This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2019.2932108, IEEE Embedded Systems Letters

Enlarging I/O Size for Faster Loading of Mobile Applications

Yongsoo Joo, *Member, IEEE,* Dongjoo Seo, Dongyun Shin, and Sung-Soo Lim, *Member, IEEE Computer Society*

***Abstract*—As the size of mobile applications grows rapidly, the importance of application loading performance is increasingly emphasized in mobile devices. However, current OSes rely on demand paging to load the working set of applications into memory, which typically generates small size I/Os that are not handled well by mobile flash storage devices.**

We propose an aggressive merging scheme, which consists of an explicit application loading method and a series of optimization techniques: I/O reordering, I/O merging, and I/O padding. The key idea behind our scheme is to enlarge I/O size for application loading to increase the effective storage throughput. Experiments show that our scheme effectively increases the average I/O size by 5.6X, leading to 30% reduction of working set loading time.

***Index Terms*—User perceived performance, application loading performance, mobile application, flash storage.**

I. Introduction

APPLICATION loading time critically affects user-perceived performance, and thus it has been one of the key performance metrics for computing systems. A delay between 100 and 1,000 ms is known to be perceptible, and a delay over 1,000 milliseconds makes users uncomfortable [1]. As the size of mobile applications grows rapidly, the need for optimizing application loading performance is expected to continue or even increase in the foreseeable future.

Modern mobile OSes rely on demand paging to bring code and data pages of user applications into memory. Specifically, it generates I/O requests only for the requested but missed pages. While performing well in most situations, it may reveal the weakness of an underlying mobile storage device such as eMMC and UFS for abruptly changing working sets, e.g., launching applications or resuming background applications.

The I/O behavior of demand paging is represented as low queue depth and small random read I/Os. First, it does not issue multiple I/Os concurrently because the next page fault can occur only after the current page fault has been handled. In other words, the maximum number of outstanding I/O requests, or queue depth, is effectively limited up to one. Second, it issues read I/Os only for the missed page, and thus the resulting I/O size becomes 4 KB, the size of a single page.

Unfortunately, such an I/O pattern does not achieve the maximum I/O throughput of flash storage devices consisting of multiple flash dies. Each die provides lower throughput, but two or more can be activated at the same time to achieve higher throughput. One way to accomplish this is to make a large size I/O, as it is split and sent to the associated flash dies at a time. Another way is to increase the number of outstanding I/Os that are processed simultaneously in the storage. However, demand paging does not exploit either of the two.

We propose an aggressive merging scheme, which consists of an explicit application loading method and a series of optimization techniques: I/O reordering, I/O merging, and I/O padding. Our key idea is to enlarge I/O size for application loading to increase the effective storage throughput.

The authors are with School of Software, Kookmin University, Seoul, 02707 South Korea. e-mail: sslim@kookmin.ac.kr.

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035), and by the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIT) (No. 2018R1D1A1B05044558). Sung-Soo Lim is the corresponding author.

Manuscript received March ??, 2019; revised April ??, 2019.

II. Related Work

Linux readahead [2] is a sequential prefetching technique that detects a sequential I/O pattern to issue readahead I/O requests before next page faults occur. While it can also improve application loading to some extent, it cannot perform global optimization over an entire working set. Hint-based I/O optimization techniques [3][4] make an accurate guess for future I/O requests using hints provided by the target application. However, this approach requires modifying application code. On the other hand, history-based techniques predict future I/O requests of applications by analyzing their past I/O requests, requiring no modification of application code. Application prefetching techniques [5][6][7] and our method fall in this category. While the application prefetchers focus on improving application startup performance, our method is applicable to quick switching between background and foreground applications as well. The idea of explicit loading was previously introduced by Windows Prefetch [6], but it applies only I/O reordering in each file to minimize disk head movement. In contrast, our method works at block level, allowing inter-file optimization, and is effective for flash storage as well.

III. Aggressive Merging Scheme

Explicit application loading. Demand paging implicitly loads an application by fetching only the missed page in an on-demand manner. While working great in keeping track of a continuously changing working set, it severely limits the chance for I/O optimization as the optimization window size is limited to a single I/O request. To overcome the limitation, we suggest to explicitly load the working set at once as follows:

1) Profile the working set of a target application.
2) Decide when to initiate application loading.
3) Pause the target application.
4) Load its working set with I/O optimization techniques.

1943-0663 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2019.2932108, IEEE Embedded Systems Letters

5) Resume the target application.

Profiling working sets. Our approach is complementary to demand paging in that it is applicable only when a working set is predetermined by profiling. Such situations include starting up applications and resuming background applications.

