SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2015; **45**:1229–1249
Published online 11 June 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2279

Profit data caching and hybrid disk-aware Completely Fair Queuing scheduling algorithms for hybrid disks

Hsung-Pin Chang[1,*,†], Syuan-You Liao[1], Da-Wei Chang[2] and Guo-Wei Chen[1]

[1]*Department of Compute Science and Engineering, National Chung Hsing University, Taichung, Taiwan*
[2]*Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan*

SUMMARY

Recently, a hybrid disk drive that integrates a small amount of flash memory within a mechanical drive has received significant attention. The hybrid drive extends the storage hierarchy by using flash memory to cache data from the mechanical disk. Unfortunately, current caching architectures fail to fully exploit the potential of the hybrid drive. Furthermore, current disk input/output (I/O) schedulers are optimized for rotational mechanical disk drives and thus must be re-targeted for the hybrid disk drive. In this paper, we propose a new data caching scheme, called Profit Caching, for hybrid drives. Profit Caching is a self-optimizing caching algorithm. It considers and seamlessly integrates all possible data characteristics that impact the performance of hybrid drives, including read count, write count, sequentiality, randomness, and recency, to determine the caching policy. Moreover, we propose a *h*ybrid disk-*a*ware Completely Fair Queuing (HA-CFQ) scheduler to avoid unnecessary I/O anticipations of the CFQ scheduler. We have implemented Profit Caching and HA-CFQ scheduler in the Linux kernel. Coupled with a trace-driven simulator, we have also conducted detailed experiments under a variety of workloads. Experimental results show that Profit Caching provides significantly improved performance compared with the previous schemes. In particular, the throughput of Profit Caching outperforms previous Random Access First and FlashCache caching schemes by factors of up to 1.8 and 7.6, respectively. In addition, the HA-CFQ scheduler reduces the total execution time of the CFQ scheduler by up to 1.74%. Finally, the experimental results show that the runtime overhead of Profit Caching is extremely insignificant and can be ignored. Copyright © 2014 John Wiley & Sons, Ltd.

Received 9 August 2013; Revised 12 May 2014; Accepted 12 May 2014

KEY WORDS: disk cache; hybrid disks; heterogeneous storage; disk scheduler; flash memory

1. INTRODUCTION

In the last decade, advances in semiconductor technology have dramatically increased processor speeds, with speeds expected to continue to double every year. However, the mechanical operations of hard disks rule out similarly dramatic reductions, and the performance gap between processors and mechanical disks will continue to increase. This problem is getting more serious because the requests of applications are interleaved, imparting more randomness and increasing head positioning time. As a result, the computer system's performance is increasingly limited by the storage subsystem.

Recently, the emergence of NAND flash-based solid state drives (SSDs) has provided a potential solution to this issue. As compared with mechanical disks, SSDs have no mechanical moving parts and thus boast many superior features such as silent, lightweight, less power consumption, and shock resistant. However, SSDs have lifetime constraints because each flash memory block can only endure a limited number of erase operations before becoming unreliable. Besides, SSDs are

*Correspondence to: Hsung-Pin Chang, Department of Compute Science and Engineering, National Chung Hsing University, 250 Kun-Kuang Road, Taichung 402, Taiwan.
†E-mail: hpchang@cs.nchu.edu.tw

considerably more expensive than mechanical disks. These factors have impeded the widespread use of SSDs as the primary storage media for most, if not all, commercial and daily operating systems.

Because of the high cost and limited lifespan of SSDs, researchers have thus proposed to extend the storage hierarchy to include flash memory as an intermediate tier between main memory and mechanical disks, that is, utilizing SSDs as the cache for mechanical disks. This can generally be accomplished through two different architectures: a hybrid drive that collocates the two storage media in the same enclosure [1] or a hybrid storage architecture that combines SSDs and mechanical drives to form a heterogeneous storage system [2–4]. For the sake of simplicity, this study focuses on the hybrid disk drive.

However, the design of an SSD caching scheme is completely different from the main memory caching because of the unique characteristics of SSDs. Compared with the mechanical disks, flash memory provides better performance, especially for random read workloads, but the sequential access of mechanical disks is comparable with that of flash memory. For example, on an Intel® X25-E SSD, random read operations are up to 7.7 times faster than on a hard disk drive (HDD), but the advantage for sequential reads is only about two times [5]. In addition, depending on the choice of SSDs and mechanical disks, the sequential write performance of flash memory can be sometimes worse than that of mechanical disks [6]. Finally, as stated earlier, flash memory has significant lifetime constraint. Therefore, a well-optimized data caching scheme should exploit the complementary properties of SSDs and HDDs and seamlessly bridge the two storage media such that flash memory can make up for the disadvantages of mechanical disks, and vice versa.

Unfortunately, current caching schemes fail to fully exploit the potentials of such hybrid architecture, because they only consider a subset of data characteristics that are correlated to caching performance. For example, Random Access First (RAF) considers only the data's sequentiality and frequency [3]. Nevertheless, previous studies of buffer cache replacement algorithm have shown that recency is also an important performance factor for a caching algorithm [7]. In addition, read and write operations in flash memory entail different costs, with read operations running several times faster. In addition, a non-empty page must be erased before new data can be written to it. Previous studies have largely ignored the asymmetric characteristics of read and write operations.

Furthermore, current input/output (I/O) scheduling algorithms are optimized for mechanical disks. Thus, they addresses the impact of seek time and rotational latency in scheduling I/O requests. Flash memory is a semiconductor-based device and has no mechanical moving parts, so current I/O schedulers are poorly suited and should be revised for use in hybrid disk drives.

This paper addresses the aforementioned issues and presents the design and implementation of hybrid drive architecture. The architecture consists of two major parts: a novel data caching and management scheme, called Profit Caching, and a hybrid disk-aware Completely Fair Queuing scheduler, called HA-CFQ. Our design seeks to exploit the complementary performance properties of these two storage media. In addition, the implementation adheres to the module design principle that minimizes system changes. In summary, our proposed hybrid disk architecture makes the following contributions.

- To identify performance-important and endurance-critical blocks, Profit Caching distinguishes and exploits all possible metrics and seamlessly encodes them by defining the profit for data. Profit Caching also uses efficient data structures to efficiently manage data metrics and calculate the profit value.
- The current Linux default CFQ scheduler incorporates an I/O anticipation scheme to solve the problem of deceptive idleness, so as to avoid unnecessary movement of the disk head [8]. However, flash memory is a semiconductor-based device, so the cost of idling far exceeds the limited benefit of I/O spatial proximity. Through awareness of the discrepancy between flash and rotating media, the proposed HA-CFQ scheduler identifies and eliminates unnecessary I/O anticipations.
- We present a comprehensive design and implementation of our hybrid drive architecture. In addition, we exploit the module design discipline that encapsulates almost all of the complicated details within the device driver. Consequently, the OS kernel and applications are left completely intact. This guarantees compatibility and portability, which are both critical in practice systems [5].

DOI: 10.1002/spe

The remainder of this paper is organized as follows. Section 2 reviews the characteristics of flash memory, along with previous work on heterogeneous storages and I/O scheduling algorithms. Sections 3 and 4 respectively present the proposed Profit Caching and HA-CFQ scheduler schemes. The implementation details are shown in Section 5. Finally, the experimental results and concluding remarks are respectively given in Sections 6 and 7.

2. BACKROUND AND RELATED WORK

In this section, Section 2.1 introduces flash memory. Section 2.2 then reviews the literatures on heterogeneous storage and hybrid disk drives. Finally, Section 2.3 presents the I/O scheduling algorithm in the current Linux kernel.

2.1. Flash memory characteristics

A NAND flash memory module is composed of a number of blocks, and each block consists of a number of pages. Flash memory exposes three types of operations: read, write, and erase. Reads and writes are page oriented. However, flash memory does not support in-place update. A write to a non-empty page must be erased first before a write operation can commence, and the erase is conducted in units of blocks. Compared with read/write, the erase operation is extremely time consuming. Moreover, each block can only endure a limited number of erase operations, usually ranging from 10K to 100K, before becoming unreliable. Thus, it should be avoided to erase an entire block each time when a page is overwritten. Usually, this is accomplished by directing each page overwrite to a free physical page. Then the original page containing the stale data is marked invalid and reclaimed later by a *garbage collection* procedure. To further prolong the flash memory lifespan, the erase and overwrite operations should be evenly distributed among all the blocks in the flash memory. In this way, no single block fails soon because of a high concentration of overwrite and erase operations. This technique is called *wear leveling* or *even wearing* [9].

Table I respectively shows the specifications of single-level cell (SLC) and multi-level cell (MLC) based Samsung Electronics K9WAG08U1M NAND flash memory, respectively [10]. There are two primary types of NAND flash memory: SLC and MLC. The SLC memory stores one bit per cell, whereas the MLC memory stores multiple bits per cell. As shown in Table I, for SLC, an erase operation is six times slower than a write operation, which, in turn, is three times slower than a read operation. In addition, the endurance level of SLC flash is about 100,000 cycles, whereas an MLC block can only endure about 10,000 erase operations.

2.2. Related work on hybrid storage

In the past few years, many proposals have been made to combine NAND flash memory and mechanical disks. These researches differ in their approaches to physically integrating flash memory and to logically managing flash memory in the hierarchy of the storage stack. In terms of physical integration, there are two key architectures, *on-disk* and *on-board*. The on-disk method embeds a small capacity of flash memory in a mechanical disk, forming a hybrid drive. For example, in May 2010, Seagate launched the Momentus XT 7200 RPM 2.5-in. solid state hybrid drive [1]. The on-board approach, by contrast, plugs both the SSD and the mechanical disk into the I/O bus. For example, in October 2012, Apple Inc. launched a fusion drive that integrates stand-alone SSDs with a mechanical drive [11].

In addition to the hardware integration method, two widely used software management methods are used for such heterogeneous storages [12]. One utilizes SSDs as the cache for mechanical disks, and the other treats SSDs and mechanical disks as peers. Usually, the on-disk hardware integration scheme treats SSDs as the cache for mechanical disks, whereas the on-board method might manage

Table I. Specification of NAND flash memory (Samsung Electronics K9WAG08U1M).

Flash type	Page size (KB)	Block size (KB)	Read (μs)	Write (μs)	Erase (μs)	Endurance (erase cycles)
SLC	2	128	72.8	252.8	1500	100,000
MLC	4	512	165.6	905.6	1500	10,000

DOI: 10.1002/spe

SSDs as a cache or peer for the mechanical disks. The following two subsections introduce previous work related to these two software management schemes.

2.2.1. Peer-based heterogeneous storage systems. Peer-based heterogeneous storage systems treat SSDs as peers for mechanical disks; thus, the resulting storage space combines the capacities of SSDs and HDDs. Exploiting the superior performance of SSDs requires migrating data dynamically and asynchronously between SSDs and HDDs. For example, hot data are migrated to SSDs, whereas cold data are left in mechanical disks, so as to maximize the aggregated response time of the heterogeneous storage system. Usually, the migration unit can be a block or a file, depending on the software management module used at the device level. These two methods are discussed in the following.

Management at the file level: File-level approaches take file-level knowledge or statistics into account to manage the storage space across the heterogeneous devices. For example, pattern-based popular data concentration proposes an energy-efficient hybrid storage solution by properly migrating files between SSDs and mechanical disks [13]. It monitors the access type and frequency of each file and places frequently read files on SSDs and frequently written files on mechanical disks, so as to separate read and write I/O requests. Hot random off-loading (HRO) extends this concept by considering not only the access frequency of files but also their random-access characteristics [14]. Specifically, it monitors the hotness and randomness of each individual file. HRO then applies the 0/1 knapsack model to migrate or allocate files between the mechanical disks and the SSDs. Similarly, Combo Drive utilizes not only file access behavior but also file type information and then migrates executable files, program libraries, and randomly or frequently accessed files to the SSDs [15].

Management at the block level: Device-level-based approaches manage heterogeneous storage space at a sector, block, or chunk level [16]. For example, in [6], the authors proposed a performance-driven approach that dynamically migrates data blocks between mechanical disks and SSDs to equalize the response times of the different storage devices. Similarly, Hystor monitors the request size and frequency of each block to identify performance-critical blocks [5]. Hystor also extracts the file-system information to identify semantically critical blocks, for example, metadata blocks. These performance-critical or semantically critical blocks are then stored in SSDs. Instead of addressing the performance issue, energy-efficient disk focuses on the energy saving of magnetic disks; it monitors the data access patterns and moves hot data from the magnetic disk to the flash memory when the disk is idle, so that the disk drive can stay in the low power state to conserve energy consumption [17].

2.2.2. Caching-based heterogeneous storage systems. To exploit SSDs to improve disk I/O performance, another solution is to extend the memory hierarchy using SSDs as cache for mechanical disks. Unlike in heterogeneous approaches, because of the storage hierarchy, the SSD cache resides below the file-system level and thus is managed only at the block level. Consequently, in addition to the page cache, SSDs act as a *block-based* second-level cache for mechanical disks. For example, FlashCache treats SSDs as a write cache and uses SSDs to absorb the workload of the disk storage tier [2]. In addition, FlashCache proposes a wear-level aware replacement policy that selects a victim block. Similar to the FlashCache, PASS redirects all of the I/O requests to the SSDs [18]. However, instead of a write-back cache, PASS writes the updated block to the HDD asynchronously.

Random Access First improves on FlashCache by differentiating random access from sequential access and only caches random-access data. In addition, RAF partitions the SSDs into read and write caches to separately service read and write requests. The read cache is managed by a Least Frequently Used cache replacement policy, and the write cache performs as a circular write-through log. Because random write severely degrades the lifetime of flash memory, RAF spreads the wear evenly across the SSDs by periodically moving the location of the write cache.

Unfortunately, current approaches fail to fully exploit the potential of SSDs, leading to poor performance in practice. For example, FlashCache does not identify blocks that are valuable for

DOI: 10.1002/spe

caching in SSDs, resulting in SSDs absorbing too much workload of disk tier, and heavy workloads can accelerate wear. In addition, this approach wastes the potential bandwidth of hard disks, resulting in performance degradation. Similarly, RAF considers only the sequentiality and frequency of data access patterns and ignores the recency metric and the asymmetric costs of read and write operations. As a result, random-accessed data that are written repeatedly would be cached in SSDs by RAF, resulting in excess wear and decreasing the SSD lifetime. Similarly, sequential-accessed data that are often read would not be cached in SSDs, failing to fully utilize the superior performance of SSDs. Furthermore, the logical address sent by the host is not the actual physical address sent to the flash memory. Thus, a wear-leveling algorithm is usually implemented in the SSD and is integrated with the design of the flash translation layer. Consequently, periodically moving the 'logical' location of the write cache to evenly distribute wear is useless.

This paper addresses the use of SSDs as a second-level mechanical disk cache using the on-disk hardware integration method. Because both architectures have their respective advantages and disadvantages, it is not our intention to determine the superiority of one over the other [16]. Both approaches are based on different assumptions and result in different design points and benefits. In addition, although this paper focuses on the hybrid disk drive, the data management scheme can also be applied to the hybrid storage architecture.

2.3. CFQ scheduler

The current Linux kernel supports four disk I/O schedulers, that is, noop scheduler, deadline scheduler, anticipatory scheduler, and CFQ. From kernel 2.6.18, Linux adopted the CFQ as the default scheduler. Thus, in this subsection, we introduce the CFQ scheduler.

The CFQ scheduler aims to share disk I/O bandwidth equally among all of the competing processes. To realize this, it provides a separate queue for each process, assigns each queue a time slice, and visits each queue in a round-robin fashion. In this manner, CFQ then distributes the available I/O bandwidth equally among all competing processes. Like other schedulers, CFQ also strives to minimize seeking time. It exploits the rotational nature of the mechanical disks and serves requests of the same queue following the one-way elevator algorithm, so as to minimize disk head travel. Finally, it uses the concept of an anticipatory scheduler to solve the 'deceptive idleness' problem [8]. Usually, applications issue read requests in a *synchronous* manner, that is, by issuing successive read requests with short periods of computation. Thus, a process can issue a new read request only after its previous read request has finished. As a result, after serving a process's read request, a work-conserving scheduler may serve another request issued by some other process, causing the disk head move to a different portion of the disk that may be far from its current position. Then, when the next request from the original process occurs, the disk head must again move back, resulting in excessive seeking overhead. To solve this deceptive idleness problem, the anticipatory scheduler pauses briefly after a read request in anticipation of a soon-arriving desirable request. In contrast to the anticipatory scheduler that pauses for each read request, CFQ pauses only when the current request queue is empty.

Flash memory is an electronic device with no mechanical moving parts. Thus, the performance benefits of I/O anticipation are negative because the cost of device idling far exceeds the extremely limited benefit of I/O spatial proximity [19]. Therefore, existing I/O schedulers that have been specifically optimized for rotational mechanical HDDs must be redesigned for the hybrid disks.

3. PROFIT CACHING: IDENTIFYING HIGH-PROFIT BLOCKS

3.1. Problem definition

Many workloads show that a small data set contributes a large percentage of aggregate data accesses. Thus, to improve the performance of hybrid disks or heterogeneous storage systems,

DOI: 10.1002/spe

we should identify the most appropriate blocks from the data set and cache these blocks in the flash memory. We formally formulate the problem as follows.

Assume that there are n data blocks in the mechanical disk $\boldsymbol{B} = \{b_1, b_2, \ldots, b_n\}$ and the i-th block b_i is denoted by two parameters $\{s_i, v_i\}$, where s_i is the block size and v_i is the benefit value of the block. Then, we must find m block $\boldsymbol{C} = \{b_{Z(1)}, b_{Z(2)}, \ldots, b_{Z(m)}\}$ that resolves $\mathbf{max}_{\forall Z} \left\{\sum_{i=1}^{m} v_Z(i)\right\}$ under the constraints that $\sum_{i=1}^{m} s_{Z(i)} \leq CAPACITY_{flash_{memory}}$ and cache these m blocks in the flash memory. In other words, the objective of the problem is to find and cache a combination of disk blocks in flash memory that maximize the total benefit value subject to the flash memory capacity constraint.

Nevertheless, this problem is similar to the classic 0/1 knapsack problem, which in turn is NP-complete [14]. Thus, we develop an approximation algorithm called Profit Caching, which heuristically calculates the benefit value of each block and selects only the high-benefit value blocks to cache in flash memory.

3.2. Determining the performance and endurance-critical metrics

To determine the benefit value of a block, we should identify a set of metrics and associate each block with the selected metrics. Notably, the benefit value is used to determine whether a block should be cached in the flash memory or not. Consequently, the metrics selected should favor blocks that should exploit the complementary properties of flash memory and mechanical disks.

