SAC: Rethinking the Cache Replacement Policy for SSD-based Storage Systems

Zhiguang Chen
State Key Laboratory of High Performance Computing, School of Computer, National University of Defense Technology
chenzhiguanghit@gmail.com

Nong Xiao
State Key Laboratory of High Performance Computing, School of Computer, National University of Defense Technology
nongxiao@nudt.edu.cn

Fang Liu
State Key Laboratory of High Performance Computing, School of Computer, National University of Defense Technology
liufang@nudt.edu.cn

Abstract

Solid-state drives (SSDs) are widely used in storage systems. However, algorithms adopted by existing operating systems generally consider the underlying devices as hard disks, and thus are rarely optimized for SSDs. In this paper, we focus on a classical research issue, the cache replacement policy, and design a new policy by taking the parallelism of SSDs into account.

A typical SSD contains several parallel channels. Some channels contain more hot data, thus are busy with read requests. The other channels may only contain cold data. So, workloads among these channels may be unbalanced. Requests issued to busy channels may take a while to address, whereas requests issued to idle channels may be served rapidly. We design a new cache replacement policy for read data, which considers the unbalanced workloads among channels. The policy gives higher priority to evicting pages from idle channels because they are more easily retrieved. On the other hand, pages from busy channels are protected. In this manner, the average latency for obtaining pages is reduced. However, SSDs are black boxes, with operating systems are blind to the channel from which a page comes. Therefore, we propose a simple scheme that determines whether a page is from a busy or an idle channel. The scheme monitor the page requests issued to the underlying storage device. When a page request is returned, and many page requests issued earlier than it have not been returned, the page is assumed to be from an idle channel.

We compare our cache replacement policy with others via trace-driven simulations. The performance is measured in terms of average response time. Comparison candidates include LRU, ARC and the policy adopted by Linux. The experimental results show that our policy outperforms the other policies on most traces when workloads among channels are unbalanced.

SYSTOR'12, June 4--6, 2012, Haifa, Israel.

Categories and Subject Descriptors D.3.3 [**Computer Architecture**]: Storage Systems, Parallel Input and Output Systems –*solid-state drive, parallel IO.*

General Terms Algorithms, Performance, Design, Experimentation.

Keywords *Flash memory; SSD; cache replacement policy; storage; SAC.*

1. INTRODUCTION

The NAND flash memory has undergone significant advancements in recent years. Flash memories continuously meet the demand for large-scale storage systems as their capacity becomes upgraded and their price gets reduced. Accordingly, flash-based solid-state drives (SSDs) are attracting increasing attention because of their superior performance compared with hard disks. For example, the access latency of SSDs is an order of magnitude lower than that of hard disks, and their energy consumption may be as low as one-sixth that of hard disks. In addition, SSDs are lighter, and they produce no noise. In particular, SSD-based systems exhibit parallelism in multiple levels. Multilevel parallelism produces an aggregate bandwidth that exceeds the limitation of the host interface. Given all these advantages, SSDs have been widely accepted in industrial communities and are considered promising substitutes for hard disks.

At present, SSDs are incorporated into storage systems as replacements for hard disks. Software components over SSDs remain traditional and hard disk-based. SSD-oriented optimizations have been rarely explored [1, 2]. In this paper,

we review a classical issue of storage systems: the cache replacement policy. Traditional caches usually pursue a higher hit ratio by evicting the page most unlikely to be reused (typically, the least recently used page). They assume that latencies for obtaining different pages are identical, and that a higher hit ratio indicates higher performance. However, in SSD-based systems, this assumption is inaccurate.

Latencies for obtaining pages from SSDs may vary for the following reasons. In this study, the parallelism exhibited by SSDs is abstracted as multiple parallel channels. Within a period of time, workloads issued to these channels may be unbalanced. Some channels are busy, with issued requests taking a while before they can be served. Some channels are idle, and issued requests can be rapidly served. In other words, latencies for obtaining pages are no longer uniform. The principle behind this phenomenon is that the cache should protect pages from busy channels, because these pages are expensive to retrieve once evicted. On the other hand, pages from idle channels should be evicted with higher priority because they can easily be retrieved. Therefore, we design a novel cache replacement policy for storage systems composed of SSDs. This policy gives higher priority to evicted pages from idle channels.

However, SSDs are black boxes, with their inner configurations remaining the intellectual properties (IP) of manufacturers. Therefore, these devices are invisible to system software, and operating systems are blind to the channel from which a page originates. As a result, the cache cannot select the proper page to evict. In this paper, we propose a simple mechanism to predict whether a page comes from an idle channel. We maintain a queue. When a page request is issued to the storage devices, it joins the queue, and then waits for data. Once its data are returned, the request is removed from the queue. Page requests join the queue in the order of their issuance by the operating systems. However, they are removed from the queue in out-of-order manner because they target different channels. And the response times are different. If a page leaves the queue quickly, we assume that it comes from an idle channel. By contrast, requests waiting in the queue for a long time are assumed to come from busy channels. In this manner, pages from busy and idle channels are distinguished. When the cache is full, pages from idle channels are prioritized for eviction. The policy discussed above divides pages into two types: pages from idle channels, and those from busy channels. In fact, we propose a more reasonable metric to characterize each page. The algorithm details are explained in Section 4.

Generally, our cache replacement policy is aimed at SSD-based storage systems. Thus, we call it SSD-aware cache (abbreviated as SAC hereafter). The aim of SAC is to protect data originating from busy channels to reduce the average access latency. Even though SAC is designed for SSDs, it may also be applied to any parallel storage system such as hard disk- or SSD-based RAIDs. Furthermore, SAC is applicable to systems exhibiting no parallelism, such as a hard disk. However, in this case, SAC behaves in a manner similar to that in the traditional replacement policy.

We evaluate SAC via trace-driven simulations. SAC does not pursue a higher hit ratio; rather, it tries to achieve a lower response time. Hence, we adopt the average response time, rather than the hit ratio, as the performance metric. In the experiments, we simulate different striping schemes to produce different parallelism. Comparison candidates include LRU [3], ARC [4], and the policy adopted by Linux [5]. The experimental results show that on most unbalanced workloads, SAC achieves a lower average response time than other schemes by at least 10%.

The rest of the paper is organized as follows: Section 2 introduces some classical cache replacement policies; Section 3 analyzes the parallelism of SSD-based storage systems, and also demonstrates the unbalanced workloads among channels via trace-driven simulations; Section 4 provides a detailed explanation on the proposed cache replacement policy; Section 5 compares SAC with some other cache replacement policies; and the last section concludes our work.

2. Related works

This section introduces some related studies on the cache replacement policy. Subsection 2.1 presents some traditional studies. Subsection 2.2 introduces several policies that consider the parallelism or access latency.

2.1 Classical replacement policies

The cache replacement policy is a classical research issue. A variety of policies has appeared in the past few decades. The subsequent paragraphs enumerate some of these classical studies.

