Azor: Using Two-level Block Selection to Improve SSD-based I/O caches

Yannis Klonatos*†, Thanos Makatos*, Manolis Marazakis*, Michail D. Flouris*, and Angelos Bilas*†

*Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS)
100 N. Plastira Ave., Vassilika Vouton, Heraklion, GR-70013, Greece
†Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.
{klonatos, mcatos, maraz, flouris, bilas}@ics.forth.gr

***Abstract*—Flash-based solid state drives (SSDs) exhibit potential for solving I/O bottlenecks by offering superior performance over hard disks for several workloads. In this work we design *Azor*, an SSD-based I/O cache that operates at the block-level and is *transparent* to existing applications, such as databases. Our design provides various choices for associativity, write policies and cache line size, while maintaining a high degree of I/O concurrency. Our main contribution is that we explore differentiation of HDD blocks according to their expected importance on system performance. We design and analyze a two-level block selection scheme that dynamically differentiates HDD blocks, and selectively places them in the limited space of the SSD cache.**

We implement *Azor* in the Linux kernel and evaluate its effectiveness experimentally using a server-type platform and large problem sizes with three I/O intensive workloads: TPC-H, SPECsfs2008, and Hammerora. Our results show that as the cache size increases, *Azor* enhances I/O performance by up to 14.02×, 1.63×, and 1.55× for each workload respectively. Additionally, our two-level block selection scheme further enhances I/O performance compared to a typical SSD cache by up to 95%, 16%, and 34% for each workload, respectively.

I. Introduction

The cost and performance characteristics of current generation NAND-Flash solid-state drives (SSDs), shown in Table I, make them attractive for accelerating demanding server workloads, such as file and mail servers, business and scientific data analysis, as well as OLTP databases. SSDs have potential to mitigate I/O penalties, by offering superior performance to common hard-disk devices (HDDs), albeit at a higher cost per GB [1]. In addition, SSDs bear complexity caveats, related to their internal organization and operational properties. A promising mixed-device system architecture is to deploy SSDs as a caching layer on top of HDDs, where the cost of the SSDs is expected to be amortized over increased I/O performance, both in terms of throughput (MB/sec) and access rate (IOPS).

Recently, there has been work on how to improve I/O performance using SSD caches. FlashCache [2] uses flash memory as a secondary file cache for web servers. Lee et al. [3] analyze the impact of using SSDs in transaction processing, while [4] examines how SSDs can improve checkpointing performance. Finally, [5] examines how SSDs can be used as a large cache on top of RAID to conserve energy. In all cases SSDs demonstrate potential for improved performance.

In this work we design *Azor*, a system that uses SSDs as caches in the I/O path. In contrast to all aforementioned approaches that are application-specific and require application knowledge, intervention, and tuning, *Azor transparently* and *dynamically* places data blocks in the SSD cache as they flow in the I/O path between main memory and HDDs. In this work, we investigate the following problems:

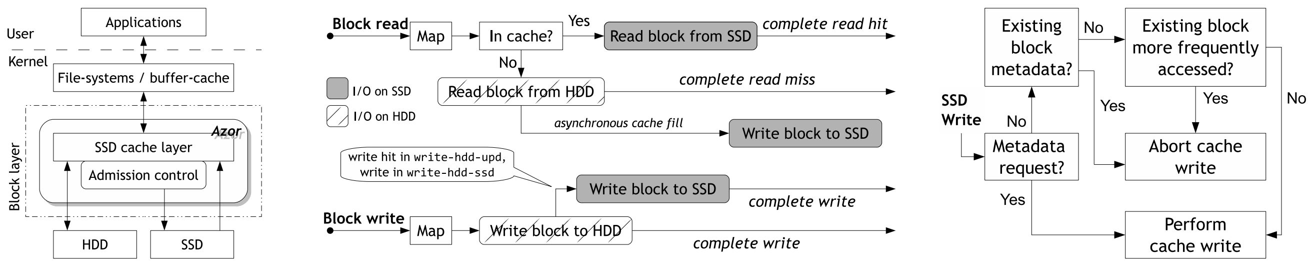
TABLE I
HDD and SSD performance metrics.

	SSD	HDD
Price/capacity ($/GB)	$3	$0.3
Response time (ms)	0.17	12.6
Throughput (R/W) (MB/s)	277/202	100/90
IOPS (R/W)	30,000/3,500	150/150

1) Differentiation of blocks: based on their expected importance to system performance. *Azor* uses a two-level block selection scheme and dynamically differentiates HDD blocks before admitting them in the SSD cache. First, we distinguish blocks that contain filesystem metadata from blocks that merely contain application data. This is important, since filesystem studies have shown that metadata handling is critical to application performance. In addition, recent trends show that the impact of filesystem metadata accesses is expected to become even more pronounced [6]. Second, for all HDD blocks we maintain a running estimate of the number of accesses over a sliding time window. We then use this information to prevent infrequently accessed blocks from evicting more frequently accessed ones. Our scheme does not require application instrumentation or static a-priori workload analysis and adds negligible CPU and I/O overhead.

2) Metadata footprint: for representing the state of the SSD cache blocks in DRAM. In our approach, metadata size grows proportionally to the SSD capacity available, rather than the much larger HDD space; still, compacting metadata to fit in DRAM is an important concern. We consider this experimental study important, since metadata footprint in DRAM will keep increasing with SSD device capacity, thereby making high-associativity organizations less cost-effective. There are two aspects of the cache design that determine the DRAM required for metadata: *cache-associativity* and *cache line size*.

First, we explore two alternatives for cache associativity: *a*) a direct-mapped organization, which minimizes the required amount of DRAM for metadata, and *b*) a fully-set-associative



(a) System architecture (b) Read and write paths (c) Admission control path

Fig. 1. *Azor* system architecture, I/O, and admission control paths.

organization that allows more informed block replacement decisions at the cost of consuming more DRAM space for its metadata. We quantify the performance benefits and potential caveats from employing the simpler-to-implement and more space-efficient cache design, under I/O-intensive workloads.

Then, we examine the performance impact from increasing the cache line size; although larger cache lines decrease the required metadata space in DRAM, doing so causes performance degradation in most cases.

3) Write-handling policies: for which we explore the following dimensions: *i*) write-through vs. write-back; this dimension affects not only performance, but also system reliability, *ii*) write-invalidation vs. write-update in case of cache write hits, and, finally, *iii*) write-allocation, for cache write misses. We experimentally find that the choice of the write policy can make up to a 50% difference in performance. Our results show that the best policy is as follows: write-through, write-update on write hits, and write-no-allocate on write misses.

4) Maintaining a high degree of concurrent I/O accesses: Any cache design needs to allow multiple pending I/O requests to be in progress at any time. *Azor* properly handles *hit-under-miss* and *out-of-order completions*, by tracking the dependencies between in-flight I/O requests. Our design minimizes the overhead of accessing the additional state required for this purpose, as this is required for all I/O operations that pass through the cache. We also compact this data structure enough so that it fits the limited DRAM space.

This is the first work that thoroughly and systematically examines the design space of SSD-based I/O caches. We implement *Azor* as a virtual block device in the Linux kernel and evaluate our design with three realistic and long-running workloads: TPC-H, SPECsfs2008, and Hammerora. We focus our evaluation on I/O-intensive operating conditions were the I/O system has to sustain high request rates. Our results show that *Azor*'s cache design leads to significant performance improvements. More specifically, as the available cache size increases, SSD-caching can enhance I/O performance from 2.91$\times$ to 14.02$\times$ and from 1.11$\times$ to 1.63$\times$ for TPC-H and SPECsfs2008, respectively. Furthermore, we show that when there is a significant number of conflict misses, our two-level scheme is able to enhance performance by up to 95% and 16% for these two workloads, respectively. We conclude our evaluation by examining the effectiveness of our design on Hammerora, a TPC-C type workload, and treating the application as a black box. For this workload, the base design of *Azor* improves performance up to 55%, compared to the HDD-only configuration, while with the use of our block selection scheme, *Azor* improves performance up to 89%.

The rest of this paper is organized as follows. Section II presents our design for resolving the aforementioned challenges without affecting access latency and I/O concurrency. Section III presents our experimental platform, representative of a current generation server for I/O intensive workloads. Section IV presents a detailed evaluation of our approach. Section V discusses some further considerations for *Azor* while Section VI provides a comparison with related work. Finally, Section VII summarizes our main findings.

II. System Design

Although SSD caches bear similarities to traditional DRAM-based caches, there are significant differences as well. First, the impact of the block mapping policy, e.g. direct-mapped vs. fully-set-associative, is not as clear as in DRAM caches. In addition, SSD caches are significantly larger, resulting in a considerably larger metadata footprint. This fact must be taken into account, considering the increasing size of SSDs. Finally, unlike DRAM caches, SSD caches can be made persistent, thus avoiding warm-up overheads.

The use of SSDs as I/O caches in our architecture is shown in Figure 1(a). *Azor* provides a virtual block device abstraction, by intercepting requests in the I/O path and transparently caching HDD blocks to dedicated SSD partitions. The address space of SSDs is not visible to higher system layers, such as filesystems and databases. *Azor* is placed in the I/O path below the system buffer cache. Although our approach can be extended to use the capacity of SSDs as storage rather than cache, more in the spirit of tiering, we do not explore this direction further in this work.

Figure 1(b) shows how we handle I/O requests. Each HDD block is first mapped to an SSD cache block, according to cache associativity. For reads, *Azor* checks if the cached block is valid and if so, it forwards the request to the SSD (*read hit*). Otherwise, data are fetched from the HDD (*read miss*) and an asynchronous SSD write I/O (*cache fill*) is scheduled.

For writes (hits or misses), *Azor* implements a *write-through* mechanism. We opt against using a *write-back* cache; such a cache design would result in the HDD not always having the most up-to-date blocks, therefore requiring synchronous metadata updates with significant implications for latency-sensitive workloads. Furthermore, a write-back cache reduces system resilience to failures, because a failing SSD drive could result in data loss. Our *write-through* design avoids these issues. *Azor* provides write policies for forwarding the write request either to both the HDD and the SSD (*write-hdd-ssd*), or only to the HDD. In the second policy, during a write hit, our system can either update (*write-hdd-upd*) or invalidate (*write-hdd-inv*) the corresponding cache block. The choice of write policy has significant implications for write-intensive workloads, