An I/O Scheduling Strategy for Embedded Flash Storage Devices With Mapping Cache

Cheng Ji, Li-Pin Chang, Chao Wu, Liang Shi, and Chun Jason Xue

***Abstract*—NAND flash memory has been the default storage component in embedded systems. One of the key technologies for flash management is the address mapping scheme between logical addresses and physical addresses, which deals with the inability of in-place-updating in flash memory. Demand-based page-level mapping cache is often applied to match the cache size constraint and performance requirement of embedded storage systems. However, recent studies showed that the management overhead of mapping cache schemes is sensitive to the host I/O patterns, especially when the mapping cache is small. This paper presents a novel I/O scheduling scheme, called MAP+, to alleviate this problem. The proposed scheduling approach reorders I/O requests for performance improvement from two angles. Prioritizing the requests that will hit in the mapping cache, and grouping requests with related logical addresses into large batches. Batches of requests are reordered to further optimize request waiting time. Experimental results show that MAP+ improved upon traditional I/O schedulers by 48% and 18% in terms of read and write latencies, respectively.**

***Index Terms*—Embedded system, flash memory performance, I/O scheduling, mapping cache.**

I. Introduction

NAND-FLASH-BASED mobile devices have gained a great success in the past decade. Due to the erase-before-write constraint, flash translation layer (FTL) is proposed to translate logical page numbers (LPNs) into physical page numbers (PPNs) for flash memories. Among various FTL designs, a page-mapping FTL demonstrates better performance than a block-mapping or hybrid-mapping FTL [1]. However, one disadvantage of page-mapping FTLs is the high space overhead of the large mapping structure.

Manuscript received November 21, 2016; revised April 24, 2017; accepted June 28, 2017. Date of publication July 20, 2017; date of current version March 29, 2018. This work was supported in part by NSFC under Grant 61402059 and Grant 61572411, in part by National 863 Program under Grant 2015AA015304, in part by the Ministry of Science and Technology of Taiwan under Grant MOST 104-2221-E-009-011-MY3, in part by the Fundamental Research Funds for the Central Universities under Grant 106112016CDJZR185512, and in part by Huawei Innovation Research Program. This paper was recommended by Associate Editor M. T. Kandemir. *(Corresponding author: Liang Shi.)*

C. Ji, C. Wu, and C. J. Xue are with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: chengji4-c@my.cityu.edu.hk; chaowu6-c@my.cityu.edu.hk; jasonxue@cityu.edu.hk).

L.-P. Chang is with the Department of Computer Science, National Chiao Tung University, Hsinchu 30010, Taiwan (e-mail: lpchang@cs.nctu.edu.tw).

L. Shi is with the College of Computer Science, Chongqing University, Chongqing 400044, China (e-mail: shiliang@cqu.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2017.2729405

Embedded flash storage devices are equipped with very limited embedded RAM because of the stringent constraints on power consumption and hardware budget. For example, in a typical embedded multimedia card (eMMC) for Android phones, the embedded RAM size might be only as large as two flash pages [2]. The scarce embedded memory space is further partitioned between a write buffer and the mapping table [3], [4]. To deploy page-mapping FTLs with very limited RAM space, the demand-based map caching scheme is proposed to cache a small active portion of the entire mapping structure [5]. However, recent studies show that the performance of flash storage based on map caching is highly sensitive to the I/O patterns [6], [7]. Unfriendly access patterns would induce frequent misses in the mapping cache, which lead to performance degradation because of frequent reading and writing of translation pages.

Many prior studies have been proposed to improve the mapping cache performance, such as exploiting the locality in I/O requests to reduce the cache miss ratio [5], [7]–[9]. However, these storage-side methods cannot reshape the I/O patterns and their efficiency is very limited under weak localities of access. Even though when the cache miss ratio is low, the cache management overhead can still noticeably affect system performance under I/O intensive workloads [6]. These observations underline the need for host-side optimization to reduce the mapping cache management cost. Host systems have rich knowledge of I/O behaviors, and thus it provides a good opportunity for such host-side optimization. Many I/O scheduling methods have been proposed considering the properties of flash memory [10]–[12], e.g., scheduling requests to reduce access conflicts among flash chips. However, few attention is paid to I/O scheduling with the consideration of the mapping cache management cost.

In this paper, a novel I/O scheduling approach, called MAP+, which reorders I/O requests for mapping-cache-friendly I/O patterns, is proposed. The design of MAP+ is based on two ideas. First, by exploiting the device-level mapping cache information, MAP+ assigns higher priorities to the I/O requests whose address translation can be resolved without any cache misses. Second, MAP+ groups I/O requests with related logical addresses into large batches to exploit sequential hits in the mapping cache. These two ideas help reshape the I/O pattern to be friendly to the mapping cache. In addition, MAP+ improves the scheduling of batched requests for reduced I/O waiting time. To the best of our knowledge, this paper is the first scheduling method with the awareness of the mapping cache inside of flash storage. A series of experiments were conducted on MAP+, and results show that

MAP+ improved the read and write latencies by 48% and 18%, respectively, compared to conventional I/O schedulers. In summary, this paper makes the following contributions.

1) Proposing a hit prioritized (HP) I/O scheduling scheme, which assigns higher priorities to I/O requests whose address translation can be resolved without any cache misses.
2) Proposing a request batched (RB) I/O scheduling scheme, which groups I/O requests with related logical addresses into large batches to exploit sequential hits in the mapping cache.
3) Proposing a batch sorted (BS) I/O scheduling scheme, which reorders batches in terms of their density to further optimize the waiting time of pending requests.
4) Presenting a series of experiments and evaluating the proposed I/O scheduling approach, showing a significant performance improvement.

The rest of this paper is organized as follows. Sections II and III describe the background and related work, respectively. Section IV presents the performance model and the motivation. Section V presents the design of the proposed mapping cache aware I/O scheduling (MAP) approach. Section VI presents experiments and discussion, and this paper is concluded in Section VII.

II. Background

A. Flash Memory Characteristics

NAND flash-based storage devices have been widely deployed in various computing systems. Compared with traditional hard disks, it exhibits several advantages, such as superior I/O performance and shock resistance. Due to the erase-before-write constraint, a flash page cannot be overwritten until its residing block is erased. Thus, page updates are serviced in different flash pages which are in free states, rather than at their original locations. As free pages becomes fewer, garbage collection will be activated to reclaim free space. However, garbage collection has negative effects on flash storage performance, especially when data inside the same flash block are fragmented [13], [14]. There have been many studies dedicating their efforts to improve garbage collection performance. For example, bounding garbage collection costs plays a critical role in meeting real-time guarantee [15]. On the other hand, valid page migration has been identified to contribute the essential overhead of garbage collection [16]. To improve garbage collection performance, executing garbage collection proactively with exploitation of plane-level SSD parallelism [17] and overwriting SLC flash pages with in-place delta compression [18] are effective in reducing cleaning costs.

B. FTL Mapping Algorithms

FTL is introduced to perform LPN to PPN translation. The translation records are kept in the mapping table, which is an array of mapping entries. Some dedicated flash pages, called translation pages, are reserved to store the mapping table [5]. There are three typical FTL designs, i.e., page-level mapping [19], block-level mapping [20], and hybrid-level mapping [21], [22]. Page-level mapping translates LPNs from host system to PPNs inside flash memory in page granularity, while block-level mapping performs address translation in block granularity. As a flash page is much smaller than a block, page-level mapping benefits from high mapping flexibility at the cost of a large mapping table size. On the other hand, hybrid-mapping has the median RAM requirement for mapping table, because it combines the above two mapping types in one FTL, where page-level mapping and block-level mapping are employed for data blocks and log blocks, respectively. However, although block-level mapping and hybrid-level mapping can reduce RAM footprint, their performance can be severely degraded due to costly block-merge operations [22]. In contrast, page-mapping has been proven showing superior performance than block-mapping and hybrid-mapping because of its efficient garbage collection [1]. To achieve page-level mapping performance with less RAM requirement, adjusting the address mapping schemes in FTL adaptively has been proven to be effective in several studies [23], [24].

III. Related Work

A. Demand-Based Mapping Cache

As the capacity of flash storage increases, addressing mapping management with limited RAM resource becomes a challenge even for block-level mapping [25]. The situation becomes worse for page-level mapping, because page-level mapping requires a large mapping structure due to its fine-grained address translation method. To reduce the space overhead, map caching schemes are proposed to cache an active portion of the mapping structure. However, due to power and cost considerations, embedded flash storage devices have a limited size of RAM that serves as the working space of the FTL and the storage of the mapping table [4]. Because the entire mapping table cannot be stored in the RAM space, demand-based page-level mapping is proposed to selectively cache a small portion of the mapping table. Demanding based flash translation layer (DFTL) [5] was the first FTL design based on the demand-based page-level mapping method. It takes advantage of temporal locality of access to improve the mapping cache hit ratio. Caching mechanism for demand-based flash translation layer (CDFTL) [9] and spatial-locality-aware flash translation layer [8] used a whole translation page as a caching unit to further improve mapping cache hit ratio by exploiting the spatial locality of access. Translation page-level caching mechanism for demand-based flash translation layer [7] employs two-level least recently used (LRU) lists to organize mapping entries in an efficient way for reduced mapping cache management overhead. The localities of I/O requests are well exploited by these works to reduce mapping cache miss frequency. However, the performance of these storage-side mapping cache designs are subject to the characteristics of the host I/O patterns. When the host I/Os demonstrate weak localities, they suffer from high management overheads of the mapping cache, as reported in [7].

B. I/O Scheduler for Flash Storage

Conventional I/O schedulers, including complete fair queueing, no-operation (NOOP), and deadline, are designed and

optimized for the performance model of hard disks. Flash memory has different access characteristics and internal organization compared to hard disks, and therefore the conventional schedulers do not work well on flash storage devices. Recent studies have introduced new I/O scheduling schemes in consideration of efficient flash management. Jung *et al.* [10] proposed an I/O scheduler, called PAQ, to improve random read performance by minimizing conflict among read requests inside of SSDs. They also introduced a device-side I/O scheduling approach to achieve high internal parallelism of SSDs by relaxing the dependency among requests [26]. Park and Shen [27] proposed a new scheduler, called FIOS, to improve scheduling fairness and SSD internal parallelism at the same time. Elyasi *et al.* [28] proposed a scheduling mechanism in controller, called Slacker, to exploit the slack between the subrequests pending at different flash chips for improved SSD response time. Kim *et al.* [12] proposed to assign higher priority to read requests to prevent time-consuming write requests from delaying read requests. Shen and Park [29] proposed to achieve high responsiveness of flash memory without violating fairness by eliminating some unnecessary features specific to disk I/O scheduling. Mao and Wu [30] proposed to assign high priorities to small requests based on the shortest-job-first principle to reduce the waiting time of pending requests for flash memory. Different from many-chip SSDs in which resource utilization is the most critical target for performance improvement, embedded flash storage devices have very different design constraints because they are highly sensitive to hardware costs, i.e., they have limited internal parallelism and small working RAM space. In addition, these existing work do not consider the overhead of mapping cache management, and we shall show in a later section that the overhead can significantly affect I/O performance on embedded flash storage devices.

IV. Model and Motivation

In this section, we first present a model to characterize the performance of flash storage with a mapping cache. Based on this model, the motivation of this paper is then presented.

A. Performance Model

1) Storage Access Time: A mapping cache is commonly adopted by a page-mapping FTL to translate LPN into PPN with limited RAM space. With a mapping cache, the handling of host I/O requests involves: address translation, page access, and garbage collection whenever needed. We assume that translation pages and all the other pages (i.e., those storing host data) use the same bit-cell density, i.e., both in SLC or MLC.

Fig. 1 shows a flow diagram of address translation for the flash storage with demand-based mapping cache. As plotted in the figure, the address translation cost is as follows: if the referred LPN already exists in the mapping cache, then the cost is $T_{\text{RAM\_access}}$ only for RAM access. The time needed for the mapping cache hit is

$$T_{c\_\text{hit}} = T_{\text{RAM}}. \tag{1}$$

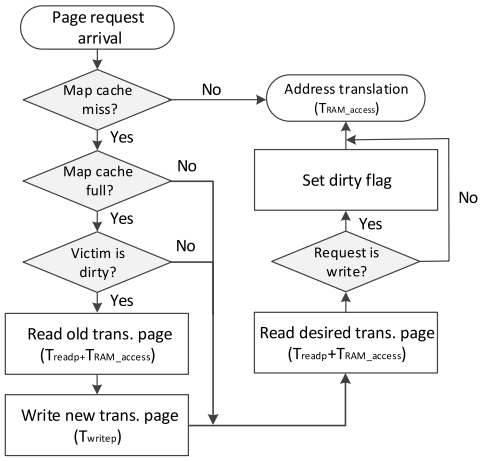


Fig. 1. Flow diagram of address translation with demand-based mapping cache.

If the reference to a new LPN causes a cache miss, the desired mapping entry is fetched from flash with a page read time cost T_{readp} and it will be then loaded into the RAM with cost T_{RAM}. Before fetching the new mapping entry, an old mapping entry is evicted from the cache. With a probability p_{c_dirty} that the evicted entry is dirty. Evicting a dirty entry involves reading an old translation page with a time cost T_{readp}, merging the dirty entry with the existing entries in the old page, and writing a new translation page with up-to-date mapping entries with a time cost T_{writep}. The total time overhead of handling a cache miss is

$$T_{c\_\text{miss}} = p_{c\_\text{dirty}} \times \left(T_{\text{writep}} + T_{\text{readp}} + T_{RAM}\right) + T_{\text{RAM}} + T_{\text{readp}}. \tag{2}$$

The storage access time of reading a logical page is

$$T_r = \left(1 - p_{c\_\text{miss}}\right) \times \left(T_{c\_\text{hit}} + T_{\text{readp}}\right) + p_{c\_\text{miss}} \times \left(T_{c\_\text{miss}} + T_{\text{readp}}\right) \tag{3}$$

and the storage access time of writing a logical page is

$$T_w = \left(1 - p_{c\_\text{miss}}\right) \times \left(T_{c\_\text{hit}} + T_{\text{writep}}\right) + p_{c\_\text{miss}} \times \left(T_{c\_\text{miss}} + T_{\text{writep}}\right). \tag{4}$$

Writing host data can possibly trigger garbage collection, which involves a series of page reads and writes. For simplicity, let garbage collection never be triggered by writing translation pages. Suppose *waf* is the write amplification ratio. Let n_p be the average number of valid pages migrated before a block is erased for garbage collection, and let T_{erase} be the time to erase a block. The storage access time of a host write request of N pages is

$$T_{\text{access\_}w} = N \times (\text{waf} - 1) \times \left(T_{\text{writep}} + T_{\text{readp}} + T_{\text{erase}}/n_p\right) + N \times T_w \tag{5}$$

and the storage access time of a host read request of N pages is

$$T_{\text{access\_}r} = N \times T_r. \tag{6}$$

Because RAM access is fast, T_{c_hit} is negligible. The terms T_r and T_w in the storage access time $T_{\text{access\_}w}$ ($T_{\text{access\_}r}$) of write (read) largely depend on the miss ratio (i.e., p_{c_miss}) and the miss penalty (i.e., T_{c_miss}).

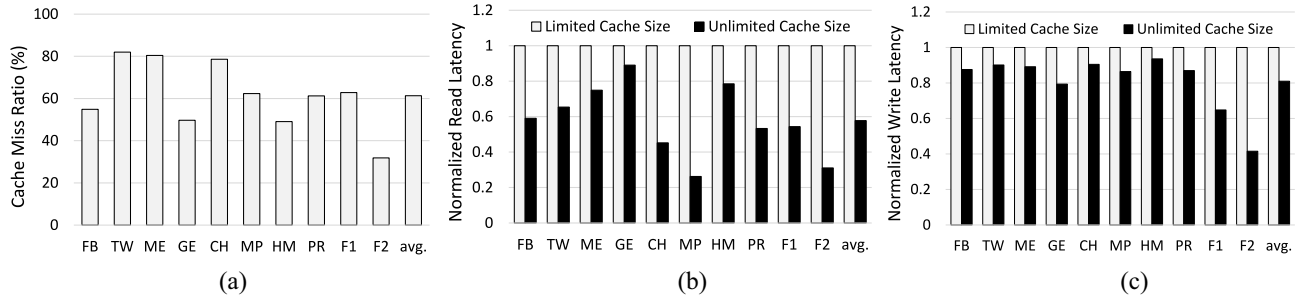


Fig. 2. Comparison between a fixed-sized cache and an unlimited-sized cache in terms of (a) cache miss ratio, (b) read latency, and (c) write latency.

2) Waiting Time: The latency of an I/O request consists of two parts: 1) the storage access time and 2) the waiting time. The waiting time of a request is the time duration between the arrival and the dispatch of the request. The latency of request i is:

$$T^i_{\text{latency}} = T^i_{\text{wait}} + T^i_{\text{access}} \tag{7}$$

where T^i_{wait} and T^i_{access} represent the waiting time and the storage access time of request i, respectively. The storage access time is highly related to the management cost of the mapping cache, as discussed in the prior section. The waiting time, on the other hand, depends on the storage access time of all the requests in the queue that precede the request i. Consider two consecutive requests $i-1$ and i in the queue. The waiting time of the request i is

$$T^i_{\text{wait}} = t^{i-1}_{\text{arrival}} + T^{i-1}_{\text{latency}} - t^i_{\text{arrival}} \tag{8}$$

where t^{i-1}_{arrival} and T^{i-1}_{latency} are the arrival time and latency of the request $i-1$, respectively, and t^i_{arrival} is the arrival time of the request i. The waiting time of the request i depends on the latency of the request $i-1$, which recursively depends on the waiting time and storage access time of the request that precedes the request $i-1$ in the scheduler queue (if any). In other words, if a request experiences a mapping cache miss, not only its storage access time is increased but also the waiting time of every pending request in the queue is lengthened.

B. Motivating Observation

The mapping cache management overhead affects the storage access time, which in turn affects the waiting time of requests. Flash storage devices, especially those embedded storage, have limited RAM space for the mapping cache.

To understand the impact of mapping cache overhead on I/O latency, we conduct a series of experiments based on the NOOP [i.e., first-in first-out (FIFO)] scheduler with a 16 KB mapping cache (the cache parameter settings and trace characteristics can be found in Section VI-A). Fig. 2(a) shows the cache miss ratio under various workloads. Overall, the miss ratios (i.e., p_{c_miss}) are not low, ranging from 32% to 82% (average 61%). Fig. 2(b) and (c) shows the read and write latencies, respectively. For comparison, we also evaluated the latencies with an infinitely large mapping cache. The results show that, with the 16 KB mapping cache, the read and write latency time are degraded by 42% and 19%, respectively. Interestingly, a low miss ratio does not imply a small latency degradation. For example, the miss ratio under the Financial2 workload is about 32%, which is relatively low among those under the other workloads. However, the read and write latencies under the Financial2 workload are degraded by 3.2 times and 2.4 times, respectively, and these degradations are relatively large among those under all workloads. This is because the interarrival times under the Financial2 workload are occasionally short, and the runtime depth of the scheduler pending queue is large. The waiting time of a request is affected by the cache miss penalty of a large number of preceding requests. In this test, the mapping cache flushed as many dirty mapping entries as possible in background to exploit the request interarrival times. However, because write requests arrived in bursts, the mapping cache cannot accommodate all the dirty mapping entries and many incoming write requests were still blocked on cache flushing.