First, for application startup, we can use I/O tracing tools such as *blktrace* [8] to obtain a working set immediately after launch completion. Specifically, we flush the page cache and capture the I/O requests generated during the launch process. Our previous work [5] has demonstrated that this method yields a deterministic set of pages referenced during launch, not varying much across different captures.

Second, for an application returning from background to foreground, the working set immediately before the application enters background becomes of interest. Here we assume that most pages, if not all, in the working set are evicted from memory while the application stays in background. This set of evicted pages can be tracked in the same way the Linux swap system [9] does. In this work, we deal with only the first case, leaving the second for future work.

Initiation of application loading. An OS can precisely detect when a user triggers application startup or activates a background application. However, it is not always beneficial to initiate application loading upon such events. For example, when most pages of the working set remain in memory, triggering application loading will only delay launching or activating the application. The OS should check the page cache status and selectively initiate our method, e.g., only when the portion of the working set residing in the cache is below a threshold. In this work, we assume a cold start scenario, i.e., no page of the working set is in memory, to focus on developing I/O optimization techniques for application loading.

I/O reordering and I/O merging. I/O reordering and I/O merging have long been performed by I/O schedulers [10] for pending I/Os in a queue. While effective for nonblocking I/Os such as write requests and async reads, the I/O schedulers have little effect on application loading processes, which are dominated by blocking read I/Os. Specifically, the next I/O can be issued only after the current blocking read I/O is completed, limiting the maximum number of blocking read I/Os appearing in the I/O queue to one.

Our explicit application loading method has access to a predetermined working set before issuing actual I/O requests, allowing a perfect reordering of all of its page chunks. According to our observation, I/O reordering itself does not affect much the working set loading time on flash storage, which is also reported in prior work [11]. However, it provides more chances for I/O merging, which in turn significantly increases the I/O size while loading the working set.

I/O padding. I/O reordering and I/O merging are conservative in that they do not change the amount of data in the working set. We relax this constraint by deploying an I/O padding technique successively to I/O reordering and I/O merging. It is aggressive in that padding pages can be inserted between adjacent chunks, increasing the amount of data transferred. In this way, I/O padding further increases I/O size, thus improving the effective I/O throughput of application loading.

In order for the I/O padding technique to be successful, the benefit from the enlarged I/Os should exceed the overhead due to padding pages. An optimal padding decision should consider the followings:

- Padding candidate sizes;
- The sizes of the surrounding chunks for each candidate;
- The padding decisions of nearby padding candidates;
- Storage I/O throughput for different transfer sizes.

In the following section, we present an efficient I/O padding technique based on a dynamic programming model.

IV. Proposed I/O Padding Technique

Problem statement. After completing I/O reordering and I/O merging, the working set of a target application with n chunks is given as $W = (b_0, b_1, \ldots, b_{n-1})$, such that b_is are chunks of contiguous pages sorted in ascending LBA (logical block address) order, and every pair of two adjacent chunks has a gap equal or greater than a single page size. Given the input $W_{0,n-1}$, we can define a padding vector $P = (p_0, p_1, \ldots, p_{n-2})$, where p_i is a padding candidate for the gap between b_i and b_{i+1}. Each p_i is set to one if one decides to insert padding pages between b_i and b_{i+1}. For example, $P = (1,1,\ldots,1)$ means that all the gaps in W are filled with padding pages, generating a single I/O request that covers the entire working set.

Now, we state the padding decision problem as follows:

Problem 1: For a given working set W, find the optimal padding vector P such that the loading time of the target application is minimized.

Dynamic programming model. We solve Problem 1 with dynamic programming. First, we denote the subset of W as $W_{i,j} = (b_i, \ldots, b_j)$, and its padding vector as $P_{i,j-1} = (p_i, p_{i+1}, \ldots, p_{j-1})$. Next, let $read(i, j)$ be the storage access time to load $W_{i,j}$ with $P_{i,j-1} = (1,1,\ldots,1)$, i.e., a single I/O request covering the span from b_i to b_j. Finally, let $t(i, j)$ be the time to load $W_{i,j}$ with the optimal padding vector $P_{i,j-1}$.