Mechanical disks suffer from random accesses, whereas flash memory has no mechanical moving parts. Thus, randomly accessed blocks should be cached in flash memory, whereas sequentially accessed blocks should be left and served from the mechanical disks. Consequently, the first selected metric is the degree of randomness of a block. Second, write operations are usually asynchronous and can tolerate a larger delay than read operations. Besides, in flash memory, the read operations are faster than write operations. Write operations may also trigger erase operations, which not only is time consuming but also has a negative impact on the lifetime of flash memory, and this problem is exacerbated by the intensity of traffic to caching space. Therefore, we should cache the read-dominated blocks in flash memory and leave write-intensive blocks to mechanical disks. Thus, the second and third metrics are the degree of read-intensiveness and write-intensiveness of a block. Finally, because most workloads exploit the temporary locality, we should take recency into consideration. Consequently, the selected metrics are the degree of randomness/sequentiality, read-intensiveness, write-intensiveness, and recency.

Next, we need to determine how to measure these metrics for a block. First, we use the number of read and write operations of a block to respectively represent the block's degrees of read-intensiveness and write-intensiveness. Second, we use the sequence length to describe the degree of randomness/sequentiality. A sequence is a number of blocks whose logical block addresses (LBAs) are adjacent [3]. Thus, blocks belonging to a larger sequence are likely to be accessed sequentially. In contrast, short sequences are more vulnerable to interference by other sequences and thus are prone to random access. Finally, we express the degree of recency, which is then seamlessly integrated with other metrics by the GreedyDual algorithm [20, 21]. The following subsection describes the algorithm in detail.

3.3. Profit Caching

After determining the effective metrics, we must properly and seamlessly encode the selected metrics of a block to calculate the block's benefit value. However, to associate each block with a runtime object to maintain its metrics and benefit value requires maintaining a significant number of runtime objects, incurring high space and runtime overhead, especially when the size of the flash memory is large. Because we use the sequence to represent the degree of randomness, we represent the runtime objects associated with each sequence to reduce the bookkeeping overhead. Thus, our caching policy is a variable-size caching scheme because each sequence may have a different length.

Finally, we develop the following effective equation to calculate the benefit value, called the *profit*, of the i-th sequence.

DOI: 10.1002/spe

$$Profit_i = \alpha \times \frac{readCnt_i + 1}{writeCnt_i \times seqLen_i + 1} + (1 - \alpha) \times Base\ Profit, 0 \le \alpha \le 1 \quad (1)$$

where $readCnt_i$ and $writeCnt_i$ respectively denote the number of read and write operations of the i-th sequence and *seqLen* is the sequence length that is measured in terms of the number of blocks. Finally, *BaseProfit*, which is used to encode the degree of recency, represents the profit value of the last evicted sequence from flash memory to mechanical disk and thus is updated each time when a sequence is evicted from flash memory. Notably, when a cache replacement occurs, we select the sequence with the smallest profit value for eviction. Consequently, the value of *BaseProfit* is increased monotonically. Therefore, a more recent sequence will have a larger value of *BaseProfit*.

From Equation (1), *readCnt* divided by *writeCnt* derives the read benefit of writing a sequence to flash memory. Then, *readCnt* divided by the product of *writeCnt* and *seqLen* represents the read benefit of writing a block of a sequence to flash memory. Therefore, blocks of a random sequence would derive a larger benefit than blocks of a sequential one, because a random sequence has a shorter sequence length. Finally, we consider the recency by letting the profit value of new sequences be offset by the value of *BaseProfit*. Thus, the definition of profit seamlessly encodes all of the performance indictor metrics and successfully exploits the complementary properties of flash memory and mechanical disks. Profit Caching uses flash memory as a bypassable cache for the mechanical disk, and data with smaller profit value would not be cached in flash memory.

4. HYBRID DISK-AWARE CFQ SCHEDULER

4.1. HA-CFQ scheduler

As stated in Section 2.3, when the current request queue is empty, the CFQ scheduler sits idle for a brief period (called *anticipation time)*, anticipating a nearby request from the same processes. Flash memory has no mechanical moving parts; thus, accessing the embedded flash memory of a hybrid disk does not cause the disk head to move. Unfortunately, in the Linux kernel, a hybrid disk is considered to be the same as a mechanical disk. But in a hybrid disk, some requests are served by flash memory, whereas others are served by magnetic disks. Consequently, based on the CFQ scheduler, we propose a HA-CFQ scheduler for use in hybrid disk drives.

As shown in Figure 1, when the current request queue is empty, HA-CFQ looks up the first outstanding request in the next request queue. If the request accesses the mechanical disk of the hybrid disk, HA-CFQ behaves like the CFQ scheduler that sits idle in anticipation of the next close-by request. However, if the request accesses the flash memory of the hybrid disk, HA-CFQ immediately serves the request, instead of pausing for the anticipation time period. This is because, as stated earlier, accessing flash memory does not cause the disk head of the mechanical disk to move. After serving the flash memory request, HA-CFQ looks up the next outstanding request and immediately serves it if it also accesses the flash memory. This process is repeated until a close-by request is finally issued within the anticipation time period, or HA-CFQ encounters an outstanding request accessing the hybrid disk's mechanical disk. In the latter case, if the accumulated service time of these flash memory requests (say T_f) exceeds the default value of anticipatory time (say T_d), HA-CFQ continues serving the mechanical disk request. Otherwise, it pauses for the remaining time, $T_f - T_d$,

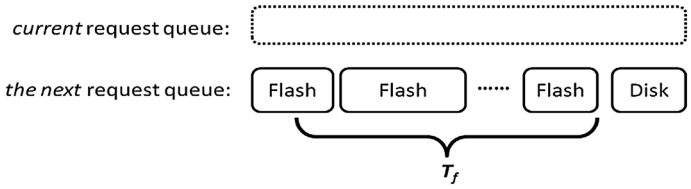


Figure 1. The diagram demonstrating the HA-CFQ scheduler.

DOI: 10.1002/spe

and continues to anticipate the next close-by request. Based on the aforementioned discussions, the HA-CFQ scheduler thus can identify and eliminate unnecessary I/O anticipations.

4.2. Analytical analysis

This subsection presents the analytical analysis of time saved by HA-CFQ. First, we calculate the actual time that the CFQ waits. After processing the last request of the current request queue, the CFQ waits to anticipate a close-by request. Assume that the time the scheduler uses to handle the last request is T_c and the arriving time of the next close-by request is T_n. Then, the inter-arrival time of these two requests, T_y, can be calculated as $T_y = T_n - T_c$. Assume that the anticipation time is T_x. Thus, the actual time interval T_w that the CFQ waits is calculated by Equation (2).

$$T_w = \min(Tx, Ty) \tag{2}$$

We now discuss the waiting time of HA-CFQ for I/O anticipation. After the scheduler processes the last request of the current request queue, if the first outstanding request of the next request queue is a flash memory request, HA-CFQ immediately serves the request along with any outstanding flash memory requests, until a close-by request is finally issued within the anticipation time, or an outstanding disk request is encountered. As shown in Figure 1, assume that the accumulated service time of these flash requests is T_f. Then, the waiting time of HA-CFQ for I/O anticipation, say $T_w^{'}$, depends of the values of T_f and T_w. If $T_f < T_w$, after serving the flash requests of the next queue, HA-CFQ still needs to wait for a period of $(T_w - T_f)$, that is, $T_w^{'} = \left(T_w - T_f\right)$. In contrast, if no close-by request is issued within the anticipation time, that is, if $T_f \geq T_w$, HA-CFQ does not need to continue waiting after it serves the flash requests of the next queue, that is, $T_w^{'} = 0$.

From the aforementioned discussion, the time savings by HA-CFQ (say T_s) is thus calculated as $T_s = T_w - T_w^{'}$. If $T_f < T_w$, the time savings by HA-CFQ $T_s = T_w - (T_w - T_f) = T_f$. In contrast, if $T_f \geq T_w$, the time savings by HA-CFQ is $T_s = T_w$. In summary, the time savings by HA-CFQ, denoted as T_s, can be calculated by Equation (3).

$$T_s = \begin{cases} \min\left(T_x, T_y\right), & \text{if } T_f \geq T_w \\ T_f, \text{otherwise} \end{cases} \tag{3}$$

5. DESIGN AND IMPLEMENTATION

5.1. Architecture

To demonstrate the effectiveness of Profit Caching and the HA-CFQ scheduler, we implemented a prototype system in the Linux kernel. Figure 2 shows the system architecture. First, as shown by the direction of the bold arrows in Figure 2, we only cache data sent to the mechanical disks and do not cache data read from the mechanical disks. This is because, similar to RAF [3], we implement the *lazy caching* scheme; data are not immediately cached when it is read from the mechanical disks, even if the corresponding profit value is high. This is because, if we cache data of read requests, the copy cached in the flash memory would become obsolete whenever the data cached in the page cache are modified. As a result, we must either synchronously or asynchronously update the cached copy in the flash memory. Thus, we cache data when the data are later evicted from the page cache, if its profit value is sufficiently high.

Second, the design philosophy behind the architecture aims to avoid undesirable significant changes to the existing kernel and applications while effectively accomplishing the management of hybrid disks. Thus, as shown in Figure 2, we adopt a modular design approach. Except for the interception of the *clean* pages evicted from the page cache, almost all modifications, including the implementations of Profit Caching and HA-CFQ, are encapsulated in the device driver of the hybrid disks. Consequently, the implementation is transparent to the software above the device drivers and is therefore compatible and portable, which are both important characteristics for real system implementations.