I/O latency can be reduced by improving the mapping cache algorithm for a low miss ratio and reduced overhead of translation page accesses. There have been various mapping cache designs that exploit localities in access patterns. However, these approaches do not reorder requests to improve the waiting time, which is also part of the I/O latency. Differently, in this paper, we propose to improve the mapping cache efficiency and the waiting time through I/O scheduling.

C. Mapping Cache Verification on Embedded Flash Storage Devices

Demand-based map caching is a popular design option for flash-based storage devices, especially embedded flash storage such as eMMC devices [31]. In this section, we attempt to verify the existence of a mapping cache inside of the eMMCs of two smartphones, Ascend P7 (P7) and Galaxy S3 (S3), through tests with different I/O patterns. To minimize the effect of garbage collection inside of eMMCs, the phones were cleared up by a factory reset before each test. We issued 10 000 random reads or writes on the smartphone, and all requests were of 4 KB. To minimize the effect of host page caching, all read requests were direct I/Os and all writes were synchronous. The I/O requests were uniformly distributed within a controlled region in the storage space, ranging from 1 MB to 2 GB. For performance comparison, the block-level traces produced by each test were collected and replayed on a flash-storage simulator with a mapping cache based on LRU replacement.

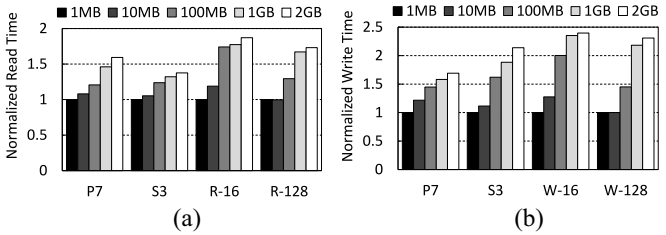


Fig. 3. Total I/O times reported by a simulator and eMMCs in two smartphones with I/O requests uniformly dispersed in a controlled region in the storage space. (a) Random read. (b) Random write.

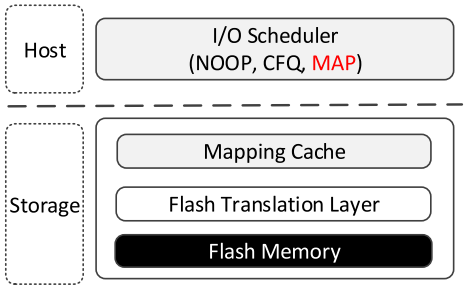


Fig. 4. Architecture of the I/O subsystem in a system that employs flash storage.

We measured the total I/O time on the smartphones and the simulator. The results of R-16 and R-128 denote the total read times reported by the simulator with a 16KB and a 128KB mapping cache, respectively. The results of W-16 and W-128 are defined for writes accordingly. Fig. 3 shows that the smartphones and the simulator produced highly consistent results: the total I/O time increased as the I/O region size was enlarged. Interestingly, even though read request did not produce dirty mapping entries and thus did not introduce translation page writes, the total read time still degraded as the I/O region size enlarged. This is because when the I/O region size was large, the mapping cache experienced more cache misses and the overhead of reading translation pages amplified the total read time. Even though these tests did not reveal the exact mapping-cache design in the eMMCs, the results sufficiently verified that the mapping cache management overhead does affect the I/O performance.

V. Mapping Cache Aware I/O Scheduling

As discussed in Section IV-A, the management overhead of the mapping cache largely affects I/O performance, and the mapping cache misses noticeably prolong the I/O waiting time under many workloads. In this section, we will propose a novel I/O scheduler to deal with this issue. The basic idea of the proposed I/O scheduler is to reorder I/O requests for mapping-cache-friendly I/O patterns. Fig. 4 shows the system architecture of a system that adopts flash-based storage. The proposed method, called mapping-cache-aware (MAP) scheduling, is a pluggable I/O scheduler module in the host operating system.

MAP is designed with two basic ideas. The first idea is *prioritized I/O scheduling*. MAP preferentially schedules the I/O requests whose mapping information are already in the mapping cache inside of the flash storage. In this way, address translation for the prioritized I/O requests can be resolved without cache misses, and thus the request waiting time can be improved by the reduced storage access time. The second idea is *batched I/O scheduling*. MAP groups the pending I/O requests whose mapping information are stored in the same translation page as a batch. Because logically continuous pages store their mapping information in the same flash page, the mapping cache can handle address translation for these pages by reading a single page of mapping information. These two ideas exploit temporal and spatial localities among the pending I/O requests for better system I/O performance. Lastly, MAP is enhanced to MAP+ by *sorting the batches* in consideration of their density to further optimize the waiting time of pending requests.

A. Hit Prioritized I/O Scheduling

The I/O pattern generated by the host system can largely affect the management cost of the mapping cache. If the host issues many I/O requests whose mapping information are already in the mapping cache, all the address translation can be resolved without any cache misses, thereby reducing the waiting time of subsequent I/O requests.

We propose HP I/O scheduling, which is designed to schedule the I/O requests according to the presence of their mapping information in the mapping cache of the flash storage. To realize the HP, the scheduling queue (or pending queue) is reorganized into two separate queues: 1) a hitting queue and 2) a missing queue. Hitting queue consists of the requests that have their corresponding mapping entries cached in the mapping cache, and all the other requests are in the missing queue. The I/O scheduling works as follows. When a new request is submitted for scheduling, the scheduler will use its LPNs to check whether the mapping information of the request is cached or not. The check can be done either using a shadow copy of the mapping cache or based on cache simulation. We shall show in Section V-D how these two options can be implemented. Now, if the mapping information of the request is cached, then the request will be added to the hitting queue; otherwise, it is added to the missing queue. Later on, during runtime, the requests in the hitting queue are always dispatched to the flash storage before those in the missing queue, because they will not cause misses in the mapping cache. If the hitting queue is empty, requests in the missing queue are dispatched. Note that starvation may happen with this new scheduling policy. To prevent requests in the missing queue from starving, we set a deadline for the requests in missing queues. In this paper, the default deadline is set to 10 ms and the effects of deadline will be discussed in Section VI-B3.

Fig. 5 shows a comparison between the HP scheme and NOOP, which has been suggested for SSDs [32]. In this example, Fig. 5(d) shows that the mapping cache contains six mapping entries, and Fig. 5(a) and (b) shows that there are twelve requests in the scheduling queue. For NOOP, the arrivals of requests 1–4 all cause cache misses, and these misses prolong the waiting time of all subsequent requests. Now, with the HP scheme, requests are sorted to the hitting and missing queue. Requests 5 and 8–12 are added to the

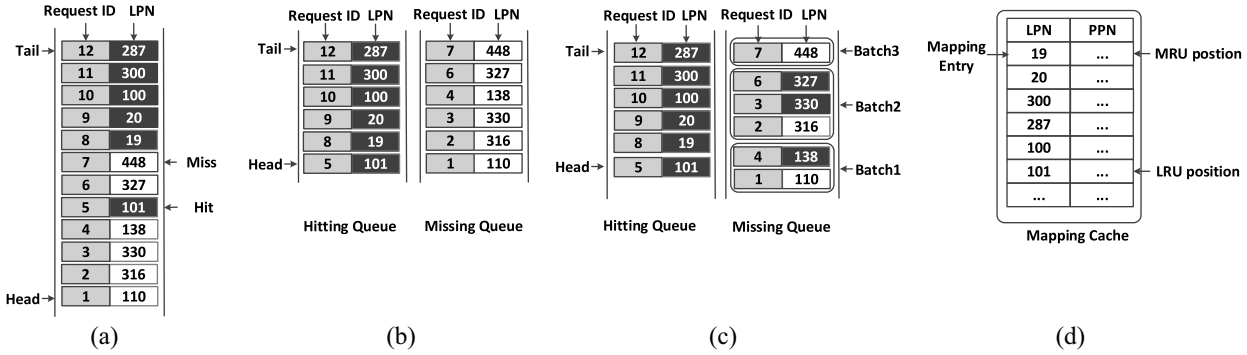


Fig. 5. Schedule comparison among (a) NOOP, (b) HP, and (c) MAP (HP plus RB). (d) Cached mapping entries.

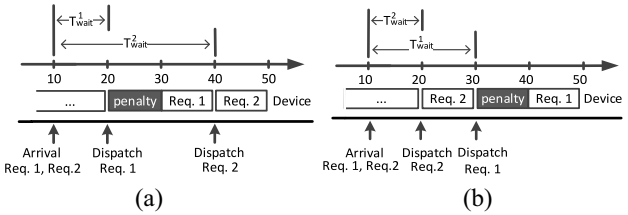


Fig. 6. Comparison of the waiting time between scheduling with and without the HP scheme. (a) Without HP scheme. (b) With HP scheme.

hitting queue because their mapping information are cached, while the rest of the requests are added to the missing queue. The requests in the hitting queue are dispatched first and then those in the missing queue. Servicing the requests from the hitting queue will not cause any cache miss, and thus the requests in the missing queue also enjoy short waiting time.