To achieve a higher hit ratio, the cache should replace the page that is the most unlikely to be reused. MIN [3] is the only one that achieves this goal. It computes the time difference between the current reference and the next reference for each page. The page with the most difference is replaced. But, MIN is an offline policy, thus is impractical. LRU [3] imitates MIN by monitoring the time difference between the last reference and the current reference. It uses the time of the last reference to a page to predict the time of the next reference, thus becomes practical.

ARC [4] is widely referenced for its simplicity and high performance. It maintains two caches: a physical cache and a ghost cache. The physical cache is a conventional cache used to accommodate data. By contrast, the ghost cache contains pages that have been evicted from the physical cache. Strictly speaking, the ghost cache simply maintains the index entries of evicted pages; the contents of these

pages are discarded. In other words, the ghost cache merely keeps the replacing history. The physical cache is organized into two lists. One list focuses on the frequency, whereas the other focuses on recency. The decision for physical cache replacements is guided by the feedback from the ghost cache. The pages in the recency list are more valuable when an access hits the ghost cache a target page is evicted from the list. Thus, ARC prioritizes the replacement of pages in the frequency list. On the other hand, page replacements in the recency list are highly prioritized if an access hits the ghost cache and the target page was evicted from the frequency list. ARC takes advantage of the feedback, thereby achieving superior performance on most workloads.

The cache replacement policy adopted by Linux [5] is a variation of the LRU. In the LRU, the queue must be locked before a page can join or leave. Competition for locking among multiple threads results in serious performance degradation. Thus, LRU is impractical in current systems, which are based on multi-cores. CLOCK [6] was proposed to improve the LRU. CLOCK maintains a queue similar to that of the LRU. When an access hits a page in the cache, the page is not moved to the MRU position. Instead, it is marked as reused. This scheme prevents queue locking when a page hits the cache. When a replacement is required, the CLOCK evicts pages in the LRU end. However, if the LRU page is marked as reused, it is moved to the MRU end. The page then has another chance to stay in the cache. Linux maintains two CLOCK queues. The first queue manages approximately one-third of the cache capacity and is used to eliminate correlated accesses [7]. It also helps in the recycling of frequently accessed pages. Pages evicted from the first queue are moved to the second queue. If a page is evicted by the second queue, it is ultimately discarded by the cache. We use this as the basis for SAC implementation in our study.

Aside from the policies discussed above, many high-performance cache replacement policies aiming at different application environments have been proposed. 2Q [8] is designed for databases. The multi-queue [9] is applicable to multi-level caches. LIRS [10] is adopted by MySQL. The common characteristic of these policies is that they assume that all I/O requests are uniform. The latency of fetching a page is identical to obtaining another page. Thus, they merely pursue the hit ratio. The access latency of missed page is rarely optimized.

2.2 Cost-aware replacement policies

In practical systems, I/O requests usually endure different latencies. The costs of retrieving missed pages are no longer identical. Therefore, the cache should avoid replacing pages that are difficult to retrieve. In fact, some replacement policies already explore this design principle.

Qureshi et al. [11] focused on memory-level parallelism (MLP). They argued that cache misses in MLP are not uniform. Some misses occur in isolation, whereas others occur in parallel with others. Isolated misses are significantly more costly on performance than parallel misses. However, a traditional cache is blind to the MLP-dependent cost difference between different misses. Hence, a new framework for an MLP-aware cache was proposed. Their policy gives higher priority to replacing data that may be missed in parallel, because this kind of data would be fetched in batches. Thus, the latency of retrieving each piece of data is relatively lower on average. This policy dramatically reduces the number of memory-related stalls.

DULO [12] is a cache replacement policy that considers the access latency for hard disks. Requests issued to hard disk must endure two types of latencies. The first is the time interval used to move the disk head to the target position. This latency type is called the access latency. The second type is the time interval used to transmit data. This latency type depends on the number of requested bytes and is called the transfer latency. Generally, the access latency is much higher than the transfer latency. DULO considers two types of requests, namely, short and long. A short request requires only a few sectors, whereas a long request requires a long sequence. Even though two types of requests require different numbers of sectors, their latencies are almost similar, because most of the latencies are attributed to the access latency, which does not depend on the length of requests. Hence, the average cost of a short request is much higher. DULO replaces long sequences with a higher priority. Once a long sequence is replaced, a larger cache capacity becomes available. In addition, the retrieval of a long sequence is easy. DULO significantly reduces the average latency for accessing hard disks. However, it assumes that, the time of retrieving a long sequence is similar with that of retrieving a short one. For SSDs, this assumption may be not true. The time of retrieving a sequence from an SSD depends on the inner configuration of the SSD. So, DULO is not applicable to SSDs.

VDF [13] is another cost-aware replacement policy designed for the cache over RAIDs. When a disk in a RAID fails, the RAID still provides service to users. If a request targets the faulty disk, the required data must be reconstructed by the surviving disks. Thus, a request to a faulty disk brings in N requests, where N is the number of surviving disks. For example, in a RAID5 consisting of four disks, a request to the faulty disk leads to three accesses that are respectively issued to three surviving disks. By contrast, requests to the surviving disks do not result in additional accesses. In conclusion, obtaining data from a faulty disk is more costly. Hence, VDF protects data from a faulty disk. This policy improves the performance of the disk array and allows more bandwidth for online reconstruction. However, it is only RAID-oriented.

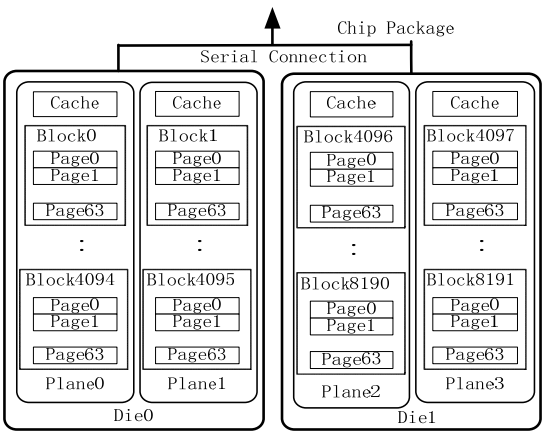

Figure 1. Configuration of a flash chip

The replacement policies introduced above consider the cost of retrieving missed data. Therefore, hard-to-retrieve data are protected. SAC conforms to this principle, as well. The difference between SAC and the other policies is the cost definition. Cost, as defined by SAC, relates to the parallelism of SSDs.

3. Motivations

Our work is motivated by two facts: parallelism existing in SSD-based storage systems, and the unbalanced workloads among channels. These two facts are explained in the next two subsections.

3.1 Parallelism in SSD-based systems

SSD-based systems exhibit parallelism in at least three different levels. First, flash chips used to construct SSDs may be accessed in parallel. Second, many flash chips are used to build an SSD. The SSD itself contains multiple channels. Moreover, several SSDs compose an array, which is also a parallel device.