DOI: 10.1002/spe

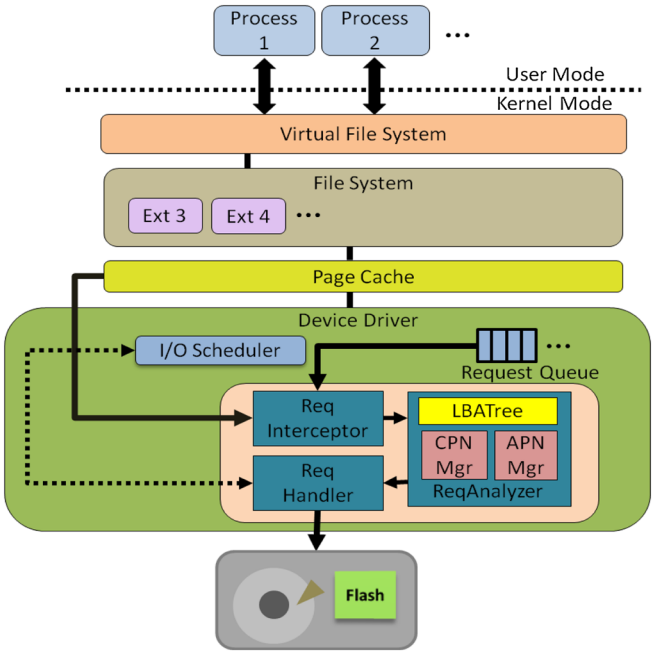


Figure 2. The system architecture.

Finally, as shown in Figure 2, the implementation of Profit Caching and HA-CFQ consists of three submodules: *ReqInteceptor*, *ReqAnalyzer*, and *ReqHandler*. ReqInteceptor intercepts disk I/O requests and *clean* victim pages evicted from the buffer cache and synchronously redirects the requests to the ReqAnalyzer module. ReqAnalyzer implements the Profit Caching. After receiving the requests sent from the ReqInterceptor, ReqAnalyzer maintains the metadata of these requests, calculates their profit values, and decides whether to cache the requested data using the Profit Caching policy. Finally, it sends the decision to the ReqHandler, which schedules the requests following the HA-CFQ scheduling algorithm. The following subsections describe the operations of these three components in detail.

5.2. *ReqInterceptor component*

The ReqInterceptor module intercepts all of the write requests sent to the mechanical disks. Furthermore, as shown in Figure 2, it interfaces with the page cache to intercept *clean* pages evicted from the page cache. This is because, in the current Linux kernel, *clean* pages selected as victims are simply discarded. However, as stated in Section 5.1, we implement the *lazy caching* scheme. Therefore, ReqInterceptor must intercept these evicted *clean* pages, which otherwise may not be cached in the flash memory, even given a large profit value. In contrast, the ReqInterceptor does not intercept *dirty* pages evicted from the buffer cache because, if the victim page is dirty, the kernel would issue explicit write requests to the driver, and these requests would be intercepted by the ReqInterceptor. Because the interception was performed at the bottom of the storage hierarchy, the caching effects in the kernel have been filtered out. Thus, the collected access statistics reflect the real storage workload [12].

Because the ReqInterceptor is implemented within the device driver, Profit Caching is a block-level caching system. However, the data evicted from the page cache follow the page-level abstraction. Thus, we must transfer the page-level abstraction of the evicted clean pages to the block-level one. As shown in Figure 3, the Linux kernel maintains the status of each page frame using a *page descriptor* structure. In addition, if a page frame is allocated in the page cache, each buffer of the page frame is described by a *buffer_head* data structure, which maintains the block number of the corresponding buffer, and all *buffer_head* descriptors associated with the same page frame are collected by the *b_this_page* field in a singly linked circular list. In addition, the *private* field

DOI: 10.1002/spe

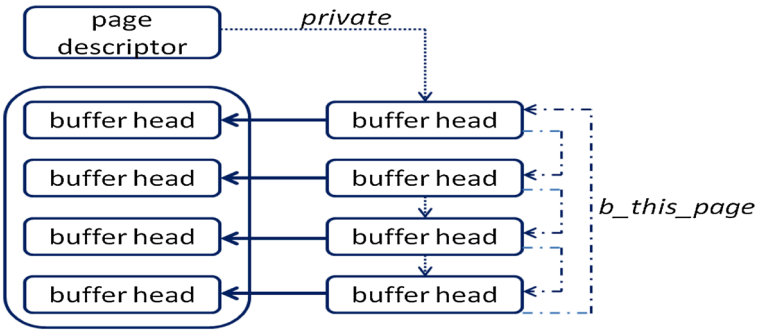


Figure 3. A buffer page including four buffers.

of the *page descriptor* points to the *buffer_head* of the first buffer in the page frame. Therefore, by the data structure shown in Figure 3, we can derive the LBAs of buffers that belong to the same page frame. Notably, in the Linux kernel, the size of a buffer is set equal to the size of the block of the mechanical disk, which is determined when the corresponding disk is formatted. In Figure 3, assume that the page size is 4 KB and the block size is formatted as 1 KB; thus, a page frame consists of four buffers.

5.3. *ReqAnalyzer component*

ReqAnalyzer implements the Profit Caching algorithm. Upon receiving an I/O request from ReqInterceptor, ReqAnalyzer first searches the associated runtime object by the accessed logical block numbers of the request. Therefore, ReqAnalyzer maintains a data structure, called *LBA_Tree*, which is indexed and searched by a logical block number. ReqAnalyzer implements the *LBA_Tree* by a red-black tree. In addition, because we associate the runtime object with a sequence, each node of the tree thus maintains the metadata of a sequence. The red-black tree is a self-balancing binary tree with a good worst-case running time, and node search, insertion, and deletion can be accomplished in O(log$n$) time, where n is the number of elements in the tree. Figure 4 shows an example of the *LBA_Tree*, where the *Beg_lba* and *Len* fields respectively denote the beginning LBA and the length of a sequence.

However, the requested data of a sequence might be cached in flash memory or still reside in mechanical disks. Therefore, each node in the *LBA_Tree* takes two forms: Approval Profit Node (APN) if the data of a sequence is cached in flash memory or Candidate Profit Node (CPN) if it is cached in the mechanical disk. Furthermore, ReqAnalyzer contains two subcomponents, *CPNMgr* and *APNMgr*, which respectively maintain the APN and CPN.

Consequently, upon receiving a request from the ReqInterceptor, ReqAnalyzer searches the *LBA_Tree* and partitions the sequence into a number of subsequences depending on whether the requested data are cached in flash memory or not. For example, in Figure 5, a request is partitioned into

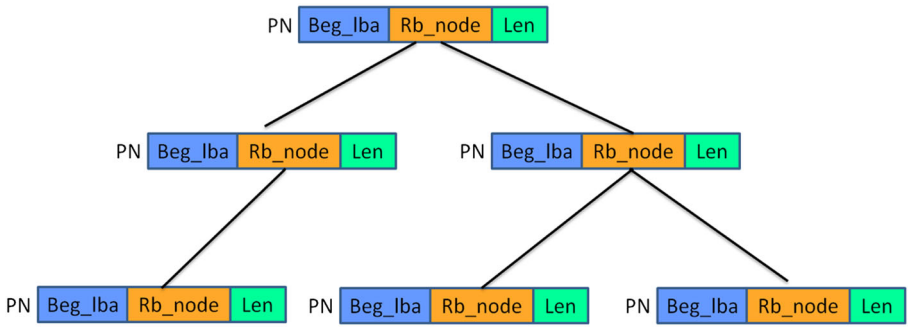


Figure 4. The *LBA_Tree* data structure.

DOI: 10.1002/spe

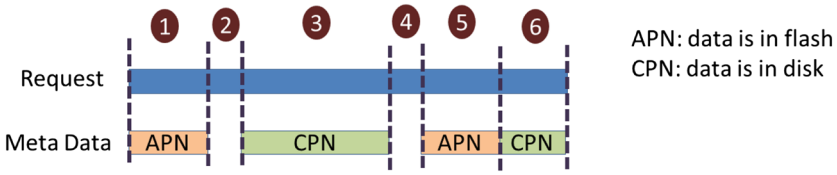


Figure 5. A request is partitioned into six request subsequences; two access flash and are described by APN nodes, two access disks and are described by CPN nodes, and two access disks but have not yet been described in the LBA_Tree.

six request subsequences. Subsequences 1 and 4 are cached in flash memory and are described by APN nodes, whereas subsequences 3 and 6 still reside in mechanical disks and are described by CPN nodes. Finally, subsequences 2 and 4 also access mechanical disks, but their runtime objects are not yet maintained in the *LBA_Tree*. Thus, ReqAnalyzer hands off subsequences 1 and 4 to the APNMgr and subsequences 2, 3, 4, and 6 to CPNMgr. Consequently, the profit value calculation is in fact based on these subsequences. The following subsections describe the two subcomponents, CPNMgr and APNMgr.

5.3.1. CPNMgr. As stated earlier, CPNMgr may receive a subsequence that has been described by a CPN or has not yet been recorded in *LBA_Tree*. First, if the subsequence does not yet have a corresponding CPN node, CPNMgr allocates a new CPN node in the *LBA_Tree*. Then, CPNMgr invokes the sequence detection scheme to merge the CPN with adjacent CPN nodes, if the LBAs of the CPN node are adjacent to the LBAs of its adjacent CPN nodes. Then, CPNMgr updates the metadata of the corresponding CPN. Notably, the *read count* or *write count* of evicted clean pages need not be updated because these pages in fact do not access the mechanical disks. Finally, CPNMgr calculates the new profit value of the CPN node. If it is larger than the minimal profit value of the APNs, the CPN is promoted to APN; the APN with the minimal profit value is demoted to CPN, and the *BaseProfit* is set to this minimal profit value.

However, the size of each sequence may be different. Thus, the size of a promoted CPN may be different from that of a demoted APN. As a result, if the sizes of the prompted CPNs are larger than that of the demoted APNs, the free space in the flash memory may be insufficient to store the newly promoted CPNs. Similarly, the capacity of flash memory is under-utilized if the sizes of the prompted CPNs are smaller than those of the demoted APNs. To prevent these two problems, we set a threshold, called *FREE_SPACE*. If the free space of the flash drive exceeds the threshold, APNs are not demoted. In addition, we repeatedly demote a set of APNs if the profit value of the prompted CPNs is larger than that of these APNs and the free space of the flash memory is smaller than the threshold.

Finally, to limit the space overhead of the CPNs, CPNMgr imposes a limit on the number of CPNs, and once the number exceeds the limit, CPMMgr evicts CPNs following the Least Recently Used (LRU) policy. Consequently, CPNMgr maintains an LRU list, called *CLURList*, which links all of the CPN nodes using the LRU policy, to select victim CPN nodes.

5.3.2. APNMgr. APMMgr manages the metadata of sequences cached in the flash memory, that is, APNs. APNs include all the attributes maintained by CPNs. APNs also maintain three additional attributes, the *profit value, converted LBA* in the flash memory, and *dirty bit*. Notably, the LBAs sent from the host falls within the range of the size of the mechanical disks. As a result, if a sequence is cached in the flash memory, APNMgr must convert its LBAs sent from the host to new LBAs, called *converted LBAs*, so that they are within the range of the flash memory. For example, an LBA number 123 might be mapped to a *converted LBA* 25, assuming that the size of the SSD is only 100 blocks. Therefore, an APN node maintains the *converted LBA* field. Currently, the transformation is achieved using a modular operation with a simple collision resolution scheme. Furthermore, the data cached in flash memory might be clean because ReqInterceptor intercepts *clean* data evicted from the page cache. Thus, APN maintains the *dirty bit* field to keep determine whether the corresponding sequence is clean or dirty. If it is clean, we do not need to write the data of the sequence to the mechanical disk when evicted. Finally, the APNs update the profit value and maintain the value in the *profit value* field.

Moreover, because the size of flash memory is limited, the number of APNs is also finite. As a result, APNMgr must be able to find the APN node with the smallest profit value. To accomplish

DOI: 10.1002/spe

this, APNMgr maintains a *ProfitTree* data structure, which is also a red-black tree, to facilitate the search for the smallest profit value of sequences cached in the SSD.

5.3.3. LBA_Tree, ProfitTree, and CLURList data structures. Based on the aforementioned discussions, to facilitate the implementation of Profit Caching, ReqAnalyzer, CPNMgr, and APNMgr respectively maintain *LBA_Tree*, *ProfitTree*, and *CLRUList*. Figure 6 shows an example of the relationship between these three data structures. *ProfitTree* links APN nodes and *CLRUList* links CPN nodes of *LBA_Tree*. In Figure 6, *ProfitTree* has two nodes, PVN1 (profit value node) and PVN2. PVN1 points to the APN1, and PVN2 points to the APN2 and APN3, which means that both APN2 and APN2 have the same profit value. Similarly, *CLRUList* links CPN1, CPN2, and CPN3, and the last one, that is, CPN3, denotes the least recently used CPN node.

5.4. ReqHandler component

ReqHandler implements the HA-CFQ scheduler. After receiving requests sent from the ReqAnalyzer, ReqHandler first sorts pending requests based on their LBAs and merges adjacent requests if possible. It then intercepts the *cfq_arm_slice_time*() function, which is invoked when the current request queue is empty and, following the approach described in Section 4, determines whether to pause in anticipation of a close-by request or to immediately serve the first flash memory request in the next request queue. Finally, it sends the request to the underlying hybrid disk drives.

5.5. Discussions

From the aforementioned discussions, Profit Caching determines whether a datum should be cached in flash memory or not. Therefore, it cannot be used in current hybrid disks. This is because, in the current disk command interface, there is no way to issue commands to the hybrid disks specifying that the data should be served by the flash memory or mechanical disks. Similarly, HA-CFQ needs to know the accessed media of each request, which is also impossible via the current disk interface. As a result, in Section 6, we implement a hybrid disk simulator to allow the host to manage the data of hybrid disks.

Fortunately, the new Revision 8 of the ATA specification will standardize a new hybrid drive command interface [22]. The new ATA-8 draft defines a new set of commands for hybrid disk drives, allowing the host to control and manage the on-disk flash memory or other types of non-volatile caches. Thus, our Profit Caching and HA-CFQ can be realized once the new ATA-8 command set is available.

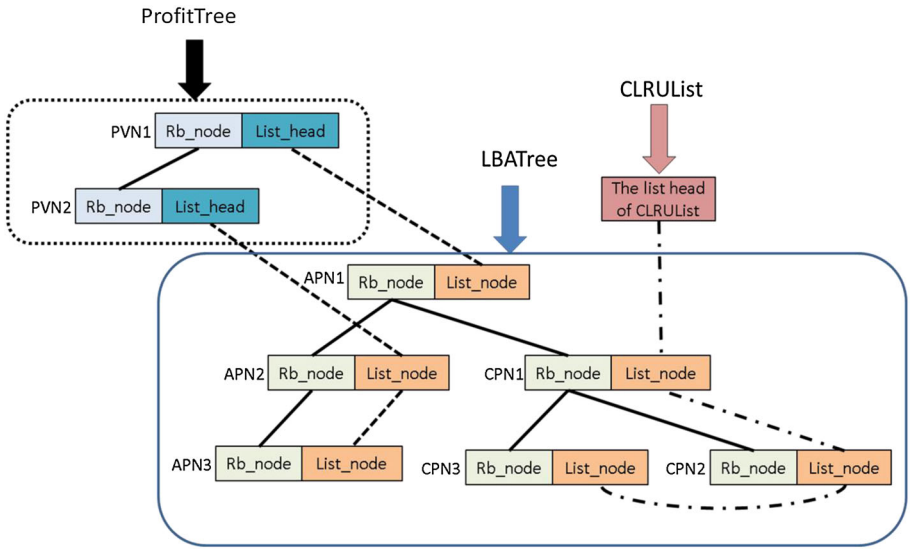


Figure 6. The relationship between *LBA_Tree*, *ProfitTree*, and *CLRUList*.

DOI: 10.1002/spe

6. EXPERIMENTAL STUDY

6.1. Experimental environment

In this section, we present the performance evaluation of Profit Caching and the HA-CFQ scheduler. First, we compare Profit Caching with previous RAF and FlashCache caching schemes. We implemented Profit Caching, RAF, and FlashCache in the Ubuntu 10.04. For Profit Caching, the *FREE_SPACE* threshold is set to 5% of the total flash memory capacity. However, as stated in Section 5.5, the current disk command interface does not allow the host to manage the on-disk flash memory of hybrid disks. As a result, we perform the experiment with the help of a trace-driven simulator. First, we log the request metadata sent by the modified Linux kernel, including the timestamp, starting LBA, request length, request type (read or write), and the accessed media (flash memory or mechanical disks) in a trace file. We then implement a hybrid disk simulator that reads the input trace file to calculate the various performance indexes, including throughput and response time. The simulated hybrid disk consists of a Maxtor ATLAS10K4 10K RMP (revolutions per minute) mechanical disk [23] and a MTRON SSD. For the Maxtor ATLAS10K4 mechanical disk, the seek time cost is calculated using the extracted data from DiskSim [24]. The rotational latency is assumed to be half the time of a full track revolution. Finally, the transfer time is calculated by dividing the request size with the 100 MB/s data rate. For the MTRON SSD, the time for page read, page write, and block erasure was obtained from the specification of the MTRON MSD-SATA3025-032 SSD, which is shown in Table II.

Five benchmarks, including IOMeter, Postmark, Filebench, IOzone, and Bonnie++, are used for performance evaluation. IOMeter is an I/O subsystem measurement and characterization tool for single and clustered systems [25]. It allows users to configure the workload and set operating parameters and then measures I/O subsystem performance for the configured and controlled load. We configured and generated three kinds of workloads, with the corresponding parameter settings shown in Table III. Postmark is an I/O intensive benchmark simulating the operation of an e-mail server [26]. In our Postmark tests, we created 30,000 files, each between 50 and 100 KB. Similarly, as shown in Table IV, we generated two kinds of workloads, *r50a50* and *r80a20*. Each workload performed 80,000 transactions, and the ratios of transaction types are shown in Table IV. Filebench

Table II. Mtron MSD-SATA3025-032 SSD parameters.

Interface	Cell density	Page size (KB)	Block size (KB)	Average data access	page read time	Page write time	Block erase time
SATA	SLC	2	128	0.1 ms	0.13 ms	0.41 ms	1.5 ms

Table III. Parameter settings of IOMeter benchmarks.

Workload	r80w90	r50w50	r20w80
Read ratio	10%	50%	20%
Sequential access ratio	20%	50%	80%
File size		9 GB	
Request size		4 KB	

Table IV. Parameter settings of Postmark benchmarks.