As discussed in Section IV-A2, the waiting time of a request depends on the storage access time of its preceding requests. HP reduces the overall waiting time by assigning high priorities to the requests whose mapping information are present in the mapping cache. Fig. 6 shows an example on how the HP scheme reduces the overall waiting time. Fig. 6(a) shows a conventional method based on first-in and first-out. We assume that Requests 1 and 2 both arrive at $t = 10$. Later on, at $t = 20$, the flash storage is ready for the next request and Request 1 is then dispatched. Because Request 1 causes a miss in the mapping cache, its storage access time involves a cache miss penalty, and Request 2 has to wait until Request 1 completes. Request 2 is dispatched at $t = 40$. In this example, the waiting time of Request 1 (T^1_{wait}) and Request 2 (T^2_{wait}) are 10 and 30, respectively, and the average waiting time is 20. By contrast, Fig. 6(b) shows that HP gives Request 2 a higher priority because the request does not involve a cache miss. Even though the storage access time of the two requests remain the same, the waiting time of Requests 1 and 2 are reduced to 20 and 10, respectively, and the average waiting time is reduced to 15.

B. Request Batched I/O Scheduling

To further reduce the impact of mapping cache misses on I/O waiting time, we propose an aggressive scheduling scheme, called RB I/O scheduling (RB), to reduce the total number of translation page reads during requests processing. The basic idea of the RB scheme is twofold: first, the requests whose mapping entries are stored in the same translation page will be grouped as a batch, and second, upon the arrival of an I/O batch, the mapping cache will load necessary mapping entries in the translation page associated with the batched request.

To realize the RB scheme, requests in the missing queue (of the HP scheme) is regrouped according to their association of translation page. When a new request arrives at the scheduler, its LPN is checked to determine whether the request can be added to the hitting queue, as it is in the HP scheme. If the request is forwarded to the missing queue, its LPN is checked again to add the request to an existing batch in the missing queue. If the request does not belong to any existing batch, a new batch is created for it. Now, the scheduler dispatches requests in the hitting queue before those in the missing queue, as it does in the HP scheme. Differently, when the hitting queue is empty, the scheduler dispatches the requests in the missing queue in terms of batches. Notice that this design does not intend to reduce the frequency of cache misses. Instead, it eliminates redundant translation page reads by prefetching necessary mapping entries in a translation page upon the arrival of a batch. In this way, in a batch, only the first request causes a translation page read, but the rest of the requests enjoy cache hits. Reducing the frequency of translation page reads can also reduce the waiting time of all pending requests. If the head (first) request in a batch consists of more than one page, upon the prefetching of mapping entries, only the first page required by this request will cause a miss, and all the remaining pages will all enjoy mapping cache hits.

Notice that our RB scheme is different from the request merging operations in conventional schedulers in two aspects. First, requests in a batch need not be strictly sequential, provided that they share the same translation page. Second, upon the arrival of an I/O batch, the flash storage controller will be notified to load all the necessary mapping information in the translation page that is associated with the I/O batch.

When HP is enhanced by RB, we refer to the scheduling as MAP. Fig. 5(c) shows how the MAP scheme works. Assume that a translation page stores the mapping entries of 100 sequential pages. Compared with HP, the MAP scheme groups the pending requests into three batches according to their LPNs. For example, Requests 1 and 4 are in Batch 1

TABLE I
I/O PATTERNS PRODUCED BY DIFFERENT SCHEDULING SCHEMES AND THEIR PRIORITIZED REQUEST NUMBER (PRN). THE PRIORITIZED REQUESTS ARE UNDERLINED

Scheme	I/O Sequence with Req. ID	PRN
NOOP	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	0
Hit Prioritized	5, 8, 9, 10, 11, 12, 1, 2, 3, 4, 6, 7	6
MAP	5, 8, 9, 10, 11, 12, 1, 4, 2, 3, 6, 7	9

because they share the same translation page. When Batch 1 is dispatched to the flash storage, the first request, Request 1, causes a cache miss, and the mapping cache loads necessary mapping entries in the demanded translation page. Later on, the arrival of Request 4 will not cause a cache miss. By contrast, if the MAP scheme is not adopted, because the mapping cache does not aggressively prefetch mapping entries for upcoming requests, the arrival of every request in the missing queue can cause a cache miss.

In Table I, we summarize the I/O sequences produced by the three different scheduling schemes, and underline the requests that can have mapping cache hits for better comparison. When HP is applied alone, the requests that will not cause cache misses are prioritized, and therefore their fast responses are helpful to reduce the waiting time of pending requests. When MAP is employed (HP plus RB), it not only prioritizes the cache-hit requests to reduce the overall waiting time but also issues request batches to reduce the frequency of translation page reads.

C. Batch Sorted I/O Scheduling

The RB scheme reduces the frequency of translation page reads through mapping entry prefetching for a batch of requests that share the same translation page. It dispatches batches in the missing queue in an FIFO manner. However, this simple policy may overlook some opportunities for latency time optimization. The average latency of I/O requests is inversely proportional to the I/O completion rate, i.e., the number of requests completed per unit of time. Because the storage access time on flash storage is in general proportional to request size [30], [33], dispatching a batch that contains a small number of large requests will degrade the I/O completion rate and thus negatively impact on the average I/O latency.

We propose an enhancement of the RB scheme, called BS, to further optimize the request waiting time. The BS scheme is based on the RB scheme: in the missing queue, requests that share the same translation page are grouped into a batch. Different from RB that dispatches batches in the FIFO manner, when selecting the next batch in the missing queue for dispatching, the BS scheme tries to increase the I/O completion rate. For this purpose, we introduce *batch density*, which stands for the density of requests in a batch. The batch density of Batch i, Density_i, is defined as

$$\text{Density}_i = \frac{\text{Number}_i}{\text{Count}_i}. \tag{9}$$

In the formula, Number_i and Count_i are the total number of requests in Batch i and the total number of pages in Batch i, respectively.

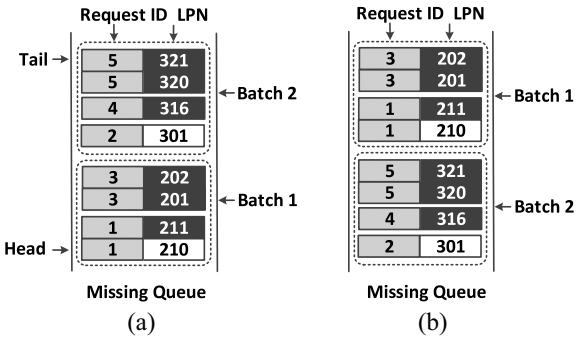


Fig. 7. Comparison between (a) RB and (b) BS scheme.

TABLE II
STORAGE ACCESS TIME (ACCESS), WAITING TIME (WAIT.) AND I/O LATENCY (LATE.) UNDER RB AND BS SCHEME (UNIT: CYCLE)

	RB scheme			BS scheme		
	Access	Wait.	Late.	Access	Wait.	Late.
Req.1	2+1	0	3	2+1	5	8
Req.2	1+1	5	7	1+1	0	2
Req.3	2	3	5	2	8	10
Req.4	1	7	8	1	2	3
Req.5	2	8	10	2	3	5
Total	11	23	33	11	18	28

Based on the definition in (9), a batch has a high density if it contains a large number of requests in a few pages. The BS scheme reorders batches in the missing queue by their densities. When selecting the next batch for dispatching, the BS scheme selects the batch whose density is the largest. By this design, the BS scheme aims at increasing the completion rate of requests, and thus reducing the average request latency time.

Fig. 7 shows an example about how the BS scheme improves upon the RB scheme in terms of I/O latency. Suppose that five read requests arrive at the missing queue at the same time in the order of Requests 1–5. Let the time overhead of reading a page be one unit of time. The arrivals of Request 1 and Request 2 created Batch 1 and Batch 2, respectively, because the two requests are associated with different translation pages. Later on, Request 3 joins Batch 1, while Request 4 and 5 join Batch 2. Fig. 5(a) shows that with the RB scheme, Batch 1 will be dispatched before Batch 2 because it is created earlier. The latency time of Request 1 involves two units of time for reading two pages plus one extra unit of time for reading a translation page. Request 1 involves reading a translation page because it is the first request in a batch. The latency time of Request 3 involves three units of waiting time (from Request 1) plus two units of time for reading two pages. As listed in Table II, the average latency is $(3+7+5+8+10)/5 = 6.6$ units of time.

The BS scheme sorts batches in terms of their densities. Batch 1 has two requests in four pages, and thus its density is $2/4 = 0.5$. The density of Batch 2 is $3/4 = 0.75$. The BS scheme dispatches Batch 2 and then Batch 1 in the descending order of density. As Table II shows, the average latency of requests is improved to $(8+2+10+3+5)/5 = 5.6$ units of time. By dispatching a batch of a large density, the BS scheme

completes a large number of requests in a small amount of time, and thus many requests enjoy small latency. By contrast, the RB scheme may dispatch a large batch of a few requests, and servicing the large batch will increase the waiting time of requests in subsequent batches.

D. Implementation and Overhead Analysis

The MAP scheduler maintains a hitting queue and a missing queue. The HP scheme uses the hitting queue, in which requests will not cause cache misses. The RB uses the missing queue, in which requests are organized in terms of their translation page association. The rest of this section is on the implementation details of the HP and RB schemes.

1) Hit Prioritized: An implementation challenge of the HP scheme is that the HP scheme needs to know the information of the mapping cache inside of the flash storage for request reordering. There can be two options for HP to obtain the device-side mapping cache information from the host side.

The first option is to add a new block command so that the scheduler can retrieve the device-side mapping cache information through the new command. The new block command can be implemented as a vendor-specific command, and the command can be based on existing I/O protocols. Periodically, the host issues the new command to retrieve the LPNs of all cached mapping entries. Because realistic workloads do not frequently change their working sets, the host-side information needs not be updated in real time. In addition, because the mapping cache is small (between 4 KB and 128 KB in this paper), the periodic mapping information retrieval will not impose noticeable performance overhead. There have been several studies that employ new block commands to share information between the host and the flash storage. In particular, Lee *et al.* [34] proposed passing the LPNs of dirty cached pages in the operating system down to the FTL so that garbage collection in the flash storage can avoid involving any soon-to-be-invalidated data. Ouyang *et al.* [35] proposed a hardware/software co-designed storage system to maximize SSD performance by exposing each flash channel to host applications.