In Equation (1), we identify the relationship between $t(i, j)$ and its sub-problems $t(s,t)$, where $i \leq s \leq t \leq j$.

$$t(i,j) = \begin{cases} read(i,i), & \text{if } i = j \\ min \begin{cases} read(i,j) \\ t(i,i)+t(i+1,j) \\ t(i,i+1)+t(i+2,j) \\ \cdots \\ t(i,j-2)+t(j-1,j) \\ t(i,j-1)+t(j,j) \end{cases}, & \text{if } i < j \end{cases} \quad (1)$$

First, if $i = j$, there is no opportunity for padding as there is only one chunk b_i, and thus $t(i, j)$ returns the time to load b_i. Second, if $i < j$, a padding vector $P_{i,j-1}$ is initialized as $(X, X, \ldots, X)$, where X represents a padding status which is not determined yet. The *min* function then finds the minimum time to load $W_{i,j}$ by comparing the following padding decisions:

1) Pad all candidates, i.e., $P_{i,j-1} = (1,1,\ldots,1)$, where $t(i, j)$ becomes $read(i, j)$.
2) Choose a single candidate p_s ($i \leq s \leq j-1$), and set p_s to 0 while the remainder to 1. For example, if s is set to

1943-0663 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2019.2932108, IEEE Embedded Systems Letters

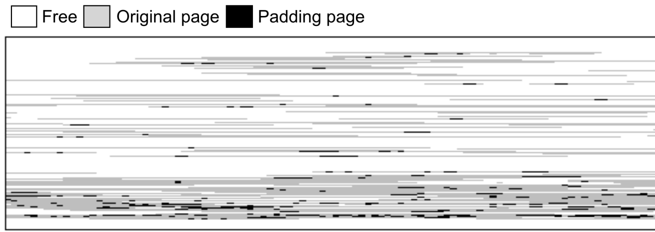

Fig. 1. A part of the logical block map showing the result of I/O padding (application: Messenger).

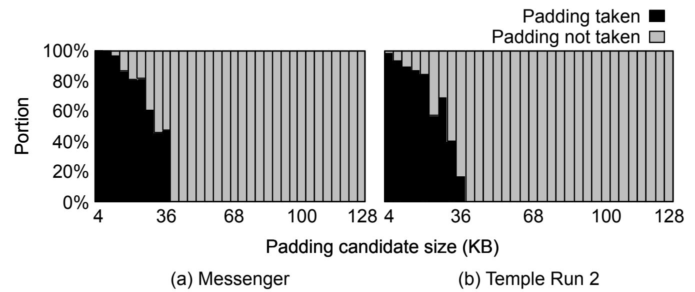

Fig. 2. The breakdown of padding decision. Padding candidates bigger than 128 KB are omitted as all of them were not taken.

$i+1$, $P_{i,j-1}$ becomes $(X,0,X,\ldots,X)$ and $t(i,j)$ becomes $t(i,i+1)+t(i+2,j)$.

Note that $t(i,j)$ can be calculated only after completing evaluation of the underlying sub-problems (e.g., $t(i,i+1)$ and $t(i+2,j)$ for p_s with $s=i+1$). We can use either a top-down approach with memoization or a bottom-up approach with tabulation to evaluate all necessary sub-problems. Both approaches can greatly reduce computation overhead by evaluating each sub-problem no more than once. Finally, for the given input $W_{0,n-1}$, we can obtain the optimal padding vector $P_{0,n-2}$ by solving Equation (1) for $t(0,n-1)$.

Storage performance model. Equation (1) requires a storage performance model that returns read access time for a given starting LBA and size. We develop a measurement-based model that uses only I/O size to estimate read access time. In particular, for a given I/O size, we repeat measuring read latency values with randomly generated LBAs. Among the obtained values, we choose median rather than mean to mitigate the effect due to outliers.

Note that it takes too much time to perform measurement for all possible I/O sizes. In our experimental setup, read access time increased almost linearly with size for I/O requests bigger than 128 KB. Thus, we construct a lookup table for I/O sizes up to 128 KB, while using linear extrapolation for larger I/Os.

V. EVALUATION

Experimental setup. We chose a Google Nexus 5 smartphone with 32 GB of eMMC flash storage as a test platform and chose 16 Android applications for a test set. We captured I/O traces using *blktrace* [8] while launching them, and run our aggressive merging scheme to obtain padding solutions. For the application loading module, we developed an emulator using *pread* [12] system call, which sends I/O requests specified in the given working set to the eMMC storage and measures actual time spent on the device.

I/O padding result. Fig. 1 shows that I/O padding effectively enlarges the size of contiguous block chunks, and thus increasing I/O size while loading Messenger. For all the test applications, average I/O size was increased by 5.6X. Fig. 2 depicts how the I/O padding technique makes padding decisions for different size of candidates. It appeared to prefer smaller candidates to be padded, while it selected only some of the same size candidates to be padded, implying that padding the others could not reduce loading time.