Workload	r50a50		r80a20
File sizes		50–100 KB	
Number of files		30,000	
Number of transactions		80,000	
Read : Append	50%:50%		80%:20%
Create : Delete		50%:50%	

DOI: 10.1002/spe

can generate a large variety of workloads by using loadable workload personalities to emulate complex applications, for example, mail, Web, file, and database servers [27]. As shown in Table V, we chose two server applications for the experiments: file server and Web server. Finally, IOzone and Bonnie++ are file system benchmarks that generate a variety of file operations and measure their respective performance [28, 29]. Their corresponding parameter settings used in the experiment are shown in Table VI.

6.2. *Profit Caching performance*

6.2.1. Selecting the proper value of α. As shown in Equation (1), calculating the profit value of a sequence consists of two parts: the first part calculates the access characteristics of the sequence, and the second part measures its recency. The parameter α is used to determine the relative weight of these two parts. Large values of α make the profit value of a sequence more responsive to access characteristics, whereas smaller values emphasize recency over access characteristics. To determine the optimal value of α, an intuition solution is to periodically try out different values of α on past data and to choose the best one that derives the highest throughput or smallest response time. However, this approach involves significant space and runtime overhead. In addition, when the workload is changed, the selected value of α might not be optimal anymore. An enhanced method is to apply some heuristics to limit the search space. However, such an approach might obtain a *local* optimal value.

Instead, we adopt a simpler and commonly used approach that off-line measures the response time and throughput of Profit Caching under different values of α to determine the proper value of α. Figure 7 shows the experimental results. When $\alpha = 0.7$, Profit Caching derives the largest throughput in *iometer r20w80*, *iometer r50w50*, *postmark r80a20*, and *bonnie++* workloads and the smallest average response time in *iometer r20w80*, *iometer r50w50*, *filebench_fileserver*, *filebench_webserver*, and *bonnie++*. Consequently, we set $\alpha = 0.7$ for the following experiments.

6.2.2. Effectiveness of Profit Caching. In this subsection, we demonstrate the effectiveness of Profit Caching. First, we present the average (sub)sequence size of each caching scheme for the mechanical disk. Figure 8 shows the experimental results. Because the mechanical disk performs better for sequential accesses, the mechanical disk should serve sequences with a larger size. As shown in Figure 8, both Profit Caching and RAF derive a larger sequence size for all workloads. In contrast, FlashCache always sends the smallest sequence to the mechanical disk. This is because both Profit Caching and RAF detect and try to serve sequential streams from the mechanical disk. In contrast, FlashCache tries to absorb all workloads, regardless of whether they are sequential or random accesses.

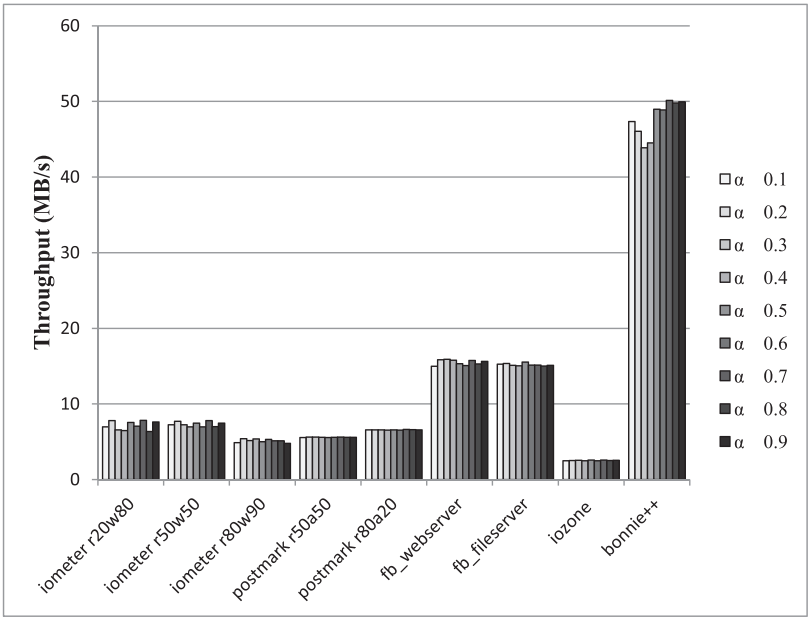
Table V. Parameter settings of Filebench benchmarks.

Workload	fb_fileserver	fb_webserver
Average file sizes	80 KB	75 KB
Number of files	117,965	105,000
Number of threads	50	100
Mean directory width	20	
I/O size	1 MB	
Mean append size	16 KB	

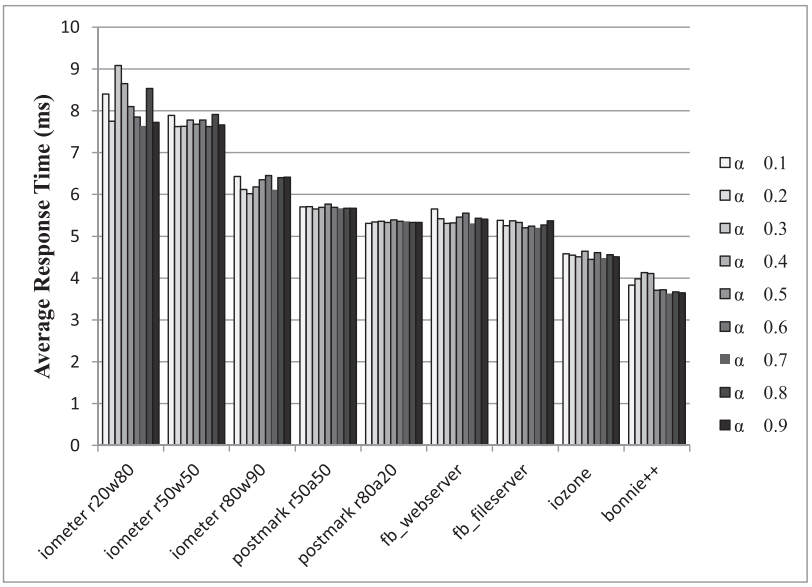
Table VI. Parameter settings of IOzone and Bonnie++ benchmarks.

Workload	IOzone	Bonnie++
File sizes	2 GB	4 GB
Transaction types	Write, re-write, random read, random write	Write, re-write, read, random seek

DOI: 10.1002/spe



(a). The throughput of Profit Caching under different value of α



(b). The average response time of Profit Caching under different value of α

Figure 7. (a). The throughput of Profit Caching under different value of α. (b). The average response time of Profit Caching under different value of α.

We then measure the *read* hit ratio for the three caching schemes. Because the read operations are faster than write operations on flash memory, we should try to absorb as much read traffic as possible through flash memory. Figure 9 shows the experimental results, with Profit Caching showing the highest read hit ratio for most workloads. This is because Profit Caching differentiates reads from writes and gives read-intensive data a higher profit value to be cached in flash memory.

In contrast, RAF derives the smallest read hit ratio because RAF does not differentiate and give different priority for read and write requests. Besides, RAF does not cache data immediately on startup, but must rather first detect randomly accessed sequences. After such a sequence is detected,

DOI: 10.1002/spe

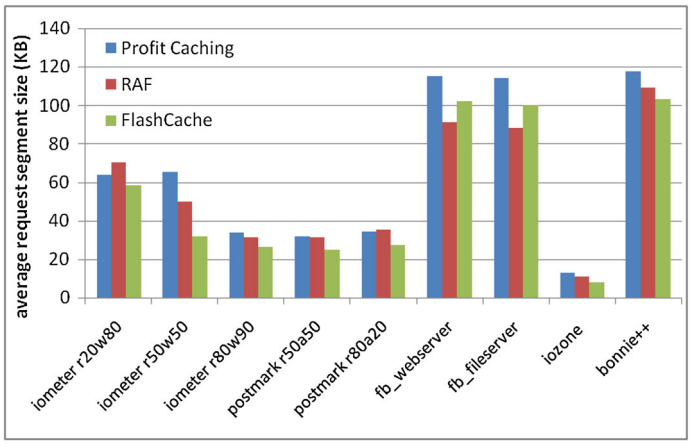


Figure 8. The average request subsequence size on mechanical disks of Profit Caching, RAF, and FlashCache under different workloads.

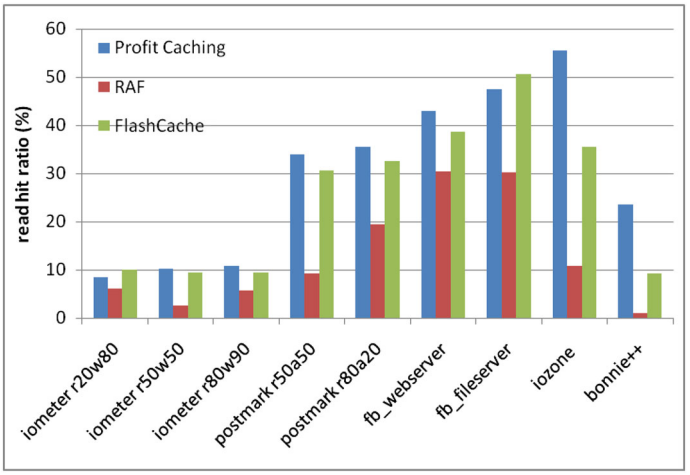


Figure 9. The read hit ratio of Profit Caching, RAF, and FlashCache under different workloads.

RAF records all of the blocks of the sequence in a random-access block hash table (RBHT) table. When the LBA of the requested data is found in the RBHT, that is, identified as a random-access data, the data are then cached in flash memory. Furthermore, RAF reserves a fixed amount of flash memory space as a write cache. However, the fixed and inflexible size of the write cache might occupy limited caching space in flash memory, which, in turn, has a severely negative impact on the read hit ratio, especially in a read-dominated workload. As a result, RAF derives a significantly smaller read hit ratio in the workloads.

Finally, we measure the impact of various schemes on the life cycles of SSDs. First, we present the number of write requests served by SSDs under the three SSD caching schemes. Figure 10 shows the experimental results. From Figure 10, Profit Caching scheme sends the least number of write requests to SSDs in almost workloads. This is because Profit Caching considers the asymmetric costs of read/write operations of SSDs and tries to send write requests, especially sequential write requests, to the mechanical disks. In contrast, FlashCache always sends the largest number of write requests to the SSDs, because FlashCache uses SSDs to absorb all of the disk workload, regardless of read or write requests. Notably, RAF also ignores the asymmetric costs of read/write operations of SSDs. However, RAF considers the data's sequentiality and requests whose sizes are larger than a threshold are classified as sequential

DOI: 10.1002/spe

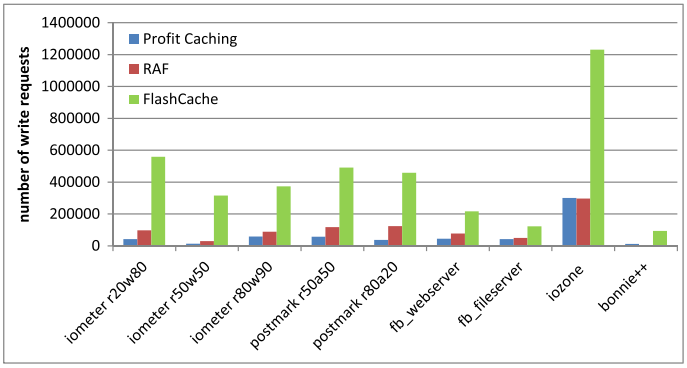


Figure 10. The number of write requests served SSDs by Profit Caching, RAF, and FlashCache under different workloads.

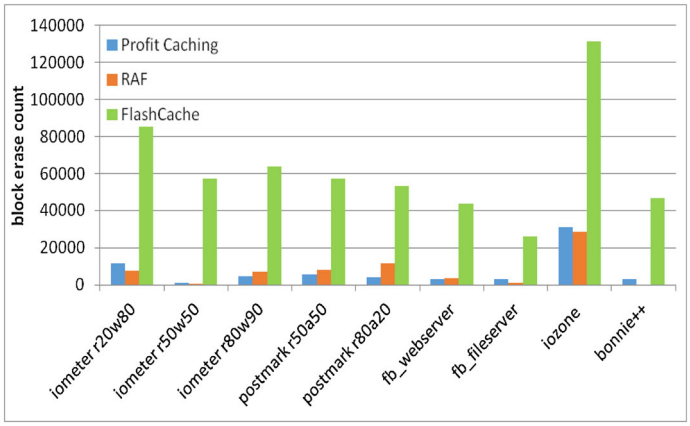


Figure 11. The erase count of Profit Caching, RAF, and FlashCache under different workloads.

and served by the mechanical disks. Thus, compared with FlashCache, RAF also derives a less number of SSD write requests. Then, we measure the number of erase operations under the various data caching schemes by a DiskSim based SSD simulator [30]. Figure 11 shows the experimental results, with both Profit Caching and RAF also deriving a smaller number of block erase operations.

6.2.3. Overall performance. Finally, we measure the overall performance of Profit Caching, RAF, and FlashCache under different workloads. Figure 12 shows the average throughput of the three caching schemes. In addition, we present the improvement of Profit Cache compared with RAF and FlashCache in Table VII. From Figure 12, Profit Caching derives the largest throughput because Profit Caching considers all of the performance-critical data indicator metrics. In contrast, FlashCache derives the smallest throughput because it only considers the data recency and thus cannot identify data suitable for caching in the SSD. Specifically, from Table VII, in comparison with RAF, Profit Caching increased the average throughput by 21.35% on average and by up to 86.23%. Also, Profit Caching outperforms FlashCache by 123.5% on average and by up to 678.42%.

Figure 13 presents the average response time of the three caching methods, and Table VIII the improvement of Profit Cache compared with RAF and FlashCache. Similar to the results shown in Figure 12, Profit Caching has the smallest value of average response time, and FlashCache derives the worst performance result. Besides, from Table VIII, Profit Cache reduces the average response time with an average of 13.28% (up to 45.1%) compared with the RAF and with an average of 47% (up to 87.12%) compared with FlashCache.

DOI: 10.1002/spe

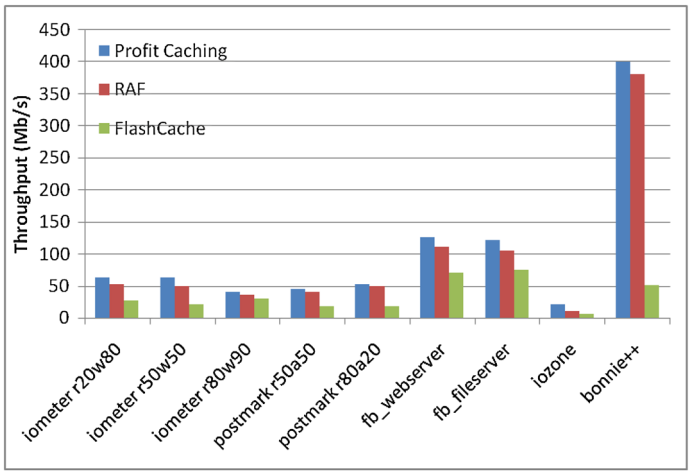


Figure 12. The throughput of Profit Caching, RAF, and FlashCache under different workloads.

Table VII. Compared with RAF and FlashCache, the throughput improvement of Profit Caching under the different input workload.

	IOMeter			Postmark		Filebench		IOzone	Bonnie ++
Workload	r20w80	r50w50	r80w90	r50a50	r80a20	fb_webserver	fb_fileserver		
RAF (%)	19.00	24.60	16.10	10.65	7.29	13.87	14.61	86.23	5.14
FlashCache (%)	135.84	196.58	34.38	147.14	205.07	77.68	62.80	252.05	678.42

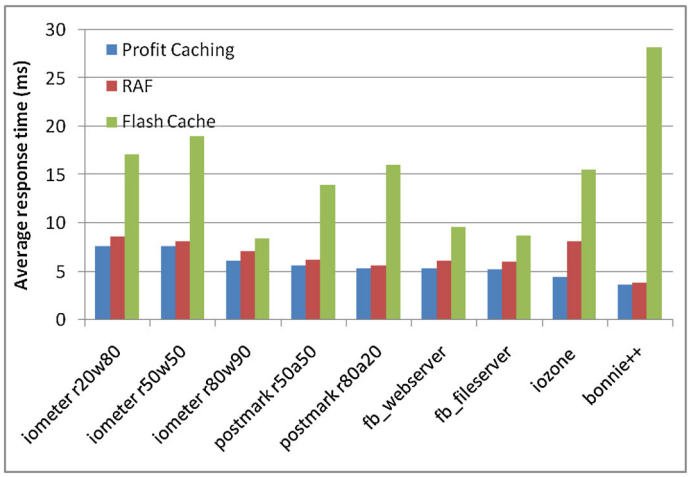


Figure 13. The average response time of Profit Caching, RAF, and FlashCache under different workloads.

Table VIII. Compared with RAF and FlashCache, the average response time improvement of Profit Caching under the different input workload.

	IOMeter			Postmark		Filebench		IOzone	Bonnie ++
Workload	r20w80	r50w50	r80w90	r50a50	r80a20	fb_webserver	fb_fileserver		
RAF (%)	11.57	6.62	13.94	9.42	5.47	12.95	14.47	45.10	4.97
FlashCache (%)	55.48	59.83	27.00	59.47	66.60	44.97	40.57	71.19	87.12

DOI: 10.1002/spe

Finally, we define a new performance metric called *erase benefit* that is calculated as follows.

$$\textit{erase benefit} = \textit{the number of read hit request} / \textit{the number of erase operations} \quad (4)$$

In SSD caching, a flash memory block is erased for sustaining new arriving data, expecting that the data are hot and, especially, will be read frequently in the future. As shown in Equation (4), *erase benefit* measures the effectiveness, that is, the average number of read hit, of erasing a block. Thus, a well-behaved SSD caching scheme should maximize the value of erase benefit. Figure 14 shows the erase benefit of each data caching scheme. Similar to the previous experiments, Profit Caching derives the largest erase benefit value, whereas FlashCache has the least result.

6.3. HA-CFQ scheduler performance

As stated in Section 4, the HA-CFQ scheduler can avoid unnecessary idling found in the CFQ scheduler. Thus, to verify the effectiveness of the HA-CFQ scheduler, we measure the time savings derived by using the HA-CFQ scheduler as a percent of the total experimental execution time. Results shown in Table IX indicate the time savings is insignificant because the CFQ scheduler only anticipates when the current request queue becomes empty, instead of anticipating every request like the Anticipatory Scheduling algorithm. Besides, HA-CFQ can eliminate or reduce the anticipation time only when the first request in the next queue is a flash request; otherwise, HA-CFQ behaves like the CFQ scheduler, thus resulting in limited opportunities to save time. Nevertheless, HA-CFQ can indeed successfully avoid unnecessary I/O anticipations in the hybrid disk environment.

