

An Adaptive Write Buffer Management Scheme for Flash-Based SSDs

GUANYING WU and XUBIN HE, Virginia Commonwealth University
BEN ECKART, Carnegie Mellon University

Solid State Drives (SSD's) have shown promise to be a candidate to replace traditional hard disk drives. The benefits of SSD's over HDD's include better durability, higher performance, and lower power consumption, but due to certain physical characteristics of NAND flash, which comprise SSD's, there are some challenging areas of improvement and further research. We focus on the layout and management of the small amount of RAM that serves as a cache between the SSD and the system that uses it. Of the techniques that have previously been proposed to manage this cache, we identify several sources of inefficient cache space management due to the way pages are clustered in blocks and the limited replacement policy. We find that in many traces hot pages reside in otherwise cold blocks, and that the spatial locality of most clusters can be fully exploited in a limited time period, so we develop a hybrid page/block architecture along with an advanced replacement policy, called BPAC, or Block-Page Adaptive Cache, to exploit both temporal and spatial locality. Our technique involves adaptively partitioning the SSD on-disk cache to separately hold pages with high temporal locality in a page list and clusters of pages with low temporal but high spatial locality in a block list. In addition, we have developed a novel mechanism for flash-based SSD's to characterize the spatial locality of the disk I/O workload and an approach to dynamically identify the set of low spatial locality clusters. We run trace-driven simulations to verify our design and find that it outperforms other popular flash-aware cache schemes under different workloads. For instance, compared to a popular flash aware cache algorithm BPLRU, BPAC reduces the number of cache evictions by up to 79.6% and 34% on average.

Categories and Subject Descriptors: D.4.2 [Operating Systems]: Storage Management

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: SSD, NAND flash memory, flash-aware cache, write buffer

ACM Reference Format:

Wu, G., He, X., and Eckart, B. 2012. An adaptive write buffer management scheme for flash-based SSDs. *ACM Trans. Storage* 8, 1, Article 1 (February 2012), 24 pages.
DOI = 10.1145/2093139.2093140 <http://doi.acm.org/10.1145/2093139.2093140>

1. INTRODUCTION

Solid state drives (SSD's) are set to supplant traditional hard disk drives (HDD's) in nearly every domain of storage computing, from server applications to home desktops

A preliminary version of this work was published as "BPAC: An adaptive write buffer management scheme for flash-based solid state drives." In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*, IEEE.

This research was supported by the U.S. National Science Foundation (NSF) under Grant Nos. CCF-1102605, CCF-1102624, and CNS-1102629. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agency.

Authors' addresses: G. Wu and X. He, Monroe Park Campus, School of Engineering, West Hall, 601 W Main Street, P.O. Box 843072, Richmond, VA 23284-3072; email: {wug, xhe2}@vcu.edu; B. Eckart, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890; email: eckart@cmu.edu. Correspondence email: xhe2@vcu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1553-3077/2012/02-ART1 \$10.00

DOI 10.1145/2093139.2093140 <http://doi.acm.org/10.1145/2093139.2093140>

Table I. Compare SSD and HDD [Western Digital 2008; Intel 2009; Shimpi 2009]

	Intel X-25M G2 SSD	Western Digital VelociRaptor HDD
Technology	NAND MLC Flash	10k RPM Platters
Capacity (GB)	160	300
Cache Size (MB)	32	16
Energy Consumption (Idle/Maximum Watts)	0.15/0.75	4.53/6.08
Operating Temperature (C)	0-70	5-55
MTBF (hours)	1,200,00	1,400,00
Shock Resistance	1500 g, 0.5 ms	300 g, 2 ms
Latency (μ s)	65-85	3000
Acoustics (dBA)	0	29-34
Sequential R/W (MB/s)	257.5/78.1	120.8/120.2
Random R/W (MB/s)	58.5/34.5	0.68/1.59
Price (USD)	440 [Newegg 2009a]	230 [Newegg 2009b]

to MP3 players. These drives are predominantly made from banks of NAND flash memory, and though significantly different from platter drives, are exported to the OS as simple block devices. As prices have dropped over 100x in the last 5 years [Mason 2009], consumer interest is growing in many markets since SSD's have many inherent benefits over HDD's.

From a reliability standpoint, solid state drives have no moving parts, no mechanical wearout, and are silent, heat resistant, and shock resistant. Solid state drives also exhibit astoundingly good performance for synthetic random workloads when compared with traditional hard drives, and equivalent or better performance for sequential workloads. Though the price per Gigabyte remains in the HDD's favor, the gap is somewhat narrowing. Table I shows some of the more salient differences between the two storage devices. Compared in this table are two high-performance drives, representative of the current top-of-the-line consumer-level offerings from both technologies. The Western Digital VelociRaptor [Western Digital 2008] is a 10,000 RPM hard disk drive aimed at the home enthusiast market and is one of the highest benchmarking drives at its price point. The Intel X-25M [Intel 2009] has the potential for a much larger niche, being eminently important for OLTP and other server applications due to its outstanding random I/O performance. On the Intel SSD, random reads are roughly 86 times faster and random writes are about 21 times faster when compared to the Western Digital HDD. Clearly, solid state drives represent a fundamental leap in performance at the storage level.

The swift and almost inevitable rise to ubiquity notwithstanding, SSD's do suffer from several performance quirks arising from the physical nature of NAND flash and architectural constraints of their controllers. The most notable problems include: the inability to modify data in-place, read/write performance asymmetry, and slow and constrained erase functionality. There have been a wealth of techniques developed to circumvent these issues, including work at the Flash Translation Layer (FTL), new caching mechanisms, and new ways to exploit the parallelism of the flash device. Our work presented in this article falls into the area of caching.

The on-disk write cache works as a buffer in-between the disk interface and the FTL. The actual commitment of writes, which happens at buffer replacement, is the major cause of the write latency. To be specific, upon a write request, the FTL writes the new data on a clean flash page and marks the old data as invalid; when there are insufficient clean pages, the FTL executes a process commonly referred to as "garbage collection" in order to free the flash pages with invalid data. Thus, a good flash-aware write cache scheme should absorb repeated writes, so as to reducing the number of cache destages, and should reform random workloads and export them to the FTL as

sequential, which minimizes the chance or the overhead of expensive garbage collection processes. Among the various *flash-aware* disk cache management schemes in the literature, BPLRU [Kim and Ahn 2008], FAB [Jo et al. 2006], and CLC [Kang et al. 2009] are the most representative. By carefully examining the ways these schemes organize data structures in the cache and the schemes they use to perform cache replacement, we found that there exist a few problems that may cause inefficient use of the cache space, which can result in unnecessary cache destages and low level of spatial locality.

In this paper, we present a novel cache management scheme, called Block-Page Adaptive Cache (BPAC), for buffering *write* requests to the SSD in the small RAM portion of the drive. Specifically, our contributions include the following.

- A new cache data structure, *Dual-list*, is proposed to partition the SSD on-disk cache space into a page-based list and a block-based list for buffering write requests, and we show that this particular architecture uses cache space more efficiently compared to the pure block-based list, which is commonly used in existing schemes.
- We develop a new metric (BIRD) to evaluate the spatial locality of SSD I/O workload.
- We present an approach adaptive to different workloads to dynamically differentiate the low spatial locality clusters from high spatial locality ones. For the former ones, we develop a replacement policy that makes differential treatments based on access patterns. We find that the combined architecture and replacement policy achieves better performance compared to existing flash-aware schemes. For instance, our simulation experiments show that, compared to BPLRU, BPAC reduces the number of cache evictions by up to 79.6% and increases the average destage size of cache evictions by up to 350%.

The rest of the article is organized as follows. Section 2 gives a brief overview of the physical design of SSD's along with the architectural and major software techniques used to ameliorate its shortcomings. In Section 3, we discuss the potential problems of previous flash-aware cache schemes and derive our BPAC's design guidelines. In Section 4 we discuss BPAC's design in detail. We analyze its performance with respect to other proposed flash-aware caching policies in Section 5, and summarize popular traditional cache policies and the flash-aware ones as well as the other SSD-related techniques in Section 6, and then conclude with our final comments in Section 7.

2. BACKGROUND

2.1. NAND Flash and SSD Architecture

In general, the data retention of NAND flash memory is done by the charge trapped in the floating gate of the flash cell, and the amount of charge determines the logical level of a certain cell. According to the maximum number of levels defined when the data are retrieved, there are two primary types of NAND flash memory: Single-level cell (SLC) and Multi-level cell (MLC). As one would expect, single-level cell flash stores one bit per transistor, while multi-level cell flash stores multiple bits per transistor. MLC is one of the efforts made for increasing the storage density of the flash. Among the others, the 20nm technologies [Intel 2010; Eetimes 2010; Engadget 2010] that have merged this year (2010) push the capacity per chip up to 32GB.

The NAND flash by itself exhibits relatively poor performance. The high performance of an SSD comes from leveraging a hierarchy of parallelism. At the lowest level is the *page*. I/O read and write requests are sent in terms in pages, which are typically on the order of 4 kB. Erase operations operate at the *block* level, which are sequential groupings of pages. A typical value for the size of a block is 64 or 128 pages. Further up the hierarchy is the plane, and on a single die there could be several planes. Planes

Table II. Parameters of a Samsung 4 GB Flash Module [Agrawal et al. 2008]

Page Read to Register	25 μ s
Page Program from Register	200 μ s
Block Erase	1.5 ms
Serial Access to Register	100 μ s

operate semi-independently, offering potential speed-ups if data is striped across several planes. Additionally, certain copy operations can operate between planes without crossing the I/O pins.

An upper level of abstraction, the chip interfaces, free the SSD controller from the analog processes of the basic operations, that is, read, program, and erase, with a set of defined commands. NAND interface standards includes ONFI [ONFI 2010], BANAND [ONFI 2010], OneNAND [Samsung 2010], LBA-NAND [Toshiba 2010], etc.

Although SSD's show some promise including good read performance and low power consumption, they do have inherent drawbacks resulting from NAND flash architecture, particularly the slow erase times at block-level granularity, lack of overwrite capabilities, read/write asymmetry, and wear-out from repeated accesses. As shown in Table II, an erase operation is over 7 times slower than a write operation; reading to register is 8 times faster than programming a register. To address these challenges, there have been many approaches working on the garbage collection process, wear-leveling, logical to physical mapping scheme, etc., as well the on-disk cache management scheme. Two of these important techniques, FTL and flash-aware cache schemes, are described in next two sections.

2.2. Flash Translation Layer

An SSD exports itself as a block device by adopting a software layer called Flash Translation Layer (FTL) in between the host interface and raw flash memory. FTL is a key component of an SSD in that it not only is responsible for managing the “logical to physical” address mapping, but it also works as a flash memory allocator, wear-leveler, and garbage collection engine. The two functionalities most related to on-disk buffer schemes are mapping and garbage collection; we discuss both subjects in this section.

2.2.1. Mapping Schemes. The mapping schemes of FTL's can be classified into two types: page-level mapping, with which a logical page can be placed onto any physical page; or block-level mapping, with which the logical page LBA is translated to a physical block address and the offset of that page in the block. Since with block-level mapping, one logical block corresponds to one physical block, we refer to a *logical block on a physical block* as a *data block*. As the most commonly used mapping scheme, *Log-block FTL's* [Rosenblum and Ousterhout 1992] reserve a number of physical blocks that are not externally visible for logging pages of updated data. In log-block FTL's, block-level mapping is used for the data blocks, while page-level mapping is for the log blocks. According to the block association policy (how many data blocks can share a log block), there are mainly three schemes, *block-associative sector translation* (BAST) [Kim et al. 2002], *fully-associative sector translation* (FAST) [Lee et al. 2005], and *set-associative sector translation* (SAST) [Kang et al. 2006]. In BAST, a log block is assigned exclusively to one data block; in FAST, a log block can be shared among several data blocks; SAST assigns a set of data blocks to a set of log blocks.

2.2.2. Garbage Collection Process. In the context of log-block FTL's, when free log blocks are not sufficient, the *garbage collection* process is executed, which merges clean pages on both the log block and data block together to form a data block full of clean pages. Normally this process involves the following routine: read clean pages from the log

Table III. Overhead Difference among Full Merge, Partial Merge and Switch Merge

	Full merge	Partial merge	Switch merge
Clean page reading	N	N_c	0
Page programming	N	N_c	0
Block erase	2	1	1

N stands for the number of pages per block; N_c means the number of clean pages in the data block.

block and the corresponding data block(s) and form a data block in the buffer; erase the data block(s) and log block; program the data on a clean physical block (block that contains no data at all). Sometimes the process can be quite simplified: if we consider a log block that contains all the clean pages of an old data block, the log block can just replace the old data block; the old data block can be erased, making one clean physical block. We refer to the normal process as *full merge* and the simplified one as *switch merge*. A *Partial merge* happens when the log block contains only (but not all) clean pages of one data block, and the garbage collection process only requires that the rest of the clean pages get copied from the data block to the log block. Afterwards, the log block is then marked as the new data block and the old data block gets erased.

To make a quantitative view of the overhead of different merge routines, Table III compares the numbers of clean page reading, page programming, and block erase, which are involved in garbage collection routine of the BAST FTL. The former two are in the order of number of pages, and the last one is in number of blocks.

2.3. Flash-Aware Cache Schemes

The FTL presents an SSD with an HDD interface to the host system and conducts garbage collection, wear-leveling, and mapping, all of which cause additional overhead. To address the extra complexity of the FTL as well as the read/write asymmetry and the erase penalty, an on-disk cache is needed that takes SSD features into account. Due to read/write asymmetry, where reads are much faster than writes, it is considered more cost-effective to dedicate the small space of on-disk buffer exclusively for writes to reduce overall latency. Among the various flash-aware cache schemes, BPLRU, FAB, and CLC are the most representative ones:

BPLRU or block-level LRU [Kim and Ahn 2008]. This scheme is proposed to exploit the spatial locality of the workload by grouping pages that belong to the same data blocks (the following two schemes do this as well) into page clusters and ordering the clusters in an LRU fashion; the recency of a cluster depends on the most recently accessed page of the cluster. BPLRU's replacement policy is simple: the clusters on the very end of the LRU list are considered to have the least temporal and spatial locality, and as such, evicting such clusters is reasonable. BPLRU also detects sequential pattern clusters and evicts them as soon as cache space is needed.

FAB or Largest Cluster [Jo et al. 2006]. This scheme maintains a list of page clusters sorted by their size in the cache, and the largest cluster is always the replacement victim. Using this replacement policy, more cache space is expected to be made while it is needed by the new pages, and thus the number of cache destages can be reduced.

CLC or Coldest and Largest Cluster [Kang et al. 2009]. It is a mixture of block-level LRU and FAB: the cluster list is manually partitioned into a "size-independent" region for clusters of high locality, which are ordered in an LRU fashion; a "size-dependent" region of low locality clusters ordered by the size. Clusters get evicted only if they are the largest of the clusters in the size-dependent region. The ratio of the number of clusters in the size-independent region to the total number of clusters is denoted as α . For comparison purposes, these same terms are used in this article.

3. MOTIVATION AND DESIGN GUIDELINES

In this section, we discuss the potential problems of the popular flash-aware on-disk cache schemes. We find most problems are rooted in the ways that locality is exploited. We conclude this section with a list of guidelines for our cache scheme design.

3.1. Potential Problems of Flash-Aware Schemes

Pure Block-Based List. With this data structure, both temporal and spatial localities determine the recency of clusters. Consider a cluster containing a small number of hot pages and mostly cold ones (for single pages, hot/cold refer to the temporal locality). The spatial locality causes pages to be grouped together, while the temporal locality, which is represented by the repeated accesses to the hot pages, will keep updating the recency of a cluster. Thus, the cold pages will stay in the cache with the hot ones if they belong to the same block, causing a waste of cache space.

Early Eviction. One major problem of FAB is that it does not protect the clusters that have high temporal or spatial locality from being evicted. Take the spatial locality for an example: In a sequential access stream $\{64, 65, 66, \dots, 127\}$, assuming a data block is 64 pages (we hold this assumption all throughout this article), a cluster of block #1 is formed; if this cluster is evicted before the page 127 is added-in, one new cluster of the same block will be formed by the rest pages. We call this situation an early eviction. Avoiding this condition makes the FTL log the pages in the cluster in one log block, and the chance of switch merge increases. But for early eviction, the FTL may log it in one log block, and if the garbage collection process is executed on this log block before FTL logs the subsequently formed cluster in the same log block, a costly partial or full merge is inevitable. For the FAB scheme, when the cache size is small and the spatial locality of the clusters can not be fully exploited, FAB's replacement policy of picking up the *largest* cluster would cause early evictions [Jo et al. 2006; Kang et al. 2009].

Efficiency of the LRU Replacement Policy. This is one major drawback of BPLRU. Considering the LRU-order cluster list (BPLRU), the set of clusters in the least recently used region (tail) may be assumed to have the same "coldness" and the temporal and spatial locality can be considered minimal. Thus, selecting the largest one in such a region can make more room than selecting the cluster on the exact end of the list, for future residents as well as the hot ones on the MRU (Most Recently Used) region, without compromising the hit ratio or causing early evictions. This situation is also the reason why CLC applies the *Largest Cluster* policy on its size-dependent region.

Non-adaptiveness in the partitioning scheme. By experimental trials with the workload, the CLC scheme finds out the optimal proportion $(1 - \alpha)$ of the size-dependent region upon which the largest cluster policy is applied; and the tuned α remains invariant throughout the workload. As said in Kang et al. [2009], accurately identifying the size-dependent region is crucial: if its size is smaller than the optimal, CLC regresses towards BPLRU; if its size is larger than the optimal, CLC will have the early eviction problem as FAB does. As we will show in Section 5, the optimal proportion depends on the workload as well as the size of cache space, and thus deciding it with experimental trials is not applicable in real-world practice.

3.2. Design Guidelines

Given the above potential problems we have described, we have the following design guidelines to be used in the development of BPAC.

Decouple the Temporal Locality with the Spatial Locality. To solve the problem of the pure block-based data structure, we intend to detach the hot pages with the cold ones, so that the two localities can be treated separately.

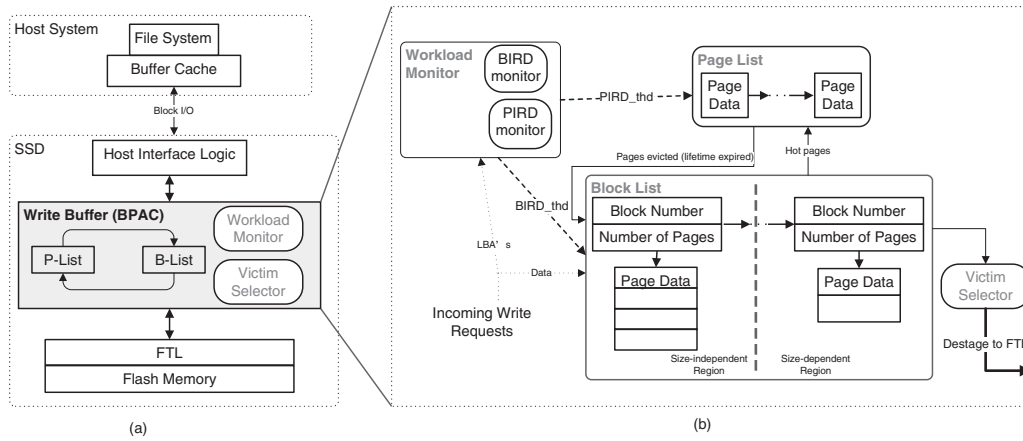


Fig. 1. BPAC overview.

Automatically Determine the Size-Dependent Region. In order to apply the largest cluster replacement policy on the clusters of which the spatial locality is already fully exploited, we need to first identify those clusters (the size-dependent region) with a dynamic method of estimating the spatial locality, instead of manually tuning the parameters as in CLC.

Differential Treatments for Various Access Patterns. There are mainly three kind of access patterns: sequential, looping and random, which show different characteristics on the spatial locality. Sequential patterns have a short inter-reference distance. Once a sequential cluster is “full” (containing all pages of the corresponding data block), its spatial locality can be considered fully exploited. Looping patterns are simply repeated sequential patterns, but they are different from hot pages in the random pattern in that the repeatedly accessed pages of a looping pattern have the same *hotness*, and due to its high spatial locality, a looping pattern should be considered as a whole. A random pattern’s spatial locality is less predictable, so a method of measuring and estimating its spatial locality is needed.

4. BPAC ARCHITECTURE AND DESIGN

In this section, we follow our guidelines to propose a hybrid adaptive write buffer management scheme for SSD’s called BPAC (Block-Page Adaptive Cache).

An overview of the system structure is depicted in Figure 1(a). The SSD’s host interface receives block I/O requests from the host system, the BPAC scheme takes charge of buffering the write requests and managing the cache, and beneath BPAC, the FTL receives the cache destages from BPAC and handles the actual page writes on flash memory. Specifically, we propose the following three techniques of BPAC.

- To decouple temporal and spatial locality, we adopt a new data structure, *dual-list*, which consists of a page-based list (p-list) to hold the hot pages for exploiting the temporal locality and a block-level list (b-list), in which pages of the same data block get grouped into clusters, to exploit the spatial locality. See Figure 1(a) for details.
- We derive the duration of the temporal and spatial locality of pages and clusters by an online *workload monitor*. We use this method and its estimations to automatically partition the cache space, solving the nonadaptiveness problem that exists in CLC.
- The *victim selector* detects various access patterns and applies different replacement policies based on the detected pattern.

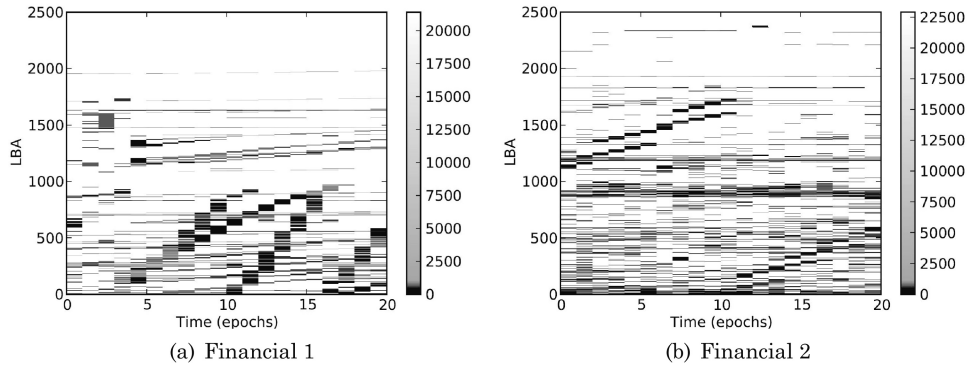


Fig. 2. X axis is time grouped into 20 epochs, Y axis is block number. The “hotter” a block is, the lower its average temporal distance, which is reflected by the gray scale (darker).

4.1. Workload Analysis and the Motivation for Dual-List

We wish to know if clustering solely by block, as in BPLRU, leads to poor cache utilization. If a cold page gets accessed while a hot page from the same block is in the cache (or vice versa), the cold page has the danger of being “dragged along” by the nature of pure block-based caches. For our analysis, we look at two popular traces, Financial 1 and 2 (F1, F2) from the UMASS Trace Repository [UMASS 2007], which are large OLTP traces having write instructions on the order of 1 million.

To verify our assumption of poor cache utilization, we need to look at the “heat distribution” of a representative block. We calculated the temporal locality at block level, and plotted a 2-dimensional histogram in Figure 2, showing the “heat” of blocks over epochs of time. The horizontal lines represent blocks with high locality regardless of time. The diagonal lines are series of sequential accesses, which have low temporal locality, as shown by the gray scale in one epoch and the absence of that gray scale in the next epoch. MRU (Most Recently Used) is the optimal policy for low locality sequential accesses. Also, the parallel diagonal lines represent looping patterns. The rest of the accesses (the dots) are blocks with high locality at certain times and low locality at other times. LRU can be applied effectively to these types of accesses. The heat maps show that both traces offer a mix of different types of locality.

To avoid to the task of setting an arbitrary point at which to call a page hot or cold, we found it useful to rank pages by hotness. Thus, we sidestep the issue since it is easier to say that one page is hotter than another page, instead of trying to define a page’s hotness in some objective, universal sense. We decided to rank by both average temporal distance and by frequency. Ranking by temporal distance means that the page with the lowest average temporal distance between accesses is ranked first, and the page with the highest reuse distance is ranked last. Frequency is more straightforward, with pages ranked by how frequently they were used in the trace.

We decided to investigate how many unique blocks the hottest pages use up. If hot pages reside in otherwise cold blocks, then the hottest pages will be spread out over many different blocks. If not, relatively few blocks will house the hottest pages. Thus, if we graph how many unique blocks are present for the x hottest pages, we can get an accurate depiction of how spread out the heat is, and if clustering by block will keep unwanted cold pages in the cache.

From Figures 3(a) and 3(b), we can see that hot pages are spread out among many different blocks. The hottest pages reside in many blocks, shown by the steepness of the curves at the leftmost part of the graphs. Cold blocks are spread out over the same blocks that contain hot pages, shown by the flat regions on the middle and rightmost

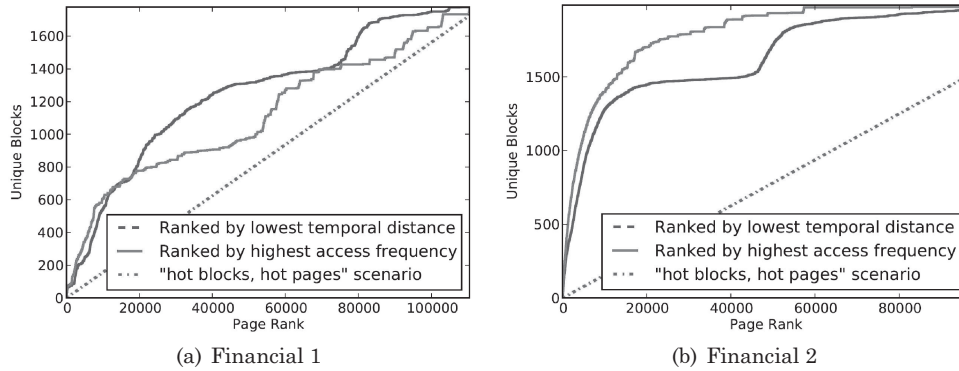


Fig. 3. Number of unique blocks vs. ranked pages. Hottest pages reside in many blocks, shown by the steepness of the curve at the leftmost part of the graph. Cold blocks are spread out over the same blocks that contain hot pages, shown by the flat regions on the middle and rightmost parts of the graph. The “hot blocks, hot pages” scenario represents an ideal case where the hot pages reside in a small number of blocks, which is opposite to the actual observation represented by the blue and green curves.

parts of the graphs. Grouping by block only will bring along unwanted cold pages, since very few hot pages actually reside in each block. It is clear that a scheme is needed that can detach hot pages with cold pages yet still evict by cluster.

4.2. BPAC Data Structure: The Dual-List

In our BPAC scheme, the cache space is partitioned into two parts, as shown in Figure 1(b), one for the p-list consisted of single pages, and the other for the b-list of page clusters. P-list serves as the repository of pages with high temporal locality (hot) ordered in an LRU fashion; b-list holds pages with low temporal locality (cold). The b-list is further divided into size-independent (clusters with high spatial locality) and size-dependent (clusters with low spatial locality) regions, in which the positions of clusters are determined by recency and size, respectively.

Due to the fact that new incoming pages’ (cache misses) “hotness” is unknown, they are first accommodated in the b-list, in which pages are always merged together into a cluster if they belong to the same data block. It is commonly held that if a page is accessed more than once, its temporal locality tends to be much higher than pages accessed only once [Karedla et al. 1994]. Thus, in our design we consider the “second” access (the first hit) as a sign of hotness. If a particular page in the b-list is accessed again, this page is moved to the p-list. When a page’s “lifetime” of staying in the p-list (the mechanism determining the lifetime is discussed in the next part) expires, it is moved back to the b-list.

While the recencies of pages in the p-list are determined by their hits, the recencies of clusters in the b-list are affected by both newly added pages or the “first hit” of the hot pages of a cluster. Since BPAC keeps the hot pages in the p-list, the repeated (the “second hit” and so on) hits are expected to happen mostly in p-list. Thus, the recency of a cluster is determined by the first hit pages (spatial locality).

4.3. Flash-Aware Locality Metrics

For SSD’s, assuming a block-level mapping scheme common to most FTL’s, the spatial locality of page accesses on different data blocks can be considered to be minimal. So for a flash-aware cache scheme design, it is important to keep in mind that only the spatial locality among pages of the same data block (*intra-block*) is meaningful. Hence we introduce a new measurement *BIRD* or Block-level Inter-Reference Distance, to

describe the spatial locality for SSD's. In the context of flash devices, *spatial locality* is about how close in virtual time the references are to the *unique* pages in the same data block. We refer to this block-level inter-reference distance as *BIRD*. For example, if two consecutive references to block #0 happens at time 10 and 34, respectively, then this BIRD is $(34 - 10 - 1) = 23$; in an example access sequence $\{10, 13, 13, 4, 5, 8, 70, 75, 10, 64\}$ (LBA's), pages from two blocks (block #0 and #1) are accessed. For block #0, its BIRD's are $\{0 (10, 13), 0 (13, 4), 0 (4, 5), 0 (5, 8), 2 (8, 10)\}$; for block #1, its BIRD's are $\{0 (70, 75), 1 (75, 64)\}$. Note in Section 4.2, with dual-list, a cluster's recency mainly depends on the spatial locality, which is measured by BIRD.

To measure *temporal locality*, the traditional inter-reference gap or IRG is used, but for comparison purpose, we refer to IRG as *PIRD* (Page-level Inter-Reference Distance). For example, in this sequence, the PIRD of page 10 is 7, while the PIRD of page 13 is 0.

4.4. Locality Estimation and Adaptive Partitioning: A Distribution-Based Approach

Our BPAC scheme requires that the p-list and b-list share the cache space, and that the size-dependent region is dynamically adjusted in the b-list. Both requirements demand partitioning mechanisms, which we intend to make adaptive to various workloads. Considering that the dual-list differentiates pages of high and low temporal locality, and that the size-independent and size-dependent regions contain clusters of high and low spatial locality respectively, we may measure the two localities, and from the measurement results, we may have hints about the time/duration for a page to stay "hot" or for the cluster to be "growing larger".

4.4.1. Partitioning between P-List and B-List with PIRD Distribution. BPAC keeps the hot pages in the p-list, however, most of the hot pages are only temporarily hot, and for these kind of temporary hot-spots, there must be some method to determine their lifetime in the p-list. We use the *PIRD*, which represents the interval of re-accesses to the same page to quantify the temporal locality. Our speculation is that, if a page is repeatedly requested by the application, such as when there are updates to some key data, the access sequence may show certain regularity. For example, the intervals within the sequence will tend to be stable; thus, the future accesses should also follow the same interval. As a result, if a hot page has not been accessed for such an interval, it is highly possible that this page's temporary hotness is gone, and thus the page should be evicted from the p-list. However, due to the interference among the access streams of different applications, the variance within a single hot page's PIRD sequence is found to be high: to analyze the PIRD sequences of each hot page, Figure 4(a) plots the distributions of the standard deviations of all hot pages' PIRD sequences using the F1 trace, and as shown in the figure, a large portion of hot pages (the "peak" in the distribution) have deviations on the scale of 10^4 . This observation implies that it is infeasible to use the PIRD average of a single page to predict its future access interval.

To eliminate the impact of interference among access streams arising from the mixture of workloads of various applications, we turn into the "overall" PIRD distribution: by putting together the PIRD's of all hot pages' PIRD sequences, we have the distribution of the occurrence counts of all these PIRD's. Although due to the interference, the regularity of a single page's PIRD sequence is compromised, with the overall distribution, we found that a set of PIRD values that contribute the most in the distribution is within a short range starting at 0. In Figure 5(a), the overall PIRD distribution is obtained with the entire F1 trace; as shown in the figure, a threshold of about 200 is enough to cover most of the distribution. The cause of this phenomenon is that, as the main contributor of the overall distribution, the set of hottest pages tend to have low PIRD's: Figure 5(b) presents the correlation between the PIRD averages of the

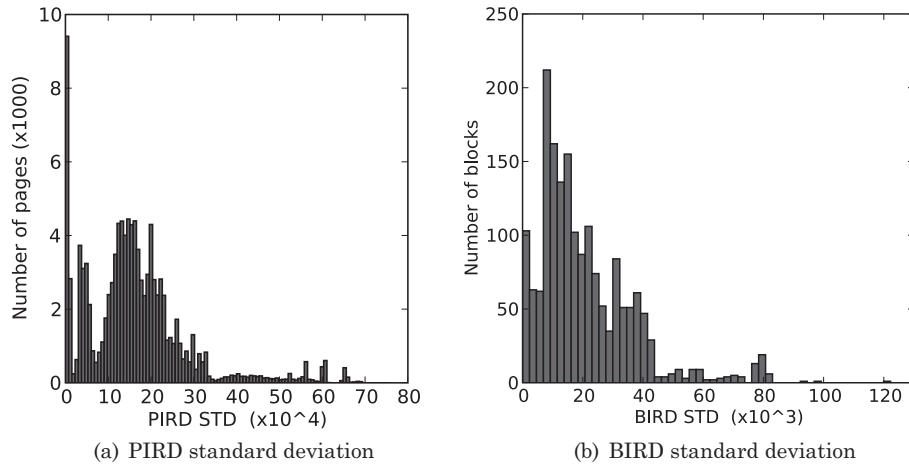


Fig. 4. Distributions of the standard deviation of PIRD/BIRD sequences, with F1 trace. Y axis represents the number of pages that have a corresponding PIRD or BIRD standard deviation value, which is marked by X axis.

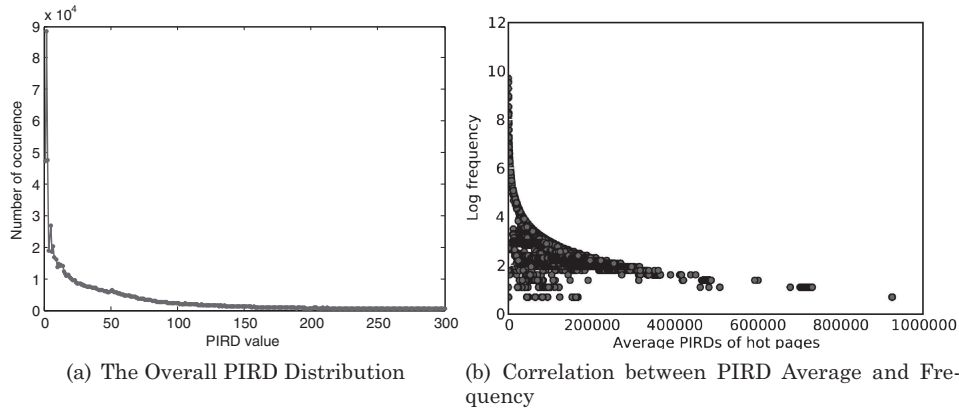


Fig. 5. Overall PIRD Distribution Analysis. (a) is the histogram of PIRD value occurrence. (b) plots the relationship between access frequency (Y axis) and average PIRD value (X axis); each dot represents a page.

hot pages and their frequencies (hotness) with the F1 trace; as the “hotness” increases along the y axis, the corresponding PIRD average decreases.

Based on the above observations, if a small threshold is used as an estimation of the upper bound of the access intervals of the hot pages, most of PIRD sequences are expected to be covered. We refer to it as “PIRD_{thd}”. If a hot page is not accessed for this upper bound interval, the chance is minimal that there will still be more accesses to this page in the near future. A series of overall distributions are sampled by collecting the PIRD’s for each consecutive period (e.g., every 10k virtual time) of the workload, and they are very similar to the overall distribution with the entire workload (Figure 5). Upon such a series of distributions, a series of PIRD_{thd}’s are located.

PIRD_{thd} is found stable within each workload of the traces we used, and the stability of PIRD_{thd} allows it to be used as the predicted lifetime of the hot pages: PIRD_{thd} is derived from the last period’s overall distribution since the next period which will have a similar PIRD_{thd}, and thus most of the hot pages can be protected by the lifetime

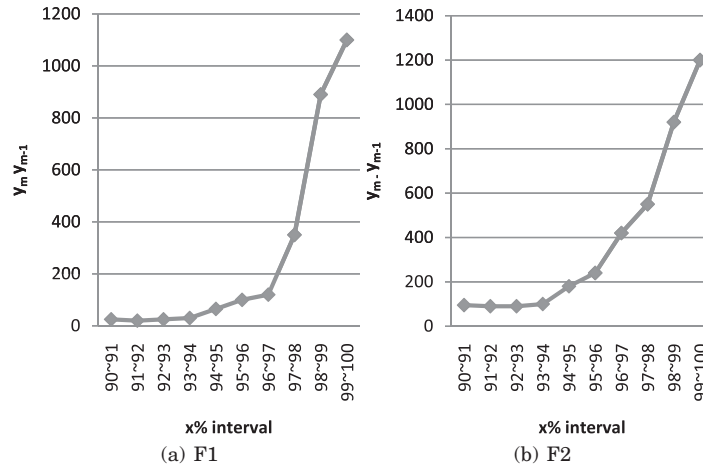


Fig. 6. BIRD Distribution Sparseness, with F1 and F2. The labels on X axis denote the interval of $x\% \sim (x - 1)\%$, while the Y axis values are the $(y_m - y_{m-1})$ series.

determined by the former threshold. However, in catching up with the variance while making predictions about the behavior of the workload, a trade-off has to be made on the sampling period. With the F1 and F2 traces (as well as the other two traces used in Section 5) containing write requests on the order of 1 million, we found that a sampling period of 10k virtual time, which results in about a few hundred of samples, are good for making such a trade-off.

4.4.2. Partitioning between Size-Dependent and Size-Independent Region with BIRD Distribution.

As proposed earlier in this section, we use BIRD sequence of a cluster as a metric of measuring the spatial locality (how fast a cluster grows). The deviations of single cluster's BIRD sequences are high, as shown in Figure 4(b). Taking PIRD_{thd} as an analogy, we use the "BIRD_{thd}" that covers the most of the overall BIRD distribution (consisting of BIRD's from all BIRD sequences) as the lifetime of the cluster staying in the size-independent region: if a cluster's lifetime expires, we state that the chance that there will be new pages to be added into this cluster (due to the spatial locality) any time soon is very low, and the largest cluster policy is applied to it while avoiding the early eviction problem. With the lifetime determined by BIRD_{thd}, the b-list is adaptively partitioned into size-independent region and size-dependent region, and thus BPAC achieves an automatic α instead of a manually-tuned one in CLC.

4.4.3. Adaptive PIRD_{thd} and BIRD_{thd} with the Inflection Point.

The proposed method for locating the PIRD_{thd} and BIRD_{thd} raises a question: What is the optimal "x%" for these two thresholds to cover? Is 90% better than 99% or not? Since we do not consider manually-tuned thresholds to be acceptable since this would require experimental, offline trials, we need an adaptive way of locating the optimal thresholds.

Upon close consideration of the overall distributions, we have decided to inspect them in another way: the *sparseness* of the occurrences of the PIRD's and BIRD's. First, we find the PIRD_{thd}/BIRD_{thd} series $y_1, y_2, y_3, y_4, \dots$, that cover the $x\%, (x + 1)\%, (x + 2)\%, (x + 3)\%, \dots$ of the distribution. Then we get the series $(y_2 - y_1), (y_3 - y_2), (y_4 - y_3), \dots$. With a larger $(y_m - y_{m-1})$, the probability that a PIRD/BIRD within this range occurs becomes smaller. We plot such data for the BIRD distribution in Figure 6 for the F1 and F2 traces. We can easily find the inflection points where the curves abruptly go vertical: in F1, the point is 97 ~ 98, and in F2 it is 94 ~ 95. This phenomenon can be explained by the random part of the distribution: beyond the

inflection point, the occurrences of PIRD's/BIRD's reflect the kind of PIRD's/BIRD's that barely repeat. Thus, we use the inflection points as the optimal thresholds, since in this way, only the non-random (popular) PIRD's/BIRD's are taken into consideration.

The workload monitor (Figure 1(b)) detects the incoming page and searches in its memory for the last reference time (upon page hit) of the page or last reference time of the corresponding cluster (upon page miss). The monitor then derives the PIRD/BIRD for this access, and then inserts this PIRD/BIRD into the PIRD/BIRD distribution. When a sampling period is over, the workload monitor will derive both `PIRD_thd` and `BIRD_thd` as the lifetimes for hot pages and hot clusters, respectively, for the next period; then it flushes the distributions and start sampling PIRD and BIRD again.

4.5. Replacement Policy: Differential Treatments

Due to different spatial locality features, various access patterns require different treatment:

Sequential Patterns. In most cache schemes, a sequential pattern stream's "marginal gain" is considered to be zero. However, for flash-aware schemes, in which the early evictions should be avoided, such stream's spatial locality must be fully exploited (the cluster contains an entire block, or for the case that the stream is not aligned to block size, a sequential cluster already contains the last page of the corresponding block; we refer to the first case as "full" and the second as "done"), before it can be evicted to the FTL. In most cases, the BIRD of sequential pattern clusters is small and stable, and thus assigning such clusters' lifetime with `BIRD_thd` is unnecessary. Since `BIRD_thd` is used to predict the lifetime of random and looping pattern clusters, of which the BIRD sequences are more variant, the BIRD sequences of sequential pattern clusters are not included in the BIRD distribution. If a sequential cluster is found "full" or "done", this cluster is considered to be ready to give up its cache space.

Looping Patterns. Within a short period (shorter than the looping period), a looping pattern can be considered to be a sequential pattern. For small cache, in which the "stack distance" of one looping cluster may be larger than the cache size, such cluster's repeated accesses cannot be captured by the cache. For large cache, a looping pattern stream appears as looping cluster(s) being sequentially scanned over and over again; the scanning/looping periods are found unstable. Looping patterns are detected in BPAC, and pages of looping clusters are kept in the b-list so as to not compromise their spatial locality. BPAC's workload monitor includes the BIRD sequences of looping pattern clusters in the overall BIRD distribution, and the their lifetime is determined by `BIRD_thd`.

Random Patterns. These clusters are neither sequential nor looping. Due to their low predictability of BIRD sequence, their lifetime is determined by `BIRD_thd`.

The victim selector selects the cluster to be evicted in the order of "full" and sequential cluster, "done" and sequential cluster, followed by the largest cluster in the size-dependent region (containing looping or random clusters that run out of lifetime).

A detailed algorithm of BPAC is described in Algorithm 1.

4.6. Overhead Analysis

4.6.1. Memory Overhead. BPAC has a few sources of memory overhead as discussed in the following.

- First, like most cache algorithms, BPAC has an index table for all the cached pages to determine the cache hit or miss.
- Second, each cache element, either a page or a block, has to keep its *recency* value. This is implemented as a timestamp denoting the virtual time at which this element is accessed most recently. This time-stamp merely requires a byte per element.

ALGORITHM 1: BPAC Algorithm

```

foreach each requested page,  $p$ , which belongs to cluster  $b$  do
  if cache hit then
    PIRD =  $current\_time - last\_reference\_time(p)$ 
    if hit in  $b$ -list then
      if  $b$  is not looping then
        update the recency of  $p$ 
        move  $p$  to  $p$ -list
      end
      update the recency of  $b$ 
    end
    else if hit in  $p$ -list then
      update the recency of  $p$ 
      insert PIRD in  $PIRD\_distribution$ 
    end
  end
  if cache miss then
    if cache full then
      call  $victim\_selector()$ 
    end
    BIRD =  $current\_time - last\_reference\_time(b)$ 
    insert  $p$  in  $b$  and update the recency of  $b$ 
    if  $b$  is not sequential then
      insert BIRD in  $BIRD\_distribution$ 
    end
  end
  if pages out of lifetime ( $recency < current\_time - PIRD.thd$ ) exist then
    move them to  $b$ -list
  end
  if sampling period is over then
     $PIRD.thd = get\_threshold(PIRD\_distribution)$ 
     $BIRD.thd = get\_threshold(BIRD\_distribution)$ 
    flush  $PIRD\_distribution$  and  $BIRD\_distribution$ 
  end
end

```

—Third, the PIRD and BIRD distributions are built with all the PIRD and BIRD values collected during one sampling period. The sampling period determines the amount of these values, which means at each virtual time, a PIRD or BIRD value is produced (according to Algorithm 1) and inserted into the distributions. Thus, PIRD and BIRD distributions maintain up to 10k values at the end of one sampling period. Therefore, assuming each value consumes one byte, BPAC ends up with extra memory usage of 10KB for those distributions, which is very small and acceptable.

4.6.2. *Computational Overhead.* The computational overhead is discussed for the following scenarios.

- For each new request, BPAC examines whether there are pages that are out of lifetime in p -list and moves them to b -list. This operation is implemented with minimal overhead by going through p -list from the LRU tail, examining whether the current page is out of lifetime, if so, re-linking the selected pages to b -list; otherwise, terminating the scanning process.
- Upon a cache hit, BPAC mainly involves the repositioning of the corresponding page or block (by updating the recency value) and the insertion of one PIRD value into the PIRD distribution.

Table IV. Statistics of Disk I/O Traces

	F1	F2	Cello99-Disk3	Cello99-Disk8
Total Requests(10^6)	1	0.65	0.72	1
Unique pages	113561	98239	267894	249387
Total pages	1930249	1029983	1516588	2315396

- Upon a cache miss, the new page is inserted in the b-list. If necessary, cache replacement is performed by calling *victim_selector()*, which searches the size-dependent region of b-list and selects the victim by the criteria described in Section 4.5. The potential victims, for example, sequential blocks, are kept track of by the victim selector to avoid scanning through the entire b-list to search for them.
- At the end of each sampling period, two critical parameters, *PIRD.thd* and *BIRD.thd*, are calculated with the corresponding distributions. This computation is simply to wipe out a portion of *PIRD/BIRD* values that barely repeat.

To conclude, the computational overhead of BPAC is kept minimal.

5. EVALUATION

5.1. Evaluation Methodology and Experiment Configuration

To verify the effectiveness of BPAC, we have conducted trace-driven simulations and compared BPAC with BPLRU and FAB, which are supported by a modified the sim-cache module of the *simplescalar* tool set [SimpleScalar LLC 2009].

Three well-known real-world disk I/O traces are used as summarized in Table IV. Where, Financial 1 and Financial 2 [Storage Performance Council 2010] are obtained from OLTP applications running at two large financial institutions; the Cello99 [Hewlett-Packard Laboratories] trace pool is collected from the “Cello” server that runs HP-UX 10.20. Because the entire Cello99 is huge, we randomly use one day traces (07/27/99) of two disks (Disk 3 and Disk 8).

Since a typical page size in most SSD’s is 4KB, we convert the LBA’s of the entries in the original traces to 4KB page LBA’s. In our experimental tests, each SSD block consists of 64 pages by default and the cache size ranges from 8M to 128M bytes.

5.2. Experimental Results and Analysis

In this section we first compare BPAC with BPLRU and FAB; since BPAC adopts the mechanism of *BIRD.thd* to adaptively tune its own α value while CLC uses a fixed α , we will then compare BPAC with CLC after the derivation of the optimal case CLC. Note that in our simulation tests, the sampling period of both *PIRD* and *BIRD* distributions is 10k virtual time.

5.2.1. Eviction Counts. Our first experiment is to measure the eviction count under different cache schemes. Here the eviction count is the number of clusters evicted/destaged to the FTL. The results are shown in Figure 7. Note that due to its relatively poor performance, FAB is excluded from Figure 7. The results clearly show that BPAC outperforms BPLRU constantly in reducing the eviction count by 34% on average. The maximum reduction of 79.6% is observed for F2 trace when the cache size is 128MB. We attribute the performance gain of BPAC to two aspects.

First, *the Effectiveness of the Largest Cluster(LC) Policy on the Size-Dependent Region.* As shown in Figure 1, we use a size-dependent region to hold the clusters of low spatial locality, where the largest cluster is the candidate to be evicted. We found this policy works better than BPLRU to make more room in the cache for future new pages and existing hot pages. BPAC adopts the same policy on sequential patterns (evict sequential clusters as soon as cache space is needed), hence applying the LC policy on the random/looping pattern clusters of the size-dependent region is the main difference

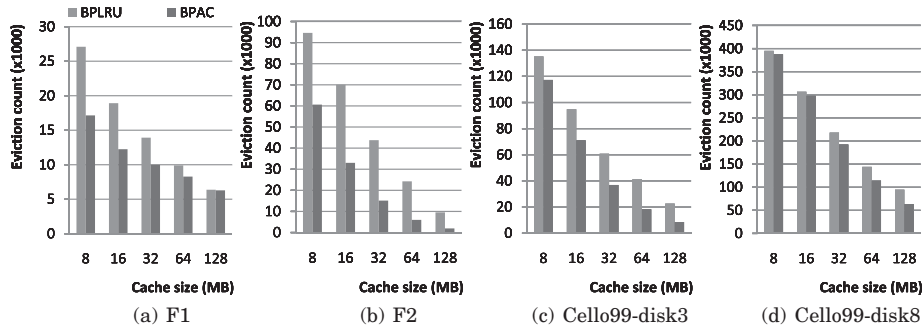


Fig. 7. Performance comparison: eviction counts.

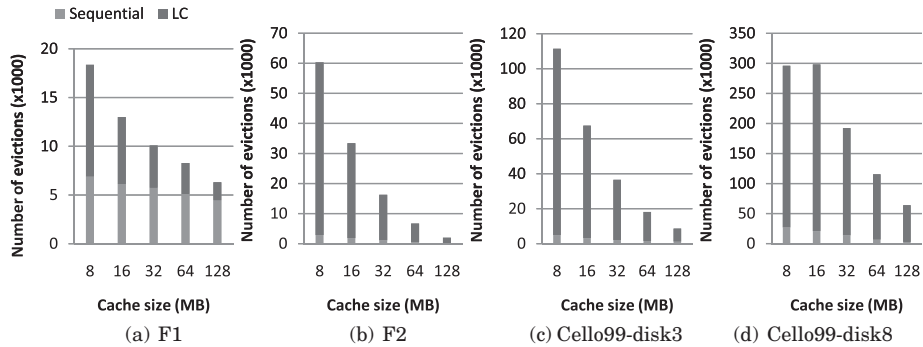


Fig. 8. Comparison between the numbers of evictions contributed by Largest Cluster (LC) policy and the “sequential” policy.