Flash chips are hierarchically configured. Figure 1 shows a typical architecture of a flash chip. A chip package contains two dies, with each die linked to the SSD controller by an independent serial connection. Thus, different dies can be accessed in parallel. A die consists of two planes that share a serial connection. However, two planes can still be accessed in parallel because each plane has a cache. Once data are kept in the cache, different planes can work concurrently. For example, in a write request, the SSD's controller first transmits data to the cache. The cache then writes data to the storage media. Hence, even though two planes share the bus, they can write data concurrently. In fact, two planes sharing a bus can perform any operation independently. In particular, if a read or write is performed concurrently with an erase, the shared bus does not become a bottleneck because the erase function does not involve any data.

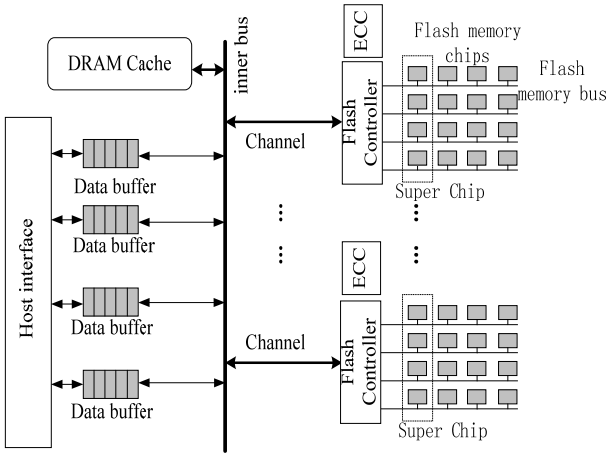

Figure 1. **SSD architecture**

Aside from the parallelism within flash chips, SSDs themselves exhibit parallelism. Figure 2 shows the configuration of an SSD. The SSD contains tens of flash chips divided into several groups. Chips in a group are linked by a channel. Channels can transmit data in parallel. To exploit as much parallelism as possible, SSDs usually issue I/O requests in a striping manner [14–16]. For example, a request demanding multiple pages is divided into several subrequests. Each of these subrequests is dispatched to a channel. A corresponding chip group then responds to the subrequest allocated to the channel. A subrequest may contain several pages, which can be further interleaved within the group. In conclusion, SSDs exhibit parallelism everywhere.

Lastly, the RAID built by SSDs also exhibits parallelism. The parallelism in RAID is widely demonstrated; therefore, we do not discuss it in detail in this study.

The parallelism in SSD-based storage systems has been studied extensively [17–21]. Compared with the listed related studies, our work presents two characteristics. First, we do not focus on any given type of parallelism, and all types of parallelism are uniformly abstracted as parallel channels. Second, our work does not intend to enhance parallelism, but to try to utilize as much parallelism as possible by balancing the workloads among channels. This goal is achieved through our cache replacement policy.

3.2 Unbalanced read workloads among channels

This subsection demonstrates the unbalanced workloads among channels. First, we focus on read data. Some of these data may be accessed frequently, whereas the others may never be accessed. However, SSDs cannot determine whether a piece of data is hot or not when the data are written to different channels. Thus, some channels may contain more hot data than others, and are likely to become hot spots. As read data are rarely moved (except for by the static wear leveling, which happens very infrequently), the hot spot remain until corresponding data become cold.

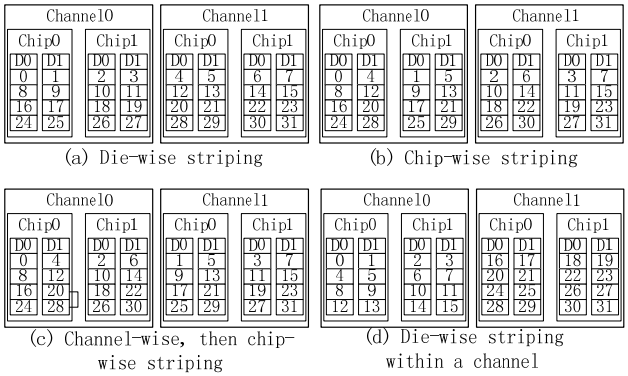

Figure 3. Different striping schemes

Second, as the hot data are randomly assigned to different channels, theoretically, all channels have the same volume of hot data. But, in a short period of time, the hot data of some channels are sleep, whereas the hot data of the other channels may be accessed frequently. So, even though the total hot data are evenly distributed to different channels, within a given short period of time, the read workloads among channels will still be unbalanced. The cache just maintains recently accessed data, it is aware of the imbalance. This subsection demonstrates the imbalance via trace-driven simulations. Adopted traces are reused in Section 5. Hence, they are characterized in a later section.

To carry out the experiment, we configure an SSD with multiple channels. These channels are modeled by simulating different striping schemes. Shin et al. [22] enumerated seven striping-based schedule schemes. However, some of the schemes exhibit the same parallelism according to the architecture described in Figure 2. Thus, we review only four schemes, which are presented in Figure 3. The striping schemes in the figure show only two channels. In our experiments, each SSD contains eight channels, with each transmitting data for four chips. A chip contains two dies.

The next issue is the characterization of the unbalanced workloads. The standard deviation (σ) can be adopted to measure the imbalance directly. However, the standard deviation merely reflects the average imbalance. Therefore, we define a new metric. In Formula 1, N is the number of channels, and c_i denotes the number of requests issued to channel i. The formula can be explained as the workload gap between the busiest and the idlest channels divided by the average workload of all channels. This formula characterizes the maximum disparity among channels and is called the maximum deviation (MD). We expect the value of MD to be as small as possible. A small MD means balanced workloads among different channels.

$$MD = \frac{MAX_{i=0}^{N-1}(c_i) - MIN_{j=0}^{N-1}(c_j)}{\sum_{k=1}^{N-1} c_k \Big/ N} \quad (1)$$

Table 1. Unbalanced workloads among channels

Traces	Maximum deviation(MD)				
	Striping 0	Striping 1	Striping 2	Striping 3	Dyna.
Develop	0.061	0.093	0.606	2.286	3.621
Exchange	0.323	0.313	1.213	1.703	1.332
LiveMaps	3.741	7.486	1.961	2.031	0.131
Proj	0.061	0.101	0.272	0.185	0.321
MSN	0.174	0.313	2.994	2.435	1.874
SRC0	0.201	0.286	0.247	0.245	0.103
SRC1	0.255	0.385	0.314	0.294	0.231
USR	0.785	1.730	1.094	2.631	1.870

Traces adopted by the simulations are characterized in Section 5. The experimental results are shown in Table 1. We analyze eight traces with four striping schemes (Figure 3). We draw two conclusions from the results. First, the workloads among the channels are unbalanced. The table shows that most MD values are higher than 30%, with some even exceeding 100%. Second, a one-size-fits-all striping scheme that could guarantee a balanced workload among channels for all traces does not exist. For example, Striping0 arranges balanced workloads for *Develop* (with an MD of 0.061), but fails to arrange balanced workloads for *LiveMaps* (MD 3.741). Striping1 arranges balanced workloads for *Develop*, but arranges unbalanced workloads for *USR*. Therefore, no SSD achieves a balanced workload for all traces, and other measures have to be taken to schedule I/O requests. In this work, we balance workloads through the cache.

An intelligent reader may argue that SSDs usually dynamically allocate write requests to different channels. That is, an idle channel is issued more write requests. Thus, workloads among channels should be balanced. Actually, write requests exert an effect on the workloads of different channels. However, this effect is limited because an SSD is usually equipped with a write buffer. Write requests are accepted by the buffer when the SSD is busy. If a channel is busy with read requests, the SSD either delays the write requests, or assigns the write requests to other channels. Both optimizations fail to eliminate busy channels. We conduct another experiment to demonstrate the imbalance when write requests are taken into account. In the experiment, the SSD dynamically assigns write requests to different channels. We monitor the workload of each channel and compute the MD value. The results are shown in the last column of Table 1. The results show that the workloads remain unbalanced. Dynamic scheduling may generate balanced write workloads. However, whether a piece of data would be frequently accessed is difficult to predict. Therefore, unbalanced read workloads are inevitable.

4. SSD-Aware Cache: SAC

The SAC protects pages from busy channels, and pages from idle channels are evicted. Therefore, workloads among the channels are more balanced. However, two is-

sues need to be addressed: determining whether a page comes from a busy or idle channel, and determining which page would be evicted when a replacement is required. The next two subsections address these problems.

4.1 Page selection from idle channels

The principle of SAC is to replace pages from idle channels with higher-priority. This task requires the identification of pages from idle channels. However, the cache is blind to the channel from which a page originates. I/O requests issued from operating systems are indexed by logical addresses, whereas parallelism relates to physical addresses. For SSDs, logical and physical addresses are dynamically mapped. Hence, SSDs cannot determine the physical channel from which a logical page originates. Furthermore, the SAC is not limited to SSD-based storage systems. The underlying storage devices may be heterogeneous, thereby making the uniform abstraction of channels from these devices difficult. Therefore, each physical channel cannot be directly monitored.

However, I/O requests issued to idle channels usually endure lower latencies. Therefore, if a required page is quickly returned, it can be assumed to come from an idle channel. Hence, we need not to determine whether a channel is busy. Instead, we determine whether a page is from a busy channel. This method appears to require a watermark to divide the pages into two groups: those waiting for a short time (shorter than the watermark) are from idle channels; and those waiting a long time (longer than the watermark) are from busy channels. However, the watermark usually does not exist.

SAC develops a new measure to determine whether a page comes from an idle channel by maintaining a queue. A page joins the queue when the page request is issued by the operating system. Once its data are returned, the page then leaves the queue. A page from an idle channel is likely to wait in the queue for a shorter time. When a page leaves the queue, some pages from busy channels may still be waiting in the queue even though they joined the queue earlier than those from idle channels. In this study, we define a new concept called the *prior page*. If page A joins the queue earlier than page B, then page A is a prior page of B. When a page from an idle channel departs from the queue, some prior pages may still be waiting. The shorter latency a page endures, the higher the number of prior pages still waiting in the queue. Therefore, the number of prior pages (still waiting in the queue) can be used to characterize the latency that a page endures.

Figure 4 illustrates this principle in a visible manner. In Figure 4(a), ten pages are in the queue: three of them had been completed (marked C), whereas the remaining 7 pages (marked W) remain waiting. Then, completed pages will be returned one and another.

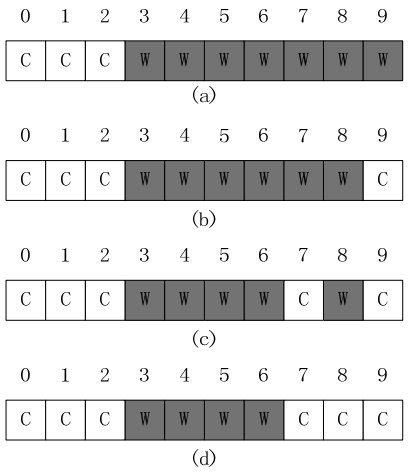

Figure 4. Illustration of the queue. Pages marked with C represent completed requests. Pages marked with W remain to be addressed.

Figure 4(b) shows that Page 9 has been completed, and some prior pages remain waiting in the queue. Thus, Page 9 can be assumed to be from an idle channel. Figure 4(c) shows that the request to Page 7 has been completed. Using the same principle, Page 7 is also assumed to be a page from an idle channel. However, compared with the channel containing Page 9, the channel containing Page 7 is slightly busier because the request to Page 7 is issued earlier but was completed later. Therefore, when a page is returned, and if some prior pages remain waiting in the queue, the page is assumed to be from an idle channel and thus should be replaced with a higher priority. The number of prior pages is used to characterize the priority of the replacement.

The implementation of the policy can be summarized as follows: When a page joins the queue, SAC labels this page with a stamp. Figure 4 shows that each page is labeled by a number. The page with the smallest stamp has waited the longest time and is thus named as the first waiting page. In Figure 4, Page 3 is the first waiting page. When a page leaves the queue, the SAC determines the difference between the stamps of the leaving page and the first waiting page. The difference is used to imitate the number of prior pages still waiting in the queue. For example, when Page 9 in Figure 4 leaves the queue, the difference between its stamp and that of Page 3 is 6. Therefore, the number of uncompleted prior pages is 6.

We can directly adopt the latency for obtaining a page to determine whether the page comes from an idle channel. However, SAC should be able to work over different storage devices, which have different access latencies. Each device needs a threshold to determine whether a page is from an idle channel. However, choosing a uniform threshold for all types of devices is difficult. Nevertheless, our method can model all devices uniformly.

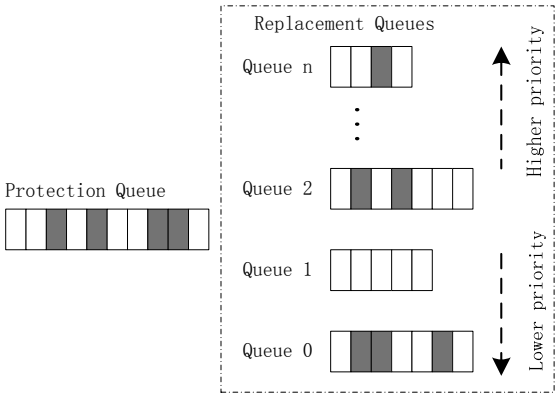

Figure 5. The configuration of SAC. The physical cache is partitioned into two parts. The second part maintains multiple replacement queues. These gray pages have been reused.

4.2 Replacement policy

SAC tries to reduce the average latency of retrieving missed pages. But, the temporal locality is the basis of cache design, thus cannot be negligible. This subsection strike a balance between maintaining pages which will be reused (high benefit) and maintaining pages that are expensive to retrieve (high cost).

We implement SAC based on the replacement policy adopted by Linux [5]. As reviewed in related studies, the policy in Linux maintains two CLOCK queues. The pages evicted from the first queue are moved to the second queue. The pages evicted from the second queue are discarded by the cache ultimately. SAC partitions the physical cache into two parts as well. Figure 5 shows that the first is managed by a protection queue, which functions in the same way as the first Linux queue. This queue protects valuable pages, as its name implies. Replaced pages are selected from the second part, which contains multiple queues.

The last subsection uses the number of uncompleted prior pages to make decision for cache replacement. For simplicity, we can always evict the page with the biggest N, where N is the number of uncompleted prior pages of a given page. This method seems simple, but, it completely ignores the locality. If a page is from an idle channel, it has a high N, and will be replaced soon. However, the page may be re-accessed shortly according to the locality.

To protect recently accessed pages, SAC assigns a priority value to each page in the cache. This priority value is defined as $\log_2 N$, where N is the number of prior pages. Pages in the second part of the cache are grouped into multiple queues according to their priority value. For a page, if the number of prior waiting pages is 8, its priority value is $\log_2 8$; this page then joins Queue 3. Pages within a given queue are ordered in the CLOCK manner. So, the page with the highest N is not immediately replaced because it is located in the MRU end upon joining the CLOCK queue.

```
Routine of Page Request
{
    If the page hits in cache
        Mark it as having been reused.
    Else //miss
        Add it to the Protection Queue.
    End if
    While the Protection Queue is full
        Take the last page of Protection Queue
        If the page has been reused
            Recycle it.
        Else
            Add it to the right queue in Part 2 ac-
    cording to its priority value.
        End if
    End while
    If the cache is full
        Trigger Replacement Subroutine.
}//Routine of Page Request

Replacement Subroutine
{
While the cache is full
    Take the non-empty replacement queue with
highest priority value.
    Take the last page of the selected queue.
    If the page has not been reused
        Evict it.
    Else
        Move it to Protection Queue.
    End if
End while
}// Replacement Subroutine
```

Figure 6. Pseudo code of SAC.

The process of replacement is described as follows: SAC partitions the physical cache into two parts using a watermark. When the first part is full, its least recently used pages are moved to the second part. Each page in the second part then joins the appropriate queue according to its priority value.

Cache replacement is required when the entire cache is full. In the second part, SAC evicts pages in the queue with the highest priority values (e.g., Queue n), followed by Queue n–1, and then by Queue n–2. Notice that, all queues are CLOCK queues rather than LRU queues. Pages marked as being reused must be recycled. The pseudo code is given in Figure 6.

A flaw exists for the cache described above. SAC always evicts pages from the queue with the highest priority value. The queue with the lowest priority (e.g., Queue 0 in Figure 5) is protected. Even though pages in Queue 0 will no longer be accessed, they have no chance to be replaced if there are some pages in a queue with higher priority value. To overcome this drawback, SAC monitors Queue 0 constantly. If Queue 0 has not been replaced for a long time, SAC resizes the two parts of cache. Pages in Queue 0 then can be replaced when the capacity of the second part is reduced.

4.3 Details of Implementation

We implement SAC based on the policy adopted by Linux [5], where the first part occupies one-third of the cache capacity, and the second part occupies the remaining capacity. SAC divides the cache capacity as previously described; however, the two parts may be resized.

As discussed in Subsection 4.1, SAC maintains a queue to compute the priority value for each page. The queue monitors page requests that had been issued to the storage device. When a page receives its data, the queue informs SAC of the number of uncompleted prior pages (denoted as *N*). Subsequently, SAC computes the priority value for the page. The queue has already been implemented in the block device driver module of the Linux kernel. A block device driver defines a structure called *block_IO*, which contains a sequence of bytes. Several block_IOs consist of requests that join a queue waiting to be issued to storage devices. The queue can be reused by SAC for our purpose. In this study, a block_IO contains a page. Several pages make up a request. Although pages in a request are issued to the storage device as a whole, they are returned separately. The SAC computes the priority value for a page according to the status of the pages waiting in the queue.

4.4 Discussion

SAC mostly targets to SSD-based storage systems and assumes that I/O workloads distributed to channels are unbalanced. However, if the system cannot meet the two preconditions, SAC collapses into the policy adopted by Linux, which still has good performance.

Case 1. Workloads are evenly distributed to each channel, or all channels are idle enough to respond to any page requests immediately. All requests in this scenario are completed in the order that they joined the waiting queue. Thus, when a page is returned, no prior waiting page remains, and the priority value is 0. Accordingly, all pages join Queue 0 when they are evicted from the protection queue. There is only one priority queue in the second part. Thereby, SAC collapses to the policy adopted by Linux.

Case 2. The underlying storage device is a hard disk. I/O requests in this scenario are initially reordered by some hard disk-oriented schedulers (e.g., CSCAN [23]). The requests then join the waiting queue. In addition, requests are returned in the same order that they joined the queue. The result is identical to that of Case 1. SAC behaves the same as the Linux policy.

Case 3. The underlying storage devices are RAID or some other heterogeneous devices (e.g., a hybrid system based on SSD and a hard disk). SAC is also effective in this scenario. SAC ignores the underlying physical devices, but abstracts all types of parallelism as channels. These channels are allowed to have different capabilities. For example, if a hard disk spends a long time on serving a request, SAC assumes that it is a channel with low capability. Therefore, pages obtained from this channel are protected from replacement with higher priority.

In conclusion, SAC is designed for SSD-based systems with unbalanced workloads. When SAC is applied to other systems, it collapses into a traditional policy.

5. Performance evaluation

5.1 Experimental setup

We evaluate the performance of SAC via trace-driven simulations. The adopted traces are all enterprise-class workloads that delegate typical applications. *SRC, USR,* and *Proj* were gathered from servers at Microsoft Research Cambridge. *LiveMaps* was gathered from the backend server of LiveMaps for 24 hours. *Develop* was collected from the Developers Tools Release server for a day. *MSNFile* was collected from the MSN Storage file server for 6 hours. *Exchange* was collected from the Exchange Server for 24 hours. All traces can be downloaded from the Internet [24], and had been published by D. Narayanan. [25].

An SSD with eight channels was configured to simulate a parallel storage device. Logical pages among these channels are stripped in a manner presented in Figure 3. Four stripping schemes in the figure are introduced to four different SSDs, identified as SSD0, SSD1, SSD2, and SSD3, respectively. SSDs adopt DFTL [27] to manage the address space. A garbage collector continuously reclaims the block containing the most garbage. The write, read, and erase latencies specified by the flash chip datasheet are 300 μs, 150 μs, and 2 ms, respectively [26]. Furthermore, we simulate SSD4, which dynamically assigns write requests to different channels.

We compare SAC with LRU [3], ARC [4] and the policy adopted by Linux [5]. As a fundamental policy, LRU is extensively referenced. ARC has been widely studied for its simplicity and high performance. The Linux policy, called the Dual Queue because it maintains two CLOCK queues, is a practical scheme. All policies including SAC have constant-time complexities. The spatial complexity of ARC is twice that of the others.

Two situations can occur for a page demanded by operating systems. First, the page hits in the cache. Second, the page has to be fetched from the storage devices. In the first situation, SAC should achieve a hit ratio as high as those of other replacement policies. For the second situation, SAC must guarantee that the average latency is much lower than those of traditional policies. Therefore, the evaluation includes two aspects, namely, the hit ratio of hitting pages, and the access latency of missed pages. These two factors are introduced in the next two subsections. Subsection 5.4 combines the two aspects, whereas the last subsection explains the reason that SAC outperforms the others.

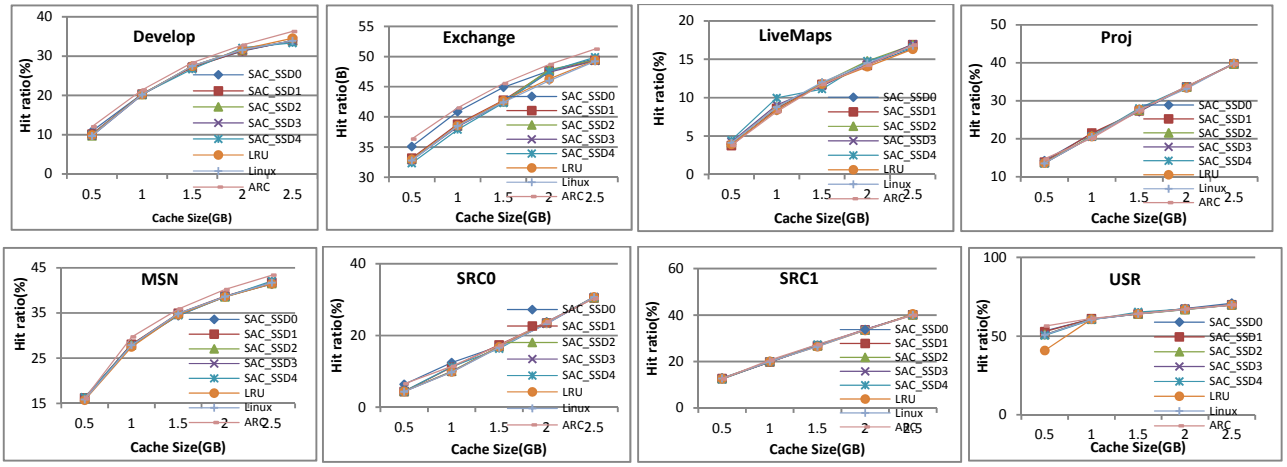

Figure 7. Comparison of hit ratios achieved by different policies. Hit ratios achieved by SAC are comparable with the others.

5.2 Hit ratios achieved by different policies

First, we compare the hit ratios achieved by different policies. The hit ratios achieved by ARC, LRU, and Dual Queue are completely determined by the access sequence. However, the replacement of SAC is related to the access latencies, which in turn are related to the SSD configuration. Thus, for a given access sequence, SAC achieves different hit ratios on different SSDs. Our experiments simulate five SSDs. The hit ratios achieved by SAC on these SSDs are denoted by SAC_SSD0, SAC_SSD1, SAC_SSD2, SAC_SSD3, and SAC_SSD4, respectively. The cache capacity ranges from 0.5 GB to 2.5 GB.

Figure 7 compares the hit ratios achieved by different policies. Generally, no remarkable disparities are found among these policies. Multi-core processors facilitate the concurrent running of multiple threads. These threads interchangeably access the underlying storage devices. The interchangeable workloads exhibit weaker localities. The locality is the fundamental element employed to achieve a higher hit ratio; thus, weaker localities indicate that further improvement of the hit ratio is much difficult. Even these well-known policies (e.g., ARC and LIRS) cannot significantly outperform the most fundamental one (LRU) on workloads with multiple work flows. Instead, the hit ratio is more dependent on the capacity of the cache. Figure 7 shows that as the cache capacity increases, the hit ratios achieved by the different policies continually increase. ARC retains some historical information and thus slightly outperforms the others. In conclusion, SAC can achieve hit ratios as high as those of other policies.

5.3 Average latency for retrieving missed pages

Traditional replacement policies assume that missed pages are identical, and that retrieving these pages requires the same latencies. However, this assumption is inaccurate, particularly in SSD-based systems. The subsequent experiment shows that a cache that considers this fact can remarkably reduce the average latency for retrieving missed pages.

In this experiment, the average latency for retrieving missed pages was used to measure the performance. The cache capacity is 2 GB. The experimental results are shown in Figure 8, which indicates that SAC achieves the lowest waiting time on most workloads.

SAC intends to reduce the waiting time of requests. However, all page requests can be addressed rapidly if the I/O traffic is not heavy. On this scenario, SAC does not significantly outperform other policies. The waiting time of all requests is the same when the traffic is not heavy, and the requests are served in the same order that they are issued to the devices. SAC becomes a traditional policy.

SAC assumes that the I/O workloads are unevenly distributed to channels. Thus, it tries to balance workloads by keeping pages from busy channels in the cache. However, if the workloads are evenly distributed, this optimization becomes ineffective, and SAC collapses into a normal replacement policy. Figure 8 shows that the waiting times achieved by the different policies are nearly the same when the trace *Develop* is simulated on SSD0. This result is due to the even scheduling of the workloads to each channel. Table 1 shows that the maximum deviation of *Develop* on SSD0 is 0.061. A small maximum deviation value indicates balanced workloads among channels.

Therefore, SAC cannot reduce the waiting time when the traffic is not heavy, or when the workloads among channels are balanced. By contrast, SAC dramatically outperforms the other policies if the traffic is heavy, such as when *SRC0* and *MSN* are applied to SSD3. In particular, SAC achieves superior performance when *LiveMaps* is applied to SSD1. In this situation, not only is the traffic heavy (the average waiting time is several milliseconds), but the workloads are also unbalanced (the maximum deviation is 7.486).

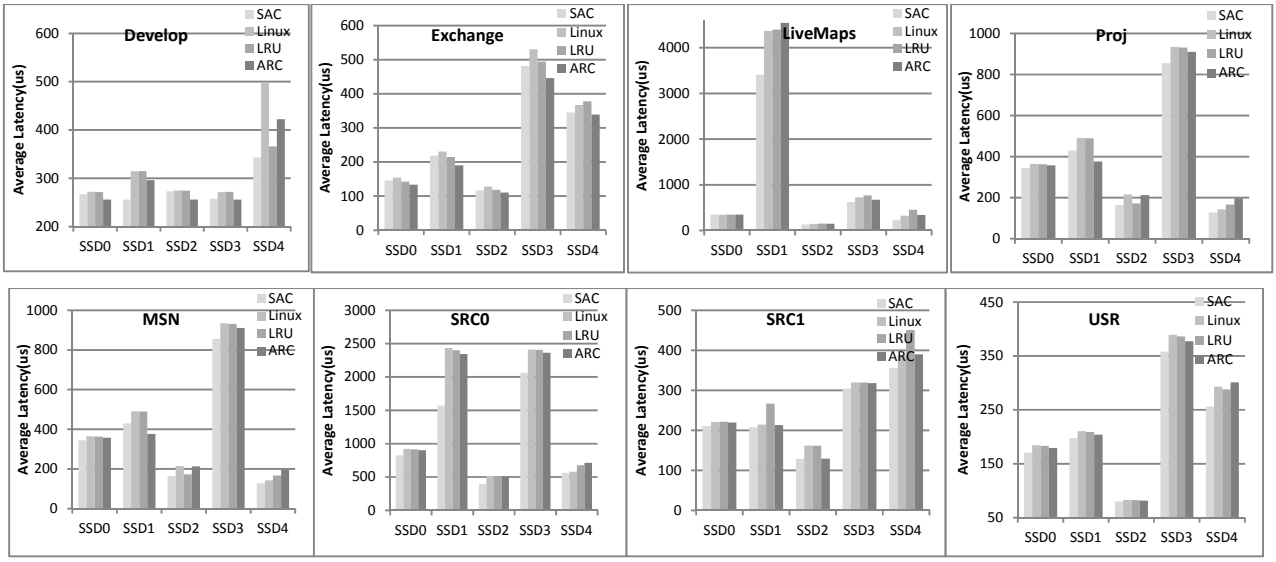

Figure 8. SAC outperforms the other policies on most unbalanced workloads in terms of the average waiting time

Table 1 shows that when *LiveMaps* is scheduled to SSD0, the workloads among channels are unbalanced (the maximum deviation is 3.741). However, SAC does not achieve a lower waiting time compared with the other policies. This result is attributed to the generally light traffic when the trace is replayed on SSD0. At the time, workloads are likely to be evenly distributed when the SSD is busy. This condition again demonstrates that SAC is effective in unbalanced heavy workloads. However, when these preconditions are not met, the performance of SAC is not reduced.

In particular, when SSD4 dynamically schedules write requests to different channels, SAC achieves a lower latency because the imbalance of the read workloads is not significantly affected by the write requests. Although the write workload is balanced, the same may not be true for the read workload.

5.4 Comprehensive comparison

This subsection combines the hit ratios and the latencies of retrieving missed pages to obtain a comprehensive metric that measures the average latency for obtaining all pages (including both hitting and missed pages). Formula 2 defines the metric.

$$AverageLatency = (1 - HitRatio) \times MissedLatency, \quad (2)$$

The latency for SSD is several orders of magnitude higher than that for DRAM. Compared with the SSD latency, the latency for DRAM is negligible. Thus, Formula 2 simply counts the missed pages. The capacity of the cache is also 2 GB. The results are shown in Figure 9. The average latencies shown in Figure 9 are nearly consistent with the latencies shown in Figure 8, indicating that, the average latency mostly depends on MissedLatency, rather than HitRatio. This condition is due to nearly similar hit ratios achieved by all replacement policies (Figure 7).

By contrast, the latencies for retrieving missed pages can be reduced significantly. Figure 8 shows that when a given trace is simulated on four different SSDs, the average latencies significantly vary even though the cache replacement policy is the same. This result implies that the latency highly depends on the schedule scheme. Thus, one promising way of reducing the latency is to reschedule the I/O requests rather than enhance the hit ratio. Among these replacement policies, SAC is the only one that reschedules the workloads among channels, thereby explaining its superior performance over those of the other policies.

5.5 Balanced workloads

The above experiments show that SAC achieves a lower average access latency. The reason for the superior performance of SAC is that it balances the workloads among different channels. To support this argument, we again characterize the workloads among channels. In Section 3 of this paper, we studied the unbalanced workloads among channels without a cache, and the metric maximum deviation was used to characterize the imbalance. In this section, we examine the workloads among channels when the system is equipped with a cache. The results are shown in Table 2. When the system employs a cache with SAC as the replacement policy, the maximum deviation values of the workloads are mostly reduced. Smaller maximum deviation values indicate that the workloads among the channels are more balanced.

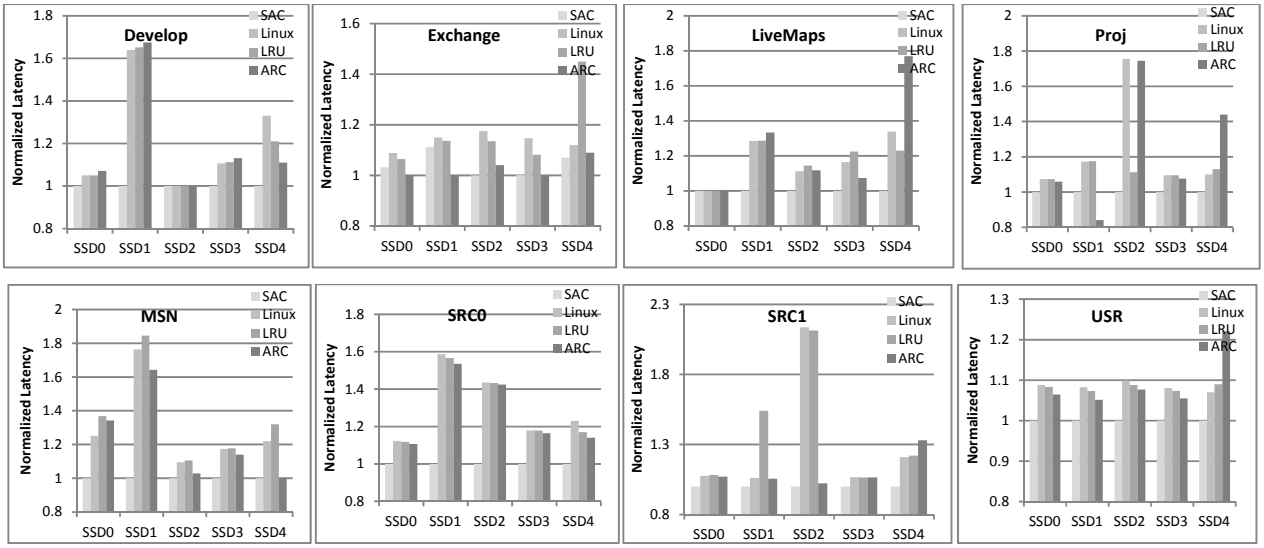

Figure 9. Comprehensive comparison of the access latency.

However, balanced workloads may be induced by two factors. First, SAC balances the workloads, as we have repeatedly stated. Second, the cache itself can balance the workloads. The cache protects hot pages and evicts cold pages, thereby resulting in the rescheduling of the I/O requests among the channels. However, we argue that the effect imposed by the cache is negligible. To support this argument, we examine the workloads among the channels when the system is equipped with a cache, and the cache uses ARC as the replacement policy. The results are presented in Table 3. Generally, ARC achieves a higher hit ratio than SAC. Thus, the cache with ARC as the replacement policy should obtain more balanced workloads. However, the results show that ARC does not produce more balanced workloads, implying that the effect of the cache on the workloads is negligible even though the hit ratio is high. Therefore, the balanced workloads are generally induced by the SAC.