We tested HP with this design option. Let the new block command be issued to update the host-side information every 1000 host requests. Based on the workloads in Section VI-A, we found that even at this low update frequency, the average I/O latency is only 2.2% worse than that of HP with full awareness of the device-side mapping cache.

The second option eliminates the potential overhead of using new block commands: the host system software can simulate a counterpart of the device-side mapping cache using the same mapping cache parameters, including the cache algorithm and the cache size. The HP scheme queries the host-side counterpart to determine whether a request should be sorted to the hitting queue or the missing queue. Provided that garbage collection inside of the flash storage does not affect the cached mapping entries (which is true for some mapping cache designs like DFTL [5]), the similarity between the host-side counterpart and the device-side mapping cache will be high. However, the device-side cache algorithm is often a piece of confidential information. We propose to employ the basic LRU algorithm as the host-side counterpart to obtain approximate information of the device-side mapping cache. The rationale behind this design is that the fundamental concept of cache algorithms is to keep the mapping entries that will be referenced again based on the principle of locality, and this concept is shared between LRU and any cache algorithm. This design requires only the mapping cache size, which is less confidential compared to the cache algorithm.

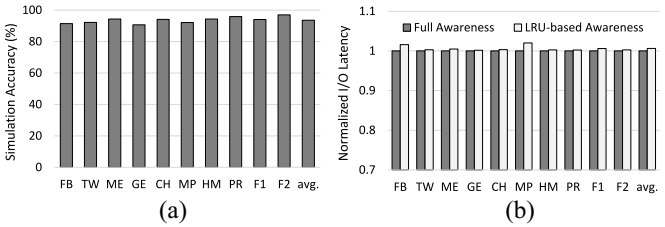


Fig. 8. Performance comparison between full mapping cache awareness and simulation-based awareness. (a) Cache hit ratio. (b) System I/O latency.

To evaluate the usefulness of the second option, we considered DFTL as the device-side mapping cache algorithm, and LRU as the host-side counterpart. We simulated the LRU and DFTL using all the workloads in Section VI-A. We found that on average, 93% of the mapping entries in the LRU are also present in the DFTL, as shown in Fig. 8(a). The small difference is caused by that DFTL employs a segmented LRU design (the protected segment size: probationary segment size is 1:7 in this test). With this high accuracy, the I/O latency of HP based on the host LRU is only 0.5% worse than that of HP based on full awareness of the device-side mapping cache, as shown in Fig. 8(b). This is because HP scheme relies on a small portion of the cached mapping entries, where most of the prioritized requests can already hit in the real mapping cache, and thus the inaccurate cache simulation results have minimal impact on the performance of the prioritized requests.

We suggest the second design option because of its simplicity. In our experiments, we assume the full mapping cache awareness since it is able to show the performance benefit of HP.

2) Request Batched: The RB scheme involves batch I/O submission and device notification for loading mapping entries. Batch I/O submission has been supported by state-of-the-art embedded storage protocols. For example, the packed command in eMMC(5.0) [36] is able to issue requests in batches. The benefit of using batch I/O submission is to reduce the frequency of context switches because an I/O batch involves only one interrupt. If batch I/O submission is not supported, especially for low-end embedded flash storage, the RB scheme can still issue the requests in a batch to the flash storage one by one, without the benefit of context-switch overhead reduction. The other requirement of the RB scheme implementation is to notify the flash storage controller to load necessary mapping entries in the translation page associated with an I/O batch. This notification can be implemented using advanced block commands, such as the data tag mechanism of eMMC [37]. Note that the RB scheme can be a stand-alone scheduling scheme, or be combined with the HP scheme. Also note that the mapping entries in

Algorithm 1 Mapping Cache Aware I/O Scheduling

```
Input: R: I/O Request
       Batch: Requests that share the same translation page
 1: State = Query(R);
 2: if State == Hit then
 3:     Add_to_Queue(R, Hit_Queue);
 4: else
 5:     if Index_Batch(R, Miss_Queue) == True then
 6:         Add_to_Batch(R, Miss_Queue);
 7:     else
 8:         Creat_New_Batch(R, Miss_Queue);
 9:     end if
10: end if
11: if Hit_Queue ! = NULL then
12:     R = Head_Req(Hit_Queue);
13:     Issue R;
14: else
15:     Batch = Sched_Batch(Miss_Queue);
16:     Issue Batch;
17: end if
```

a translation page have consecutive LPNs and they share the same LPB prefix. Because requests can be put in a batch only if they share the same translation page, the batching of requests is independent of the accuracy of the host-side cache simulation.

3) Batch Sorted: The BS scheme can replace the RB scheme for performance enhancement. The BS scheme sorts pending batches in terms of their densities. The density of a batch is recalculated when a new request joins a batch. Batches can be inserted to a balanced search tree for efficient lookup and sorting. With the BS scheme, the host notifies the flash storage of prefetching mapping entries in the same way as with the RB scheme.

4) Implementation: Algorithm 1 details the implementation of MAP. Once an I/O request arrives at the scheduling queue, the scheduler first queries the mapping cache to check whether the incoming request will have a hit in the mapping cache (line 1) or not. If yes, the request is added into the hitting queue (Hit_Queue) (lines 2 and 3). Otherwise, the request is added to the missing queue (Miss_Queue) using the following steps: Index_Batch() first searches the missing queue for an existing batch that share the same translation page with the incoming request (lines 4 and 5). If such a batch exists, the incoming requests will be added into this batch using Add_to_Batch() (line 6). If no such batch is available, a new batch is created for the request using Create_New_Batch() (lines 7–10). Lines 11–17 show the runtime dispatch policy. If the hitting queue is not empty, its head request, which has the earliest arrival time, will be dispatched to the flash storage (lines 11–14). If the hitting queue is empty, the scheduler selects a batch from the missing queue using Sched_Batch() and then dispatches all requests in the selected batch to the flash storage (lines 14–17). Sched_Batch() is implemented by either the RB scheme or the BS scheme. The selection is FIFO if RB is employed. If BS is adopted, then the batch with the largest density is selected. To avoid starvation, after the scheduler dispatches a request or batch, it collects requests in the two queues that already meet their deadlines and dispatches them to the flash storage.

5) Overhead Analysis: The space and time overhead of the proposed HP and RB schemes are very limited. The HP scheme can either employ a new block command to retrieve the information from the device-side mapping cache, or perform host-side cache simulation. In either case, the scheduler maintains only the LPNs of the cached entries. Assume that the size of an LPN is N_p bits. The space requirement of storing the LPNs is $N_e \times N_p$ bits, where N_e is the maximal number of mapping entries. Since the mapping cache size is limited, not larger than 128 KB in this paper, the RAM space requirement is very limited. For the RB scheme, it only needs to maintain a batch list, whose cost is negligible. The computation overhead involves cache simulation, mapping cache indexing, and batch looking up. Since the host is equipped with powerful processors, this overhead is considered very limited.

VI. Experiment and Analysis

A. Experimental Setup

In this paper, we use a trace-driven simulator to evaluate the proposed MAP scheduler. As reported in [31], embedded flash storage, such as eMMC, have been using demand-based mapping cache. In our simulation, the mapping cache inside of flash storage is based on DFTL [5]. The page size of flash memory is set to 4 KB, and the size of each mapping entry is 8B. The time overheads of a page read and a page write are 35 us and 350 us, respectively [38]. The simulator is based on SSDSim [39]. We added a mapping cache to the simulator but removed unused features such as multichannel and multiplane structures. Our simulation started with an empty flash to avoid the performance interference caused by flash internal activities, such as garbage collection and wear leveling. Notice that the proposed RB scheme enhances spatial locality for write requests, and this property can further reduce the cost of garbage collection [11]. We believe that the advantage of our approach could be even larger in the presence of garbage collection activities.

Embedded flash storage can only afford very limited RAM space considering the power and hardware costs. For example, the size of the working RAM in typical eMMCs is only tens of kilobytes, in which both data cache and mapping cache are stored. In this paper, the default size of mapping cache is set to 16 KB. This setting is reasonable because there are always several running applications running, even in embedded systems. In this case, each application can only have limited number of entries in the mapping cache. In our experiments, we evaluate five related schemes to show the effectiveness of the proposed schemes.

1) *Read Over Write:* The read-over-write (RoW) scheduler is implemented based on NOOP, and it organizes read and write requests into two scheduling queues, a read queue and a write queue. The read queue has a higher priority over the write queue. In order to avoid write starvation, read queue is prioritized for once, which is similar to the methods in [12].
2) *Amphibian:* Amphibian is on the extreme of waiting time reduction [30]. It sorts requests in terms of their sizes and dispatches the smallest request to the flash storage.

TABLE III
CHARACTERISTICS OF I/O WORKLOADS

Workload	Seq. Ratio	Write Ratio	Avg. Req. Size
Facebook (FB)	11.8%	92.7%	8.1KB
Twitter (TW)	13.7%	72.5%	13.2KB
Messenger (ME)	12.4%	65.2%	9.7KB
G. Earth (GE)	17.1%	99.5%	7.9KB
Chrome (CH)	9.3%	90.5%	12.8KB
Multiple (MP)	4.4%	91.1%	9.6KB
hm_0 (HM)	1.8%	81.7%	11.3KB
proj_3 (PR)	3.1%	0.1%	8.1KB
Financial1 (F1)	0.4%	76.1%	11.6KB
Financial2 (F2)	0.2%	18.8%	5.6KB

In the implementation, Amphibian is enhanced with the RoW strategy: all requests are divided into two queues, one for read requests and the other for write requests. Both queues are sorted by request size, and the read queue is serviced before the write queue. Similar to RoW, read queue cannot be successively prioritized over the write queue to avoid write starvation. Amphibian is not aware of the management overhead of the mapping cache.

3) *HP:* HP here refers to that described in Section V-A with the RoW enhancement. It maintains a hitting queue and a missing queue. Each of the two queues is further divided into a read subqueue and a write subqueue. HP serves in the order of the read hitting queue, write hitting queue, read missing queue, and write missing queue.
4) *RB:* RB stands for using the request batched scheme alone without HP. RB also employs the RoW enhancement. It organizes the pending requests in terms of batches. RB maintains separate batch lists for read requests and write requests. RB favors the read batch list over the write batch list. Batch lists are FIFO.
5) *Mapping Cache Aware I/O Scheduling:* MAP combines HP and RB, as described in Section V-B. HP is applied to the hitting queue, while RB is applied to the missing queue. The RoW enhancement is also employed. MAP dispatches requests in the hitting queue first. If the hitting queue is empty, it dispatches I/O batches from the missing queue.
6) *MAP Enhanced by BS (MAP+):* MAP+ is an extension of MAP. It replaces the RB scheme with the BS scheme.

To simulate an I/O scheduler, we added the queue(s) of the scheduler on top of the flash simulator. I/O requests in a workload were first inserted to the queue(s). The scheduler reordered and dispatched requests downward to the flash simulator. Table III lists our experimental workloads and their I/O characteristics. Because our approach aims at flash storage with very limited resources, eMMC devices in smartphones best fit our target. The FB, TW, ME, GE, and CH workloads were collected from running popular Android applications (as shown in Table III) on a Nexus 5 smartphone using the `blktrace` block I/O tracing mechanism. We collected block I/O events when I/O requests were inserted to the scheduler queue. The Android version was 4.4.4, and the Linux kernel version was 3.4. All the applications were used for at least five minutes. The MP workload was collected from running the aforementioned five applications alternately. Besides the smartphone workloads, we also considered server-class workloads because as pointed out by prior studies, demand-based address mapping is also employed by large-scale NAND flash storage systems for reducing the RAM footprint [25]. Server traces were collected from OLTP applications running at financial institutions [40] and Microsoft Research Cambridge [41]. In our experiments, the default pending queue length was set to 128. In the pending queue, we assume that there is no dependency among I/O requests to reveal the best potential of all the I/O schedulers under evaluation. The deadline employed by all the schedulers is set to 10 ms. This empirical setting can effectively capture any starving requests because most of the I/O requests in our experiments are completed within 5 ms.

B. Experimental Results

1) System Performance: In this experiment, we examine performance improvements of our proposed methods. Fig. 9(a)–(c) shows the read, write, and overall I/O latencies, respectively, all normalized to those of RoW. In order to illustrate the source of the performance improvements, the waiting time and storage access time, which constitute the I/O request latency, are separately presented in Fig. 9(c). Fig. 10 shows the percentages of prioritized requests in HP, RB, MAP, and MAP+.

a) Amphibian: Fig. 9 shows that Amphibian improves upon RoW in terms of both read and write latency. Compared to RoW, Amphibian reduces the read and write latencies by 3% and 8%, respectively. These improvements are attributed to the smallest-request-first strategy, which minimizes the waiting time of pending requests. The read latency improvement is larger than that of write because of the RoW strategy.

b) HP: HP exploits the mapping cache information for efficient I/O processing. As shown in Fig. 9(a) and (b), HP improves the average read and write performance by 1% and 3%, respectively, compared to RoW. This is because HP reduces the request waiting time, as shown in Fig. 9(c). In some cases, the performance improvement of HP is not as good as that of Amphibian. This is because HP considers only the mapping cache management overhead and it ignores any chances to reorder requests to optimizing the request waiting time. This observation suggests that a joint strategy that considers both the mapping cache overhead and the request waiting time may be more successful.

c) RB: RB exploits sequential hits in the mapping cache. Compared to RoW, RB greatly reduce the average I/O latency by 19%. Specifically, the average latency improvement of read and write are 34% and 14%, respectively. In particular, RB greatly reduces the read latency of proj_3 by about 41%, which is the highest improvement in Fig. 9(a). Read receives larger latency improvements because the flash access time of read requests are much shorter than that of write requests. Saving a translation page read can greatly improve the latency of a read request.

RB outperforms Amphibian under the majority of workloads. Because Amphibian is not aware of the mapping cache management overhead, the results indicates that the overhead of translation page reads is dominant to the latency

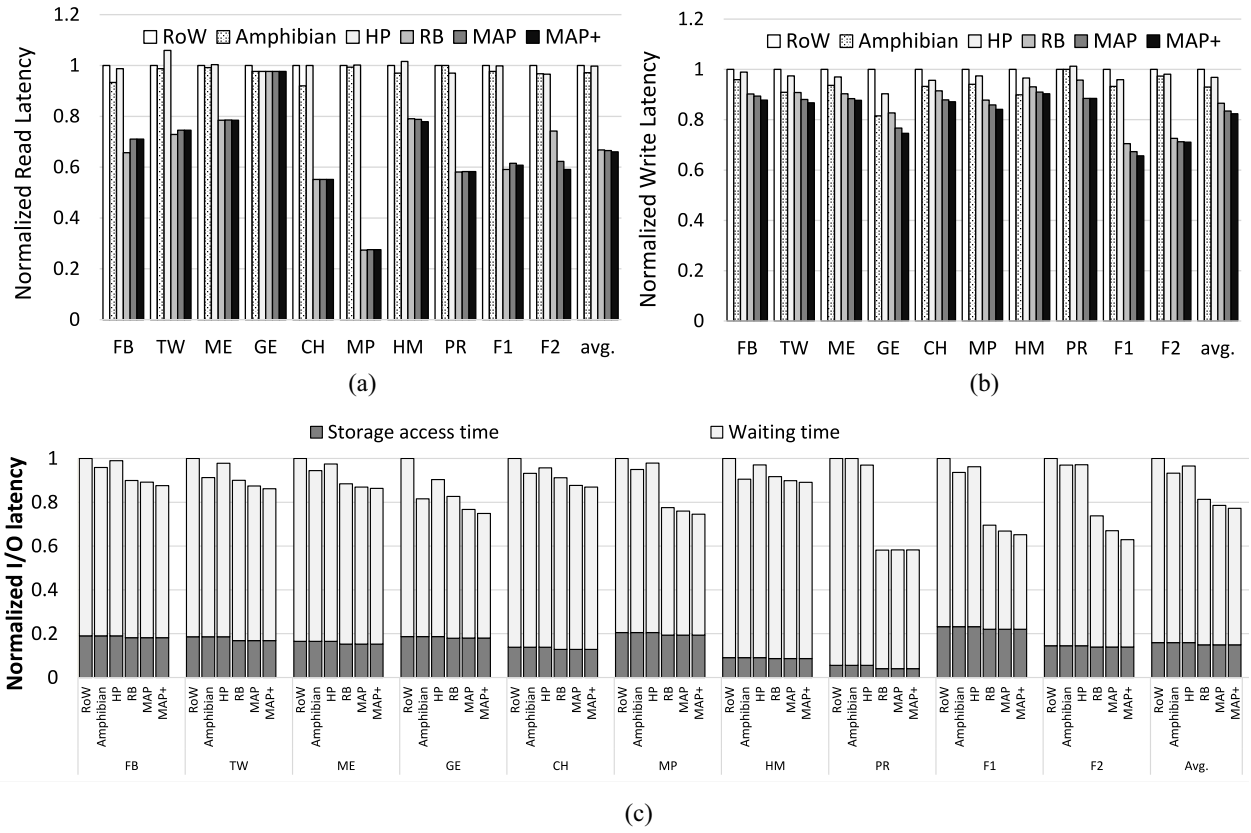


Fig. 9. Performance evaluation for (a) read, (b) write, and (c) overall I/O latency.

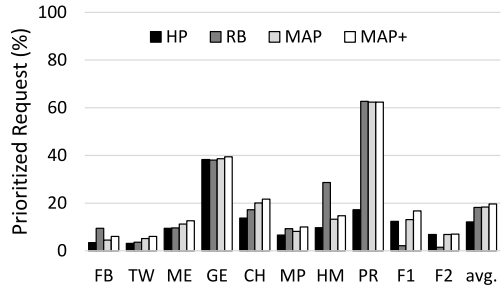


Fig. 10. Comparison among prioritized I/O requests in HP, RB, MAP, and MAP+.

under many workloads. However, RB also under-performs Amphibian under a few workloads, such as hm_0. Since the strategies of RB and Amphibian are orthogonal to each other, our results suggest that the mapping cache management overhead and request size should be jointly considered for the best result.

d) MAP: MAP combines the advantages of HP and RB. As Fig. 9 shows, compared to RoW, MAP significantly improves the average read and write latency by 33% and 16%, respectively. Not surprisingly, inheriting the advantages from HP and RB, it also outperforms HP and RB under most of the workloads. For example, the performance advantage of MAP over HP and RB is significant under the Multiple and Financial2 workloads. This is because these two workloads often produce I/O bursts that congest the scheduler queue. In this case, both prioritizing hitting requests and grouping requests sharing the same translation page are very beneficial to latency improvement. However, MAP slightly underperforms RB under a few write-mostly workloads. For example, in Fig. 9(a), the read latency of MAP is slightly worse than that of RB such as the write-mostly Facebook and Twitter workloads. This is because MAP favors the write requests in the hitting queue over read requests in the missing queue, and this preference could negatively affect the waiting time of read requests, especially under write-mostly workloads.

MAP also achieves latency improvement compared to Amphibian. Even though Amphibian is focused on optimizing request waiting time, it ignores the overhead of mapping cache management. The significant advantage of MAP over Amphibian shows that I/O scheduling without the awareness of the mapping cache will lead to suboptimal results.

e) MAP+: MAP+ enhances MAP by replacing RB with BS. By sorting batches in the missing queue based on density, MAP+ fixes the performance issue of MAP regarding the FIFO batch dispatch policy. Fig. 9(c) shows that MAP+ outperforms all the other approaches under all workloads. In particular, compared to RoW, MAP+ improves the average read and write latency by up to 34% and 18%, respectively.

f) Discussion: Fig. 9(c) shows the breakdown of the overall request latency between storage access time and waiting time. The results show that latency improvement is mainly contributed by the reduction in waiting time, while the storage

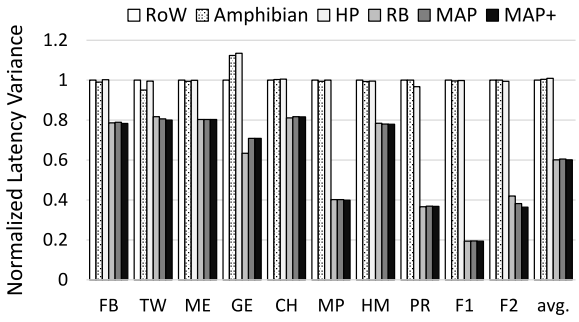


Fig. 11. Normalized latency variation.

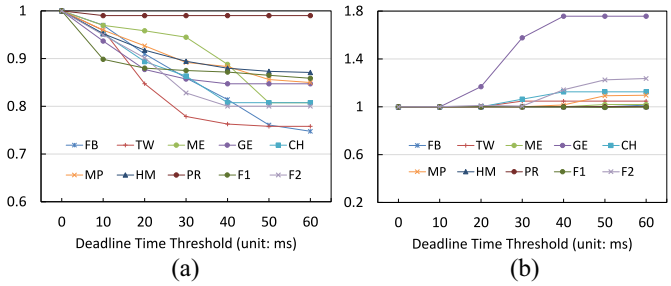


Fig. 12. Studies of average and worst-case I/O latencies under different deadlines. (a) Average I/O latency. (b) Worst-case I/O latency.

access time is slightly reduced by our approaches. However, as explained previously, the flash access time iteratively affect the waiting time of pending requests, a small improvement in the storage access time will be amplified to a large improvement in the waiting time. Fig. 10 shows the percentages of prioritized requests under all workloads. The performance improvement in Fig. 9 is not proportional to the number of preferentially scheduled requests. This is because the benefit of prioritizing a request depends on how many requests are waiting in the scheduling queue. For example, under the Financial2 workload, MAP+ prioritizes only about 7% of all requests but largely improves the average latency by 37%. This is because, as explained earlier, Financial2 occasionally produces I/O bursts that congest the scheduler queue, and prioritizing the cache-hit requests significantly reduces the overall request waiting time.

2) Latency Variation Analysis: We conducted experiments to show the latency variation of different scheduling schemes. Intuitively, the proposed methods could increase the variation because of prioritizing requests. However, interestingly, Fig. 11 shows that the variation is smaller with the proposed methods. This is because MAP+ eliminates the very large latencies caused by the cache miss penalty, and therefore the latencies of all requests become closer to the average latency. Overall, MAP+ reduces the latency variation under all workloads. In other words, in terms of the improvement of latency variation, the benefit of reducing large latencies is higher than the drawback of reordering requests. The only exception is that HP slightly increases the latency variation under the Google Earth workload. In this case, a large number of requests are reordered, and the reordering negatively affects the latency variation.

3) Discussion on Deadline Selection: Deadlines is introduced to prevent request starvation under priority-based scheduling. However, the setting of the deadline introduces a tradeoff between average latency and worst-case latency. If the deadline is too large, some requests might be postponed for a very long time before they are dispatched. In this situation, the worst-case I/O latency would be increased. On the other hand, if the deadline is inadequately short, the opportunity to reorder I/O requests would diminish and the average I/O latency would increase.

Fig. 12 shows how the average and worst-case I/O latency are affected by different deadline settings for MAP+. As shown in Fig. 12(a), the average I/O latency is improved under all workloads as the deadline increases, because more requests can be reordered under a large deadline. In particular, under the Google Earth and Financial1 workloads, the latency is significantly improved when the deadline increases from 0 ms to 10 ms. As explained earlier, request scheduling is very beneficial to the average latency because the scheduler queue often has many pending requests under the two workloads. We observed that the latency time of most of the requests are between 0 ms and 5 ms. Therefore, for most workloads, the latency improvement saturates when the deadline is 50 ms.

Fig. 12(b) shows that the worst-case latency increases as the deadline is lengthened under most of the workloads (8 out of 9). When the deadline is larger than 10 ms, its impact on the worst-case latency varies from workload to workload: the worst-case latency of most workloads are barely affected, while that of some workloads, such as the Chrome, Google Earth, and Financial2 workloads, are noticeably increased. This is because the total number of prioritized requests increases rapidly as the deadline becomes larger, and the worst-case latency is negatively affected by these prioritized requests. The deadline has no impact on the worst-case latency of the proj_3 workload. This is because grouping requests into batches contributes to most of the performance improvement under the proj_3 workload, and the batch grouping operations are not affected by the deadline.

Based on the results, 10 ms appears to be a good balance between average latency improvement and worst-case latency degradation. Under the bursty workloads such as Google Earth, the deadline should never be larger than 20 ms to avoid excessive degradation in the worst-case latency.

4) Sensitive Studies: In this section, MAP+ is evaluated with different mapping cache sizes and I/O queue lengths to show the performance characteristics of MAP+. Fig. 13(a) shows the results for different mapping cache sizes. For most applications, the performance improvement quickly saturates as the cache size increases. For example, under Financial2, MAP+ achieves the best performance improvement with small increases in the mapping cache size, as shown in Fig. 9(c). In other words, MAP+ is able to achieve a considerable I/O performance improvement even if the cache is small. We believe that the effect of prioritizing cache-hit requests is similar to using a large mapping cache. Two exceptions are the Twitter and Google Earth workloads: there are slight performance degradations when the cache sizes are increased

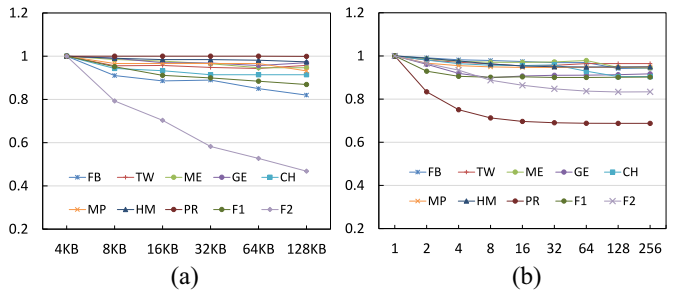


Fig. 13. Sensitive studies for mapping cache size, queue length. (a) Cache size. (b) Queue length.

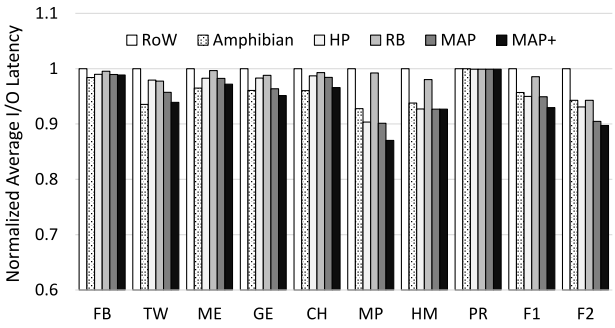


Fig. 14. Performance evaluation of different I/O schedulers on top of CDFTL.

from 64 KB to 128 KB. This is because, when the cache size is increased, some of the cache-hit requests are large, and prioritizing large requests may increase the waiting time of small pending requests [30].

Fig. 13(b) shows the results of different I/O queue lengths. The performance improves as the queue length increases, and the improvement also quickly saturates for most workloads when the length increases to 32. In other words, MAP+ is effective even with a small number of pending requests.

C. Performance Evaluation With CDFTL

There have been several proposals of efficient mapping cache designs. For example, CDFTL is an improvement upon DFTL with the exploitation of the flash page size and spatial localities in workloads [9]. We evaluated all schedulers on top of CDFTL. As shown in Fig. 14, when the underlying mapping cache was CDFTL, MAP+ reduced the I/O latencies by 13% and 11% under the Multiple and Financial1 workloads, respectively. Different from the results with DFTL, with CDFTL, HP performed better than RB under most of the workloads. This is because CDFTL itself is optimized for spatial localities, and HP further improved the hit ratio of the mapping cache by prioritizing hitting requests. Amphibian performed better with CDFTL than it did with DFTL, because the cache hit ratio under single-application workloads was very high (nearly 90%) and prioritizing small requests was beneficial to I/O latency reduction. MAP+ again outperformed RoW and Amphibian, especially under the multiapplication Multiple workload, because it achieved the highest mapping cache hit ratio (7% higher than that of RoW). In summary, MAP+ improves both spatial and temporal localities in the access pattern and is applicable to various mapping cache designs.

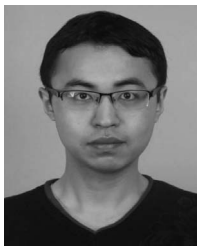
VII. Conclusion

This paper proposes an MAP approach for NAND flash-based storage systems with a small mapping cache. The proposed I/O scheduler reduces the impact of cache misses on I/O latencies based on two angles. First, it assigns high priorities to the I/O requests whose address translation can be resolved without any cache misses. Second, it batches requests that share the same translation page into a batch to reduce the overhead of translation page reads. Batches of requests are reordered to further optimize the request waiting time. Experimental results demonstrate that the proposed approach achieves significant improvements in terms of I/O latency, even when the mapping cache size is small. In the future, we plan to emulate embedded flash storage device which adopts demand-based mapping cache using Linux block device module and evaluate the efficacy of our proposed approaches in real arm-based embedded platforms.

References

[1] Y. Hu *et al.*, "Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation," in *Proc. MSST*, 2010, pp. 1–12.
[2] M. Son, J. Ahn, and S. Yoo, "A tiny-capacitor-backed non-volatile buffer to reduce storage writes in smartphones," in *Proc. IEEE CODES+ISSS*, Amsterdam, The Netherlands, 2015, pp. 21–29.
[3] C. Ji *et al.*, "An empirical study of file-system fragmentation in mobile storage systems," in *Proc. HotStorage*, 2016, pp. 1–5.
[4] J. Lee *et al.*, "TinyFTL: An FTL architecture for flash memory cards with scarce resources," in *Proc. APSYS*, 2013, p. 16.
[5] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. ASPLOS*, 2009, pp. 229–240.
[6] S. Lee *et al.*, "Application-managed flash," in *Proc. FAST*, Santa Clara, CA, USA, 2016, pp. 339–353.
[7] Y. Zhou *et al.*, "An efficient page-level FTL to optimize address translation in flash memory," in *Proc. EuroSys*, Bordeaux, France, 2015, p. 12.
[8] S. Jiang, L. Zhang, X. Yuan, H. Hu, and Y. Chen, "S-FTL: An efficient address translation for flash memory by exploiting spatial locality," in *Proc. MSST*, Denver, CO, USA, 2011, pp. 1–12.
[9] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems," in *Proc. RTAS*, Chicago, IL, USA, 2011, pp. 157–166.
[10] M. Jung, E. H. Wilson, III, and M. Kandemir, "Physically addressed queueing (PAQ): Improving parallelism in solid state disks," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 404–415, 2012.
[11] M. Dunn and A. L. N. Reddy, "A new I/O scheduler for solid state devices," Dept. Elect. Comput. Eng., Texas A&M Univ., College Station, TX, USA, Tech. Rep. TAMU-ECE-2009-02, 2009.
[12] J. Kim *et al.*, "Disk schedulers for solid state drivers," in *Proc. ACM EMSOFT*, Grenoble, France, 2009, pp. 295–304.
[13] M. Jung and M. Kandemir, "Revisiting widely held SSD expectations and rethinking system-level implications," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 203–216, 2013.
[14] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proc. USENIX Hotstorage*, Philadelphia, PA, USA, 2014, p. 13.
[15] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 4, pp. 837–863, 2004.

[16] Z. Qin, Y. Wang, D. Liu, Z. Shao, and Y. Guan, "MNFTL: An efficient flash translation layer for MLC NAND flash memory storage systems," in *Proc. ACM Design Autom. Conf.*, San Diego, CA, USA, 2011, pp. 17–22.
[17] N. Shahidi *et al.*, "Exploring the potentials of parallel garbage collection in SSDS for enterprise storage systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 561–572.
[18] X. Zhang, J. Li, H. Wang, K. Zhao, and T. Zhang, "Reducing solid-state storage device write stress through opportunistic in-place delta compression," in *Proc. FAST*, Santa Clara, CA, USA, 2016, pp. 111–124.
[19] N. Agrawal *et al.*, "Design tradeoffs for SSD performance," in *Proc. ATC*, Boston, MA, USA, 2008, pp. 57–70.
[20] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Trans. Consum. Electron.*, vol. 48, no. 2, pp. 366–375, May 2002.
[21] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *Proc. ACM EMSOFT*, Seoul, South Korea, 2006, pp. 161–170.
[22] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-aware sector translation for NAND flash memory-based storage systems," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, pp. 36–42, 2008.
[23] C.-H. Wu and T.-W. Kuo, "An adaptive two-level management for the flash translation layer in embedded systems," in *Proc. ICCAD*, San Jose, CA, USA, 2006, pp. 601–606.
[24] M.-C. Yang, Y.-H. Chang, T.-W. Kuo, and P.-C. Huang, "Capacity-independent address mapping for flash storage devices with explosively growing capacity," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 448–465, Feb. 2016.
[25] R. Chen *et al.*, "On-demand block-level address mapping in large-scale NAND flash storage systems," *IEEE Trans. Comput.*, vol. 64, no. 6, pp. 1729–1741, Jun. 2015.
[26] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *Proc. IEEE HPCA*, Orlando, FL, USA, 2014, pp. 524–535.
[27] S. Park and K. Shen, "FIOS: A fair, efficient flash I/O scheduler," in *Proc. FAST*, San Jose, CA, USA, 2012, p. 13.
[28] N. Elyasi *et al.*, "Exploiting intra-request slack to improve SSD performance," in *Proc. ACM 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2017, pp. 375–388.
[29] K. Shen and S. Park, "FlashFQ: A fair queueing I/O scheduler for flash-based SSDs," in *Proc. ATC*, San Jose, CA, USA, 2013, pp. 67–78.
[30] B. Mao and S. Wu, "Exploiting request characteristics and internal parallelism to improve SSD performance," in *Proc. ICCD*, New York, NY, USA, 2015, pp. 447–450.
[31] K. Kim, *Map Cache Design in Mobile Storage (SK Hynix)*, NVRAMOS, Jeju-do, South Korea, 2014. [Online]. Available: http://dcslab.hanyang.ac.kr/nvramos14/presentation/s9.pdf
[32] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh, "Parameter-aware I/O management for solid state disks (SSDs)," *IEEE Trans. Comput.*, vol. 61, no. 5, pp. 636–649, May 2012.
[33] P. Desnoyers, "What systems researchers need to know about NAND flash," in *Proc. HotStorage*, San Jose, CA, USA, 2013, p. 6.
[34] Y. Lee, J.-S. Kim, S.-W. Lee, and S. Maeng, "Zombie chasing: Efficient flash management considering dirty data in the buffer cache," *IEEE Trans. Comput.*, vol. 64, no. 2, pp. 569–581, Feb. 2015.
[35] J. Ouyang *et al.*, "SDF: Software-defined flash for Web-scale Internet storage systems," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 471–484, 2014.
[36] *Packed Commands*, JEDEC Standard eMMC 5.0, 2012. [Online]. https://www.jedec.org/sites/default/files/docs/JESD84-B50.pdf
[37] *Data Tag Mechanism*, JEDEC Standard eMMC 5.0, 2012. [Online]. https://www.jedec.org/sites/default/files/docs/JESD84-B50.pdf
[38] *32/64/128/256Gb Async/Sync NAND, (M73A)*, Micron Datasheet, Boise, ID, USA, 2016.
[39] Y. Hu *et al.*, "Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1141–1155, Jun. 2013.
[40] (2009). *UMass Trace Repository*. [Online]. Available: http://traces.cs.umass.edu/index.php/storage
[41] SNIA. (2009). *IOTTA Repository*. [Online]. Available: http://iotta.snia.org/



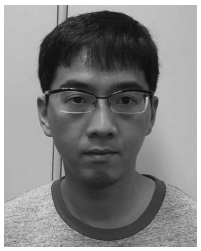
Cheng Ji received the B.E. degree from the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China, in 2011 and the M.E. degree from the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China, in 2014. He is currently pursuing the Ph.D. degree with the Department of Computer Science, City University of Hong Kong, Hong Kong.

His current research interests include embedded systems, nonvolatile memory, and architecture optimizations.

Li-Pin Chang received the M.S.E. and Ph.D. degrees in computer science and information engineering from National Taiwan University, Taipei, Taiwan, in 1997 and 2003, respectively.

He serves as an Associate Professor with the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan. He was a Fulbright Visiting Scholar with the University of Minnesota, Minneapolis, MN, USA, from 2015 to 2016. His current research interests include operating systems, storage systems, nonvolatile memory, and mobile devices.

Dr. Chang served as the Program Co-Chair, the Track Program Co-Chair, and the General Co-Chair of many conferences in research areas, including the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications and ACM Symposium on Applied Computing. He also served as a Guest Editor for *ACM Transactions on Embedded Computing Systems*, *ACM Transactions on Cyber-Physical Systems*, and Elsevier *Journal of System Architecture*.

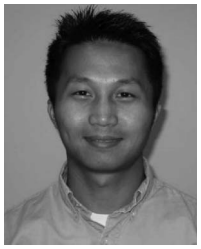


Chao Wu received the B.E. degree in automation from the Taiyuan University of Technology, Taiyuan, China, in 2007. He is currently pursuing the Ph.D. degree with the Department of Computer Science, City University of Hong Kong, Hong Kong.

His current research interests include flash memory, architecture optimizations, and mobile systems.

Liang Shi received the B.S. degree in computer science from the Xi'an University of Post and Telecommunication, Xi'an, China, in 2008 and the Ph.D. degree from the University of Science and Technology of China, Hefei, China, in 2013.

He is currently an Associate Professor with the College of Computer Science, Chongqing University, Chongqing, China. His current research interests include flash memory, embedded systems, and emerging nonvolatile memory technology.



Chun Jason Xue received the B.S. degree in computer science and engineering from the University of Texas at Arlington, Arlington, TX, USA, in 1997 and the M.S. and Ph.D. degrees in computer science from the University of Texas at Dallas, Richardson, TX, USA, in 2002 and 2007, respectively.

He is currently an Assistant Professor with the Department of Computer Science, City University of Hong Kong, Hong Kong. His current research interests include memory and parallelism optimization for embedded systems, software/hardware co-design, real time systems, and computer security.