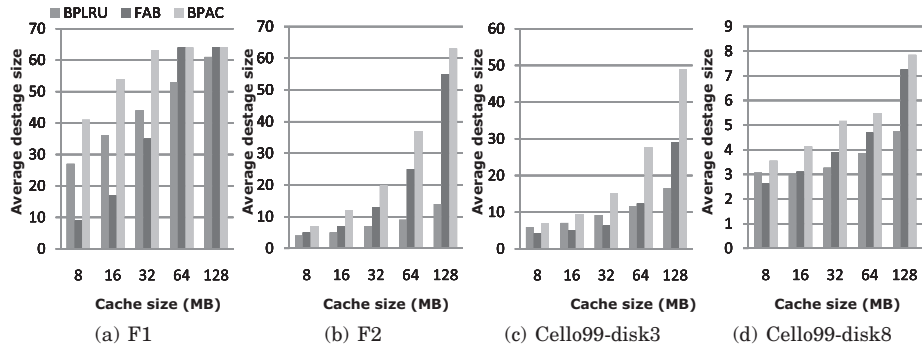


Fig. 9. Average destage size.

between the replacement policy of BPAC and BPLRU. In Figure 8, we break down the eviction counts of BPAC into two parts, one from the “LC” policy on random/looping patterns, and the other from the “sequential” policy on sequential patterns. As we can see, the main contributor for reducing the eviction count is “LC”, which is the main source of the performance gain BPAC has over BPLRU.

Additionally, we measure the average size of the destaged clusters in Figure 9. It’s obvious that BPAC increases the average destage size under different workloads. Specifically, compared to BPLRU, BPAC increases the destage size by 94.5% on average for all workloads and up to 350% for F2 trace when the cache size is 128MB. We notice

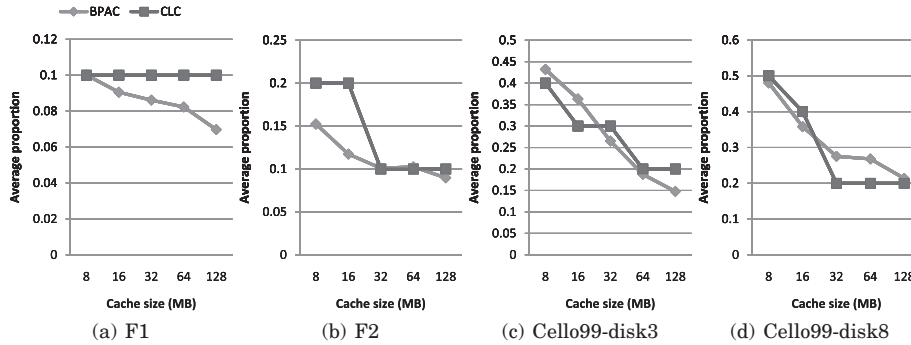


Fig. 10. The proportion of size-independent region: α .

that the average destage size is inversely proportional to the eviction count, which also shows that the idea of “LC” on the size-dependent region is effective. In contrast, FAB does not protect the hot clusters with a size-independent region and takes the entire list as a size-dependent region, and so FAB may produce many early evictions. As shown in Figure 9, FAB’s average destage size is smaller than others, particularly under small cache sizes.

As the cache size increases, the average destage size increases as well, since with larger cache space the spatial locality of the workload is better exploited. Considering BAST FTL, a larger destage size has another merit: the FTL programs pages of the destaged cluster on a log block, and when garbage collection process is executed on this log block (assuming there is no repeated writes on this log block, so a switch or partial merge can be formed), the more pages on the log block, the less clean pages will be on the dirty data block; so, the overhead of copying the clean pages from the data block to the log block is less. As discussed later in Section 5.2.2, our BPAC scheme, by reducing the number of cache destages and increasing the average destage size, results in a lighter overhead on the FTL.

Second, *the Effectiveness of BPAC’s Adaptive α* . The α value is the key factor that affects the eviction count, as we discussed about CLC in Section 3.1. To learn about the effectiveness of BPAC’s mechanism of adaptively tuning α , the `BIRD.thd`, we average the α value of each sampling period, and compare it with the manually obtained optimal α of the CLC scheme. The results are shown in Figure 10. Due to our adaptive tuning, BPAC approaches optimal α value automatically, in contrast to CLC, which can only achieve an optimal α manually. Figure 10 shows that the α values of BPAC and optimal CLC approximate each other. It is worth noting that, for certain workloads, as the cache size increases, the proportion of the size-independent region (in which the clusters are growing larger) is expected to decrease. In Figure 10, as the cache size increases, the decreasing trend of α in both CLC and BPAC is observed.

However, there exist differences, up to 0.1, between the α values of BPAC and optimal CLC. The reason is, BPAC updates its α according to `BIRD.thd` for every sampling period, so BPAC is more adaptive to the variances within the workload, while achieving a more accurate α . In contrast, CLC’s static α parameter is a prime hindrance in its design as manually tuning is not practical or even feasible in most cases under changing workloads. Furthermore, as shown in Figure 10, different α ’s should be applied to different workloads to achieve better performance. For example, the largest α ’s of F1, F2, Cello99-disk3, and Cello99-disk8, are 0.1, 0.2, 0.4 and 0.5, respectively. If the duration of spatial locality of clusters is longer in a particular workload, this means that more cache space of the size-independent region is needed to fully exploit the

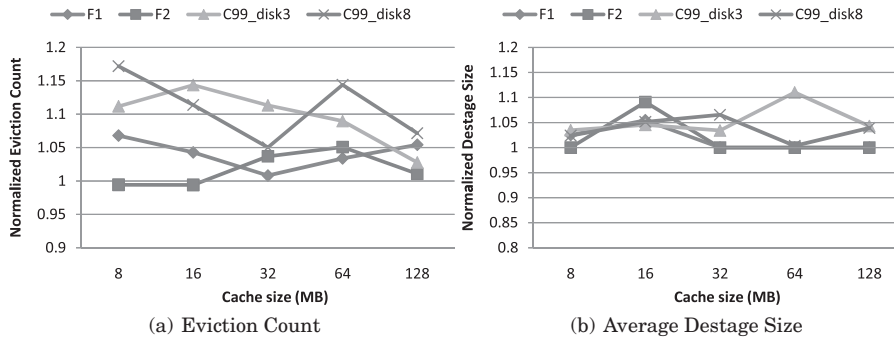


Fig. 11. Comparison between BPAC and CLC: eviction counts (CLC over BPAC) and average destage size (BPAC over CLC).

spatial locality, and thus the corresponding α should be larger. This observation is the main reason that the optimal α values vary among workloads. To conclude, BPAC's ability to adapt to different workloads and the variances inside a certain workload is the major improvement over CLC.

The α not only affects eviction counts, but also affects the destage size: if α is smaller than the optimal, early evictions occur, which split one cluster into multiple ones; if α is larger than optimal, the victims may not be the largest of all cold clusters. Thus, to support our claim of “BPAC achieves more accurate α ”, we compare the performance of BPAC with that of the optimal case CLC in Figure 11. For a clear view of the improvement BPAC has over CLC, the normalized eviction counts (normalized to BPAC) are given in Figure 11(a). BPAC has higher improvement over CLC under the cello traces than it does under F1 and F2 traces, with most cache sizes. For example, with cello99_disk3, BPAC is 10% better than CLC on average; while with F2, the improvement is between 0% to 5%. BPAC's improvement in the average destage size over CLC is given in Figure 11(b) (results are normalized to CLC).

5.2.2. Overhead on the FTL. Our next experiment is to evaluate how BPAC helps reduce the overhead on the FTL, particularly the overhead of the destaged clusters. In our experiments, we measure the overhead as the time the FTL spends on merge operations due to the shortage of log blocks. We focus on BAST FTL since FAST FTL shows similar trends. Typically a merge operation in BAST involves three steps: *read valid pages* (from the flash to page registers), *copy valid pages* (or program pages into the flash) and *erase dirty blocks*. Since the time to *read valid pages* is trivial compared to copy and erase, we do not report it in the figures. In the simulation, the number of log blocks is set to 50 and other key parameters are taken from Table II.

As we discussed previously, the average destage size is inversely proportional to the eviction count. Thus, the impact of the eviction count is clear: not only do smaller evictions and larger destages result in reduced numbers of merge operations, but each merge operation on average has lower overhead due to fewer clean page copy events. The results in Figure 12 show lower overhead in the FTL using BPAC (FAB is not shown in (b), (c) and (d) since its results are much larger than the others). For example, the overhead in the FTL using BPAC is 40% of the overhead using BPLRU with Cello99-disk3 trace and 64 MB cache.

5.2.3. Write Performance. To evaluate the impact of BPAC on the write performance, we integrated BPAC algorithm into *flashsim* [Gupta et al. 2009] simulator, which is configured using the flash parameters listed in Table II. The write performance in terms of average write latency under the four traces is illustrated in Figure 13. Compared

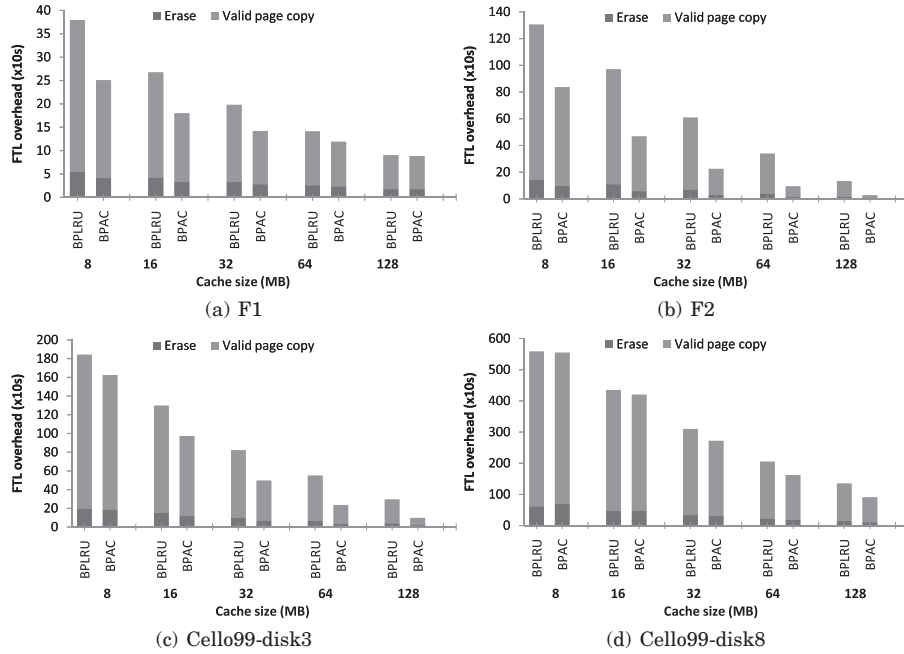


Fig. 12. Overhead on BAST FTL.

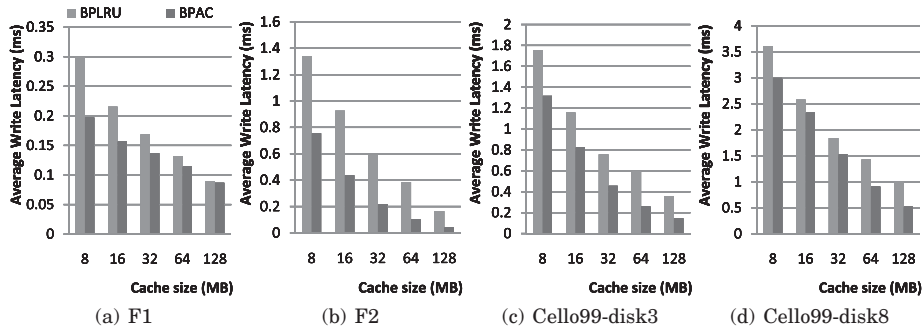


Fig. 13. Performance comparison: average write latency (in milliseconds).

to BPLRU, BPAC reduces the write latency by 37.1% on average and at most 75.1% (F2, 128MB). It is observed that the latency performance of BPAC, compared to that of BPLRU, substantially resembles the eviction counts (Figure 7).

5.2.4. Effectiveness of *PIRD_{thd}* for Partitioning P-List and B-List. Our last experiment is to evaluate the efficiency of BPAC's cache partitioning between its p-list and b-list. In our design, a threshold *PIRD_{thd}* is used to adaptively partition the cache space to hold pages with high temporal locality in the p-list and clusters of pages with low temporal locality in b-list. Figure 14 shows the breakdown of the page hits in the p-list and b-list. We can see from the figure that both the p-list and b-list contribute to cache hits. We also observe that the contribution of the p-list under Cello99 workload is smaller than that under F1 and F2 traces. The reason for this is the smaller number of hot pages present in the Cello99 traces, which can be seen in Table IV. By separating hot pages from the cold ones, we expect that the b-list absorbs the "first hit" of hot pages. Therefore, with

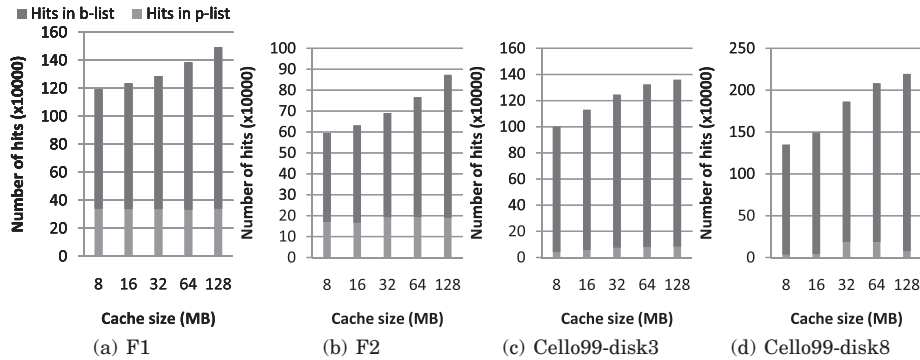


Fig. 14. Comparison between the numbers of hits in p-list and b-list of the BPAC scheme.

a larger cache space, the b-list will contribute to the increase of the total hits; while for the p-list, regardless of the cache space, it should absorb the hits of hot pages, and the hits in it should be constant. In Figures 14(a) and 14(b), we notice that as cache size increases from 8MB to 128MB, the number of hits in the p-list remains constant, which demonstrates that the dual-list data structure of BPAC effectively detaches hot pages from cold ones.

6. RELATED WORK

6.1. Traditional Cache Policies

There has been a wealth of cache research over many decades to take advantage of the locality of reference, resulting in a wide variety of techniques to achieve ever higher hit ratios [Karedla et al. 1994]. In the LRU and LFU families, there are two main directions that cache research tends towards. The first direction concerns efficiency, while the second concerns gathering more history in an effort to glean more dependable statistics about the likelihood of a page being accessed again. CLOCK/GCLOCK [Nicola et al. 1992] is a more efficient approximation of LRU, and LRU-k [O’Neil et al. 1993] looks at k accesses ago to get a mathematically tractable expectation of how hot a particular page is. 2Q [Johnson and Shasha 1994] is an approximation to LRU-2 with constant overhead. There is also LRFU [Lee et al. 2001], which seeks to generalize both the LRU and LFU families.

Recently, many policies emerged to utilize both frequency and recency (Inter-reference Gap or IRG) to determine the longevity of data in cache such as MQ [Zhou et al. 2001, 2004]. UBM [Kim et al. 2000] tries to identify sequential and looping references, and uses a marginal gain technique to evict data from cache. LIRS [Jiang and Zhang 2002] uses inter-reference recency to hold and discard data. ARC [Megiddo and Modha 2003, 2004] and CAR/CART [Bansal and Modha 2004] are scan-resistant and adaptive Clock variants. Motivated by the fact that no single caching policy could adapt to all workloads, SOPA [Wang et al. 2010] selects the optimal policy according to the on-line workload analysis.

Why not just apply one of these caching strategies to SSD’s? The erase operation is at block granularity and the write operation is much slower than read: thus, read and writes are not equal and no operations have seek time. Cache policies which take advantage of disk locality are not applicable (e.g., CSCAN looks at the disk head and arranges writes to move in one direction). SSD’s can not modify data in-place, thus, an SSD cache must minimize erase operations incurred by the updating of data. The erase and read-modify-write penalties can be ameliorated by evicting entire clusters from the cache. There exists a natural trade-off between a high hit ratio (exploiting temporal

locality) and low erase count through entire cluster evictions (exploiting spatial locality at block level). Dual-locality has been explored before, such as in DULO [Jiang et al. 2005] and WOW [Gill and Modha 2005], but not in the domain of SSD's.

6.2. Flash-Aware Cache Schemes

In addition to the schemes discussed earlier, there are a few more in the literature:

PUD-LRU [Hu et al. 2010] shares many common points with BPAC, for example, it also partitions a block based buffer list to a “hot” part and a “cold” part and applies a FAB-like replacement policy on the cold part; it also adopts a history-based method to evaluate the hotness of a certain block. Compared to BPAC, its major problem is that it uses an pre-defined threshold in partitioning the cache, which would not adapt to different workloads.

Griffin [Soundararajan et al. 2010]. Griffin is proposed to use a HDD as a write cache for SSD. By converting the update writes into a HDD-based log, which is eventually merged with the data on the SSD, Griffin takes advantage of HDD's high sequential write speed and large capacity to reduce the amount of writes directly serviced by the SSD while improve the sequentiality of the workload. Similarly, *I-CASH* [Ren and Yang 2011] uses SSD to store seldom-changed and mostly read data blocks and uses HDD to store the logs of changes made to the cold data on the SSD.

CFLRU [Park et al. 2006]. Clean-first LRU tries to leverage the read/write asymmetry by picking out pages to evict which are not dirty. Thus, the eviction will not lead to any actual write to the drive. Although it is not applicable for on-disk write cache, this technique gives is an interesting attempt at an OS-level flash-aware cache scheme.

Shim et al. [2010] proposed an adaptive method for partitioning the on-disk cache to the data buffering and the mapping table caching. The cost and benefit of assigning cache space to both parts are evaluated and the partition is adjusted on-line to achieve the optimal overall performance, which is evaluated by both read/write performance and mapping translation overhead.

NOR flash's support for byte-access makes it widely used for program execution on embedded system; however, NOR flash's storage capacity per chip is normally smaller than that of NAND flash. Motivated by the market demand for replacing NOR flash with NAND flash, Chang et al. [2010] provided a prediction-based prefetching strategy in the SRAM, for the sake of improving NAND flash's performance on the program execution workloads.

6.3. Other SSD-Related Techniques

File Systems. Early flash file systems such as YAFFS [Manning 2010] and JFFS2 [Hat 2010] are designed for embedded systems and work on the raw flash. On the contrary, DFS [Josephson et al. 2010] is implemented over the virtualized flash interface offered by Fusion-IO driver. By leveraging this interface, it avoids the complexity of physical block management of traditional file systems.

FTLs. For block level mapping, many FTL schemes have been proposed to use a number of physical blocks to log the updates. Examples include the former-mentioned FTLs (FAST [Lee et al. 2005], BAST [Kim et al. 2002], SAST [Kang et al. 2006]), and LAST [Lee, S. et al. 2008]. For page level mapping, DFTL [Gupta et al. 2009] is proposed to cache the frequently used mapping table in the on-disk SRAM so as to improve the address translation performance; μ -FTL [Lee, Y.-G. et al. 2008] adopts the μ -tree on the mapping table to reduce the memory footprint. Two-level FTL [Wu and Kuo 2006] is proposed to dynamically switch between page and block mapping. Recently, data deduplication is included in the FTL to boost up write performance and to reduce the actual write commitment on the flash [Chen et al. 2011; Gupta et al. 2011].

Heterogeneous Material. Utilizing advantages of PCRAM, such as the in-place update ability and faster access, Sun et al. [2010] describe a hybrid architecture to log the updates on PCRAM for flash. Frash [Jung et al. 2010] harbors the in-memory data and the on-disk structures of the file system on a number of byte-addressable NVRAMs. FlexFS [Lee et al. 2009], on the other hand, combines MLC and SLC as trading off the capacity and erase cycle.

Reliability-Enhancing Techniques. To alleviate the reliability problem caused by the limited life cycle of flash memory, a few techniques have been applied on various levels of the storage system: storage redundancy can be achieved by using ECC at the page level [Bez et al. 2003] and by using a RAID organization at the chip level [Agrawal et al. 2008; Hutsell et al. 2008] or at the device level [Balakrishnan et al. 2010], while the wear-leveling techniques [SiliconSystems 2005; Chang et al. 2007] try to evenly distribute the amount of wear on individual blocks.

7. CONCLUSION

In this article, we present BPAC, an adaptive flash-aware write cache that minimizes evictions by exploiting both spatial and temporal locality. According to temporal locality, hot pages are absorbed in the p-list, and blocks (clusters of pages) are cached in size-independent and size-dependent regions in the b-list according to their spatial locality. Simulation results show that compared to existing popular flash-aware schemes, BPAC reduces the number of evictions and increases the size of destages which, in turn, reduces the overhead on the FTL, and thus improves the overall performance.

REFERENCES

- AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX 2008 Annual Technical Conference on Annual Technical Conference*.
- BALAKRISHNAN, M., KADAV, A., PRABHAKARAN, V., AND MALKHI, D. 2010. Differential RAID: Rethinking RAID for SSD reliability. *ACM Trans. Storage* 6, 2, 1–22.
- BANSAL, S. AND MODHA, D. S. 2004. CAR: Clock with Adaptive Replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*. 187–200.
- BEZ, R., CAMERLENGHI, E., MODELLI, A., AND VISCONTI, A. 2003. Introduction to flash memory. *Proc. IEEE* 91, 489–502.
- CHANG, Y., LIN, J., HSIEH, J., AND KUO, T. 2010. A strategy to emulate NOR flash with NAND flash. *ACM Trans. Storage* 6, 2, 1–23.
- CHANG, Y.-H., HSIEH, J.-W., AND KUO, T.-W. 2007. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. In *Proceedings of the IEEE / ACM Design Automation Conference (DAC)*.
- CHEN, F., LEE, R., AND ZHANG, X. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE.
- EETIMES. 2010. http://www.eetimes.com/electronics-news/4207194/Toshiba_rolls_24_nm_NAND_flash.
- ENGADGET. <http://www.engadget.com/2010/04/19/>.
- GILL, B. S. AND MODHA, D. S. 2005. WOW: Wise Ordering For Writes—Combining spatial and temporal locality in non-volatile caches. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST'05)*. USENIX Association, Berkeley, CA.
- GUPTA, A., KIM, Y., AND URGAONKAR, B. 2009. 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 Programming Languages and Operating Systems (ASPLOS'09)*. ACM, New York, 229–240.
- GUPTA, A., PISOLKAR, R., URGAONKAR, B., AND SIVASUBRAMANIAM, A. 2011. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*. USENIX Association.
- HAT, R. 2010. The journalling flash file system, version 2. <http://sourceware.org/jffs2/>.

- HEWLETT-PACKARD LABORATORIES. cello99 traces. <http://tesla.hpl.hp.com/opensource/>.
- HU, J., JIANG, H., TIAN, L., AND XU, L. 2010. PUD-LRU: An erase-efficient write buffer management algorithm for flash memory SSD. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer Systems*. 69–78.
- HUTSELL, W., BOWEN, J., AND EKKER, N. 2008. Flash solid-state disk reliability. Tech. rep.
- INTEL. 2009. Intel X25-M SATA Solid State Drive. <http://download.intel.com/design/flash/nand/mainstream/322296.pdf>.
- INTEL. 2010. <http://www.intel.com/pressroom/archive/releases/20100201comp.htm>.
- JIANG, S., DING, X., CHEN, F., TAN, E., AND ZHANG, X. 2005. DULO: An effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST'05)*. USENIX Association, Berkeley, CA.
- JIANG, S. AND ZHANG, X. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM SIGMATICRIS International Conference Measurement and Modeling of Computer Systems*, 31–42.
- JO, H., KANG, J.-U., PARK, S.-Y., KIM, J.-S., AND LEE, J. 2006. FAB: Flash-Aware Buffer management policy for portable media players. *IEEE Trans. Consum. Elect.* 52, 2, 485–493.
- JOHNSON, T. AND SHASHA, D. 1994. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 439–450.
- JOSEPHSON, W., BONGO, L., LI, K., AND FLYNN, D. 2010. DFS: A file system for virtualized flash storage. *ACM Trans. Storage* 6, 3, 1–25.
- JUNG, J., WON, Y., KIM, E., SHIN, H., AND JEON, B. 2010. FRASH: Exploiting storage class memory in hybrid file system for hierarchical storage. *ACM Trans. Storage* 6, 1, 1–25.
- KANG, J. U., JO, H., KIM, J. S., AND LEE, J. 2006. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the International Conference on Embedded Software*.
- KANG, S., PARK, S., JUNG, H., SHIM, H., AND CHA, J. 2009. Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices. *IEEE Trans. Comput.* 58, 6, 744–758.
- KAREDLA, R., LOVE, J. S., AND WHERRY, B. G. 1994. Caching strategies to improve disk system performance. *IEEE Comput.* 27, 3, 38–46.
- KIM, H. AND AHN, S. 2008. BPLRU: A buffer management scheme for improving random writes in flash storage abstract. In *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST'08)*.
- KIM, J. M., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. 2000. A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI'00)*. 119–134.
- KIM, J., KIM, J. M., NOH, S., MIN, S. L., AND CHO, Y. 2002. A space-efficient flash translation layer for compact-flash systems. *IEEE Trans. Consum. Electron.* 48, 2, 366–375.
- LEE, D., CHOI, J., KIM, J., NOH, S., MIN, S., CHO, Y., AND KIM, C. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.* 50, 12, 1352–1361.
- LEE, S., HA, K., ZHANG, K., KIM, J., AND KIM, J. 2009. FlexFS: A flexible flash file system for MLC NAND flash memory. In *Proceedings of the USENIX Annual Technical Conference*. USENIX.
- LEE, S., SHIN, D., KIM, Y.-J., AND KIM, J. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.* 42, 6, 36–42.
- LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S.-W., AND SONG, H.-J. 2005. FAST: An FTL scheme with fully associative sector translations. In *Proceedings of the UKC Conference*. UKC.
- LEE, Y.-G., JUNG, D., KANG, D., AND KIM, J.-S. 2008. uFTL: a memory-efficient flash translation layer supporting multiple mapping granularities. In *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT'08)*. ACM, New York, 21–30.
- MANNING, C. 2010. Yet another flash file system. <http://www.yaffs.net/>.
- MASON, L. 2009. Rethinking SSDs. <http://www.denali.com/wordpress/index.php/dmr/2009/07/23/rethinking-ssds>.
- MEGIDDO, N. AND MODHA, D. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. 115–130.
- MEGIDDO, N. AND MODHA, D. S. 2004. Outperforming LRU with an adaptive replacement cache algorithm. *Computer* 37, 4, 58–65.

- NEWEGG. 2009a. Intel X25-M Mainstream SSDSA2MH160G2C1 2.5-inch 160GB SATA II MLC Internal Solid state disk (SSD). <http://www.newegg.com/Product/Product.aspx?Item=N82E16820167017>.
- NEWEGG. 2009b. Western Digital VelociRaptor WD3000HLFS 300GB 10000 RPM 16MB cache SATA 3.0Gb/s 3.5-inch internal hard drive—OEM. <http://www.newegg.com/Product/Product.aspx?Item=N82E16822136322>.
- NICOLA, V., DAN, A., AND DIAS, D. 1992. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, 35–46.
- O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*. ACM, New York, 297–306.
- ONFI. 2010. <http://onfi.org/>.
- PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S., AND LEE, J. 2006. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. ACM, New York, 234–241.
- REN, J. AND YANG, Q. 2011. I-CASH: Intelligently Coupled Array of SSD and HDD. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
- ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1, 26–52.
- SAMSUNG. 2010. <http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products-OneNAND.html>.
- SHIM, H., SEO, B.-K., KIM, J.-S., AND MAENG, S. 2010. An adaptive partitioning scheme for DRAM-based cache in solid state drives. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*. 1–12.
- SHIMPI, A. L. 2009. Intel x25-m g2: Dissected and performance preview. <http://www.anandtech.com/storage/showdoc.aspx?i=3607>.
- SILICONSYSTEMS. 2005. Increasing flash solid state disk reliability. Tech. rep.
- SIMPLESCALAR LLC. 2009. The simplescalar tool set. <http://www.simplescalar.com/>.
- SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. 2010. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. USENIX.
- STORAGE PERFORMANCE COUNCIL. 2010. SPC trace file format specification. <http://traces.cs.umass.edu/index.php/Storage/Storage>.
- SUN, G., JOO, Y., CHEN, Y., NIU, D., XIE, Y., CHEN, Y., AND LI, H. 2010. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA-16)*. IEEE, 141–153.
- TOSHIBA. 2010. <http://www.toshiba.com/taec/news/press-releases/2006/memy-06-337.jsp>.
- UMASS. 2007. Umass trace repository. <http://traces.cs.umass.edu/index.php/Storage/Storage>.
- WANG, Y., SHU, J., ZHANG, G., XUE, W., AND ZHENG, W. 2010. SOPA: Selecting the optimal caching policy adaptively. *ACM Trans. Storage* 6, 2, 1–18.
- WESTERN DIGITAL. December 2008. WD VelociRaptor SATA hard drives. <http://www.wdc.com/en/library/sata/2879-701282.pdf>.
- WU, C.-H. AND KUO, T.-W. 2006. An adaptive two-level management for the flash translation layer in embedded systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'06)*. ACM, New York, 601–606.
- ZHOU, Y., CHEN, Z., AND LI, K. 2004. Second-level buffer cache management. *IEEE Trans. Parallel Distrib. Syst.* 15, 6, 505–519.
- ZHOU, Y., PHILBIN, J., AND LI, K. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX Annual Technical Conference (General Track)*. USENIX Association, Berkeley, CA, 91–104.

Received November 2010; revised May 2011; accepted June 2011