6. Conclusions

SSDs have been extensively used in storage systems. However, existing operating systems still regard these underlying storage devices as hard disks. Thus, the advantages of SSDs are not completely exploited. In this paper, we redesign a cache replacement policy called the SAC, which mostly targets SSD-based storage systems. SAC focuses on the intrinsic parallelism of SSDs, and abstracts the parallelism as virtual channels. Workloads among channels are usually unbalanced. Therefore, SAC reschedules the workloads among these channels to achieve a lower response time. Two preconditions should be met for SAC to achieve high performance: heavy I/O traffic, and unbalanced workloads. If these preconditions are not met, SAC behaves the same as traditional policies.

The SAC is designed for SSD-based systems, but is also applicable to any storage systems exhibiting parallelism, such as RAIDs. SAC would still work even if the underlying storage device is a single hard disk, which does not exhibit parallelism. In this scenario, SAC collapses into a traditional cache replacement policy.

Table 2. Unbalanced workloads among different channels when the storage system employs SAC as the cache replacement policy. The workloads are more balanced compared with the original workloads when the system does not employ a cache.

Traces	Maximum deviation (MD)			
	Striping 0	Striping 1	Striping 2	Striping 3
Develop	0.07	0.01	0.28	2.06
Exchange	0.01	0.01	1.11	1.07
LiveMaps	2.73	4.46	1.06	2.07
Proj	0.04	0.07	0.08	0.06
MSN	0.17	0.22	2.14	2.52
SRC0	0.14	0.20	0.08	0.07
SRC1	0.17	0.29	0.18	0.15
USR	0.17	0.25	0.13	0.09

Table 3. Unbalanced workloads among different channels when the storage system employs ARC as the cache replacement policy. ARC does not achieve as good performance as SAC does.

Traces	Maximum deviation (MD)			
	Striping 0	Striping 1	Striping 2	Striping 3
Develop	0.07	0.10	0.47	2.28
Exchange	0.01	0.01	1.39	2.11
LiveMaps	3.93	7.48	1.96	2.10
Proj	0.03	0.06	0.16	0.03
MSN	0.17	0.29	2.97	2.60
SRC0	0.19	0.26	0.14	0.10
SRC1	0.15	0.26	0.23	0.15
USR	0.19	0.25	0.22	0.18

Acknowledgments

We are grateful to our anonymous reviewers and to our shepherd, Peter Desnoyers, for their suggestions. We also appreciate Prof. Hong Jiang for his help to polish this paper. This work was supported by the National Natural Science Foundation of China under NSFC61025009, NSFC61070198, NSFC61170288, NSFC60903040, and the Program for New Century Excellent Talents in University under NCET-08-0145.

References

[1] MICROSOFT. 2007. Windows Vista's features explained. http://www.microsoft.com/windows/products/windowsvista/features/details/performance.mspx.

[2] J. Matthews, S. N. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. ACM Transactions on Storage, 4:(2), May 2008.

[3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. In IBM Systems Journal, pages 78-101, 1966.

[4] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In Proceedings of 2nd USENIX Conference on File and Storage Technologies, pages 115-130. San Francisco, CA, USA, Mar 2003.

[5] M. Gorman, Understanding the Linux Virtual Memory Manager, Prentice Hall, April, 2004.

[6] F. J. Corbato. A paging experiment with the Multics system. In MIT Project MAC Report MAC-M-384, May 1968.

[7] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In Proceedings of the ACM SIGMOD international Conference on Management of data, pages 297-306. Washington, D.C., USA, May 1993.

[8] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In Proceedings of the Twentieth International Conference on Very Large Databases, pages 439-450. Santiago de Chile, Chile, USA, September 1994.

[9] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In Proceedings of the 2001 Annual USENIX Technical Conference, pages 91-104. Boston, Massachusetts, USA, June 2001.

[10] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In Proceedings of the 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 31-42. Marina Del Rey, California, USA, June 2002.

[11] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In Proceedings of the 33th international symposium on Computer architecture 2006 (ISCA2006), pages 167-178. Boston, MA, USA, June 17-21, 2006.

[12] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In Proceedings of the 4th USENIX Conference on File and Storage Technologies, pages 1-16. San Francisco, CA, USA, December 2005.

[13] S. Wan, Q. Cao, J. Huang, S. Li, X. Li, S. Zhan, L. Yu, and C. Xie. Victim Disk First: An Asymmetric Cache to Boost the Performance of Disk Arrays under Faulty Conditions. In Proceedings of the USENIX Annual Technical Conference 2011 (ATC2011), pages 173-186. Portland, OR, USA, June 15-17, 2011.

[14] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS2009), pages 217-228. Washington, DC, USA, 2009.

[15] Y. J. Seong, E. H. Nam, J. H. Yoon, et al. Hydra: A block-mapped parallel flash memory solid-state disk architecture. IEEE Transaction on Computer, 59: 905-921, 2010.

[16] N. Xiao, Z. G. Chen, F. Liu, M. C. Lai and L. F. An. P3Stor: A parallel, durable flash-based SSD for enterprise-scale storage systems. Science China Information Science, 54: 1129-1141, 2011.

[17] C. Dirik and B. Jacob, The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization.In Proceedings of the 36th annual international symposium on Computer architecture, pages 279-289. New York, NY, USA, 2009.

[18] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, Design tradeoffs for SSD performance. InProceedings of USENIX Annual Technical Conference2008, pages 57-70. Boston, USA, June 22-27, 2008.

[19] Y. Hu, H. Jiang, L. Tian, H. Luo and D. Feng. Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity. In Proceedings of the 25th International Conference on Supercomputing (ICS2011), pages 96-107. Loews Ventana Canyon Resort, Tucson, Arizona, USA, May 31-June 4, 2011.

[20] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In Proceedings of high performance computer architecture 2011 (HPCA2011), pages 266-277. SanAntonio, TX, February 2011.

[21] F. Chen, R. Lee, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In Proceedings of the 2009 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 181-192. Seattle, WA, June 15-19, 2009.

[22] J. Y. Shin, Z. Xia, N. Y. Xu, R. Gao, X. F. Cai, S. Maeng, and F. H. Hsu. FTL design exploration in reconfigurable high-performance SSD for server application. InProceedings of International Conference on Supercomputing2009, pages 338-349.New York, USA, June 8-12, 2009.

[23] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithm for modern disk drives.In Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 241-251. New York, NY, USA, 1994.

[24] http://iotta.snia.org/traces.

[25] D. Narayanan, A. Donnelly and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In Proceedings of the 6th USENIX Conference on File and Storage Technologies, pages 253-267. San Jose, CA, USA, February 26-29, 2008.

[26] MT29F8G08MAAWC datasheet. www.micron.com.

[27] A.Gupta, Y. Kim Y, and B.Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In: Proceedings of the 14th international conference on Architectural Support for ProgrammingLanguages and Operating Systems (ASPLOS09), pages 229-240.Washington, DC, USA, 2009.