allocation. *Commun. ACM 17*, 10 (Oct. 1974), 558–562.

[15] YOO, Y.-S., LEE, H., RYU, Y., AND BAHN, H. Page replacement algorithms for nand flash memory storages. In *Proceedings of the 2007 International Conference on Computational Science and Its Applications - Volume Part I* (Berlin, Heidelberg, 2007), ICCSA'07, Springer-Verlag, pp. 201–212.