Application loading time. Fig. 3 shows the measured loading time with different I/O optimizations. I/O reordering itself had little impact on loading time, but allowed I/O merging to reduce loading time by 19.3%. I/O padding could further reduce loading time to yield a total reduction of 29.9%. The performance of I/O merging and padding appeared to be significantly different for different applications. For Temple Run 2, I/O merging achieved 27.1% reduction, while the additional reduction by I/O padding was only 2.9% point. In contrast, for Messenger, I/O merging achieved only 8.0% reduction, but I/O padding could add 25.1% point reduction.

Interference by background I/O. We repeated the same test of Fig. 3 with running three different background processes to observe how background I/O affects application loading time. Video playback, FTP download, and application update represent weak, medium, and high I/O intensity, respectively. To reproduce deterministic I/O interferences, we captured the background I/O sequence of each process using *blktrace* [8], and then abstracted it in terms of R/W ratio, I/O per seconds, and I/O size. Finally, we configured *fio* [13] accordingly and measured loading time to obtain the result of Fig. 4. It shows that the higher the background I/O intensity, the greater the relative performance gap between the baseline and our method.

Interference by Linux readahead. Both our method and Linux readahead perform optimization by modifying original I/O requests issued by applications, and thus two can affect each other. Fig. 5 shows that selectively disabling readahead during working set profiling leads to a further performance gain of I/O padding. However, even temporarily disabling readahead can severely degrade I/O performance of other processes, as shown in the baseline configuration. Hence, we plan to expand our method to fully exploit this chance for optimization while not affecting the runtime I/O performance achieved by Linux readahead.

Computation overhead. I/O reordering and I/O merging spent negligible computation time for our test applications, e.g., 14 ms for Messenger. On the other hand, I/O padding runs dynamic programming, which incurred considerable time and space overhead. Hence, we split the original working set into multiple subsets at every occurrence of padding candidates greater than 128 KB, without affecting solution quality (refer to Fig. 2). For Messenger, we could reduce the table size of the dynamic programming from 555,985 (2,171 KB) to 3,756 (15 KB), and its computation time from 213s to 0.07s.

Padding overhead. Padding pages may incur both memory

1943-0663 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2019.2932108, IEEE Embedded Systems Letters

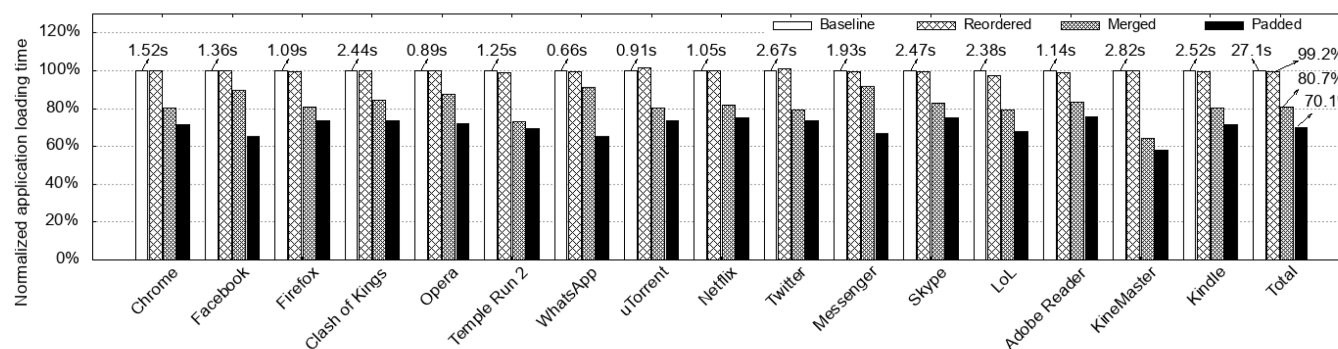

Fig. 3. Average loading time of 16 Android applications, for each of which ten measurement were performed. Baseline: original I/Os requested by demand paging. Reordered, Merged, Padded: optimization techniques are applied cumulatively from I/O reordering, I/O merging, and I/O padding.

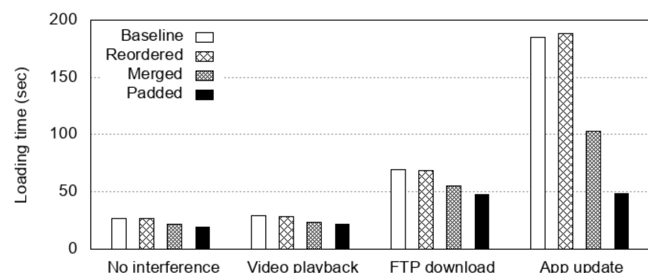

Fig. 4. Application loading performance under background I/O interference. All the 16 applications were loaded in succession.

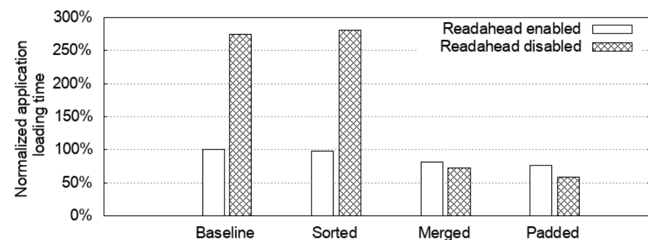

Fig. 5. The effect of readahead on the application loading performance. Readahead was selectively enabled or disabled only for profiling phases. When measuring application loading time, readahead was always enabled.

and energy overhead as they are not originally included in the working set of applications. For all test applications, the total size of padding pages was 85.0 MB (7.4% of the original working set), which may evict other pages used by other applications. We avoided this problem by discarding padding pages as soon as they are loaded into the application loading module. Hence, only the pages in the working set are eventually sent to the OS page cache. Nevertheless, the energy consumption of the flash storage might change because the I/O requests themselves for padding pages are not eliminated. Hence, we measured the energy consumption of loading the working sets of all test applications. We used the Battery Historian tool and repeated experiments 200 times for each configuration. Fig. 6 shows that contrary to expectation, I/O padding could reduce energy consumption by a third, implying that enlarging I/O size improves not only the I/O throughput of eMMC storage but also its energy efficiency.

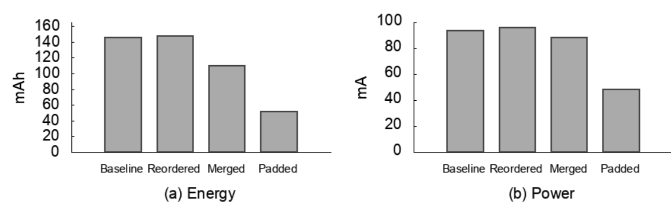

Fig. 6. Energy and power comparison of different configurations.

VI. Conclusion

In this paper, we measured the read throughput of mobile storage for different I/O sizes, leading to the insight that enlarging I/O size can be an effective way to improve application loading performance. We then proposed an aggressive merging scheme, which consists of an explicit application loading method together with I/O reordering, I/O merging, and I/O padding. Finally, we developed a dynamic programming model to find the optimal padding decision for I/O padding. Experiments show that our scheme increased the average I/O size by 5.6X on a Google Nexus 5 smartphone, leading to 30% reduction of application loading time.

References

[1] L. C. Hogan, *Designing for Performance: Weighing Aesthetics and Speed*. O'Reilly Media, Inc., 2014, pp. 11–11.
[2] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the Design of a New Linux Readahead Framework," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 75–84, Jul. 2008.
[3] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," in *Proc. SOSP*, 1995, pp. 79–95.
[4] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson, "Informed Mobile Prefetching," in *Proc. MobiSys*, 2012, pp. 155–168.
[5] Y. Joo, J. Ryu, S. Park, and K. G. Shin, "FAST: Quick Application Launch on Solid-State Drives," in *Proc. FAST*, 2011, pp. 259–272.
[6] M. E. Russinovich and D. Solomon, *Microsoft Windows Internals*, 4th ed. Microsoft Press, 2004, pp. 458–462.
[7] Y. Joo, S. Park, and H. Bahn, "Exploiting I/O Reordering and I/O Interleaving to Improve Application Launch Performance," *ACM Trans. Storage*, vol. 13, no. 1, pp. 8:1–8:17, Feb. 2017.
[8] J. Axboe and A. D. Brunelle, *Blktrace User Guide*, February 2007.
[9] S. Bokhari, "The Linux Operating System," *Computer*, vol. 28, no. 8, pp. 74–79, 1995.
[10] J. Axboe, "Linux Block IO—present and future," in *Proc. Ottawa Linux Symp.*, 2004, pp. 51–61.
[11] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems," in *Proc. SYSTOR*, 2013, pp. 22:1–22:10.
[12] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, "Asynchronous I/O Support in Linux 2.5," in *Proc. Linux Symp.*, 2003, pp. 371–386.
[13] J. Axboe, "Fio-flexible io tester," *http://freecode.com/projects/fio*.

1943-0663 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.