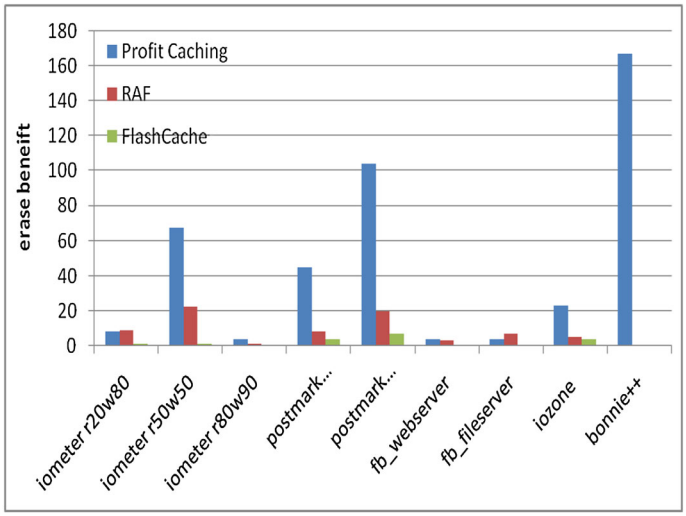


Figure 14. The erase benefit of Profit Caching, RAF, and FlashCache under different workloads.

Table IX. Compared with CFQ, the percentage of time savings by HA-CFQ in the total experimental execution time.

	IOMeter			Postmark		Filebench		IOzone	Bonnie ++
Workload	r20w80	r50w50	r80w90	r50a50	r80a20	fb_webserver	fb_fileserver		
Percentage of time savings	1.74	1.16	0.86	1.61	3	0.72	0.89	0.68	0.96

DOI: 10.1002/spe

Table X. The timing overhead of Profit Caching in different input workloads.

	IOMeter			Postmark		Filebench		IOzone	Bonnie ++
Workload	r20w80	r50w50	r80w90	r50a50	r80a20	fb_webserver	fb_fileserver		
Timing overhead (%)	0.20	0.17	0.25	0.27	0.15	0.24	0.21	0.18	0.24

6.4. Timing overhead of Profit Caching

As stated in Section 3, Profit Caching needs to maintain the metadata of each (sub)sequence and calculate the corresponding profit value. Thus, in the last experiment, we measure the timing overhead of Profit Caching. Table X shows the experimental results for timing overhead of Profit Caching as a percentage of the total experimental execution time and indicates that the timing overhead of Profit Caching is very small. This is because, first, as stated in Section 3.3, we associate a runtime object with each (sub)sequence, instead of blocks, for maintaining the metadata and calculating the profit. Because a (sub)sequence consists of a number of blocks, the number of runtime objects maintained is significantly decreased. Consequently, we can reduce the timing overhead for maintaining and searching the runtime objects. Second, as stated in Section 3.3, Profit Caching maintains the *ProfitTree* and *LBA_Tree* using the red-black tree and seamlessly integrates these data structures. Consequently, both trees can be efficiently searched and maintained. From the aforementioned discussion, the timing overhead of Profit Caching is insignificant and, as shown in Table X, can be disregarded.

7. CONCLUSIONS AND FUTURE WORK

The tremendous growth of flash memory is driving the development of hybrid disks or heterogeneous storage that strikes a balance between performance and cost. To fully exploit the effectiveness of hybrid disks or heterogeneous storage, we present a unique solution that effectively positions flash memory in the storage hierarchy and achieve the optimization goal. First, by dynamically monitoring I/O traffic on the fly, the proposed Profit Caching mechanism can transparently and automatically identify the most valuable data and places them in the flash memory cache. Profit Caching considers all possible performance indicators, including read count, write count, sequentiality, randomness, and recency, and seamlessly combines these indicators by the GreedyDual algorithm. Second, based on the CFQ scheduler, we propose an HA-CFQ scheduler to avoid unnecessary I/O anticipations in hybrid disks.

A prototype was implemented on the Linux kernel following modular design principles to minimize system changes. Coupled with a trace-driven simulator, detailed experiments were also conducted under a variety of workloads. Results indicate that the throughput of Profit Caching is almost 1.8 times that of RAF and 7.6 times that of FlashCache. In addition, the HA-CFQ scheduler can reduce unnecessary idle time found in CFQ by at most 1.74% of the total execution time.

In addition to the caching issue, the entire I/O path should also be re-designed for heterogeneous storage systems. Thus, our future work will investigate buffer cache management schemes and prefetching mechanisms in heterogeneous storage environments. Besides, we will also extend our work to the cloud computing environment. Usually, cloud computing delivers services in a 'pay-as-you-go' model, and thus, cloud providers need to meet different QoS parameters of each individual consumer that are negotiated in specific service-level agreements. Our future work will also extend the multi-tiered storages in cloud computing environment and investigate the SSD caching scheme in such a multi-tenant and multi-dimensional cloud environment.

ACKNOWLEDGEMENT

This work is supported by the National Science Council under project NSC 99-2221-E-005-076- and NSC 101-2221-E-006-098-MY3.

DOI: 10.1002/spe

REFERENCES

1. Seagate Momentus XT overview. http://knowledge.seagate.com/articles/en_US/FAQ/214735en?language=en_US.
2. Kgil T, Mudge T. FlashCache: a NAND flash memory file cache for low power web servers. *International Conference on Compilers, Architecture and Synthesis for Embedded Systems* 2006: 103–112.
3. Liu Y, Huang J, Xie C, Cao Q. RAF: a random access first cache management to improve SSD-based disk cache. *IEEE International Conference on Networking, Architecture, and Storage* 2010: 492–500.
4. Saxena M, Swift MM, Zhang Y. FlashTier: a lightweight, consistent and durable storage cache. *Proceedings of the 7th ACM european conference on Computer Systems* 2012: 267–280.
5. Chen F, Koufaty D, Zhang XD. Hystor: making the best use of solid state drives in high performance storage systems. *Proceedings of the international conference on Supercomputing* 2011: 22–32.
6. Wu XJ, Narasimha Reddy AL. Exploiting concurrency to improve latency and throughput in a hybrid storage system. *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* 2010: 14–23.
7. Jiang S, Zhang XD. Making LRU friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance. *IEEE Transactions Computers* 2005; **54**(8): 939–952.
8. Iyer S, Druschel P. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* 2001: 117–130.
9. D Yuhui, Z Jipeng. Architectures and optimization methods of flash memory based storage systems. *Journal of Systems Architecture* 2011; **57**(2): 214–227.
10. Samsung Electronics 1gx8bit/2gx16bit NAND flash memory. http://www.samsung.com/Products/Semiconductor/NANDFlash/SLCLargeBlock/16Gbit/K9WAG08U1M/K9WAG08U1M.htm.
11. Fusion Drive: high capacity meets high performance. http://www.apple.com/imac/performance/#fusion
12. Appuswamy R, van Moolenbroek DC, Tanenbaum AS. Integrating flash-based SSDs into the storage stack. *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)* 2012: 1–12.
13. Kim YJ, Kwon KT, Kim J. Energy-efficient file placement techniques for heterogeneous mobile storage systems. *Proceedings of the 6th ACM&IEEE International Conference on Embedded Software (EMSOFT)* 2006: 171–177.
14. Lin L, Zhu YF, Yue JH, Segee B. Hot random off-loading: a hybrid storage system with dynamic data migration. *Proceedings of the IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* 2011: 318–325.
15. Payer H, Sanvido MA, Bandic ZZ, Kirsch CM. Combo drive: optimizing cost and performance in a heterogeneous storage device. *Proceedings of the Workshop on Integrating Solid-state Memory into the Storage Hierarchy* 2009: 1–8.
16. Wu XJ, Narasimha Reddy AL. Managing storage space in a flash and disk hybrid storage system. *Proceedings of the IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009: 1–4.
17. Deng Y, Wang F, Helian N. EED: Energy efficient disk drive architecture. *Information Sciences* 2008; **178**(22): 4403–4417.
18. Xiao WJ, Lei XQ, Li RX, Park NH, Lilja DJ. PASS: a hybrid storage system for performance-synchronization tradeoffs using SSDs. *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications* 2012: 403–410.
19. Park S, Shen K. FIOS: a fair, efficient flash I/O scheduler. *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)* 2012: 13–13.
20. Cao P, Irani S. Cost-aware WWW proxy caching algorithms. *Proceedings of the USENIX Symp. Internet Technologies and Systems* 1997: 18–18.
21. Young N The k-server dual and loose competitiveness for paging. *Algorithmica* 1994; **11**(6): 525–541.
22. Stevens CE. AT Attachment 8 – ATA/ATAPI Command Set (ATA8-ACS). http://www.t13.org/documents/UploadedDocuments/docs2007/D1699r4a-ATA8-ACS.pdf
23. Maxtor ATLAS10K4 146G, 10000rpm SCSI, http://ovahldy.blogspot.com/2011/03/recent-diskmodels-for-disksim.htm.
24. DiskSim, http://www.pdl.cmu.edu/DiskSim/
25. IOMeter, http://www.iometer.org/
26. Katcher J. Postmark: a new file system benchmark. http://www.netapp.com/techlibrary/3022.html.
27. Filebench. http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench.
28. IOZONE file system benchmark. http://www.iozone.org/
29. Bonnie++. http://www.coker.com.au/bonnie++/.
30. Microsoft 2009. SSD extension for DiskSim simulation environment. http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/

DOI: 10.1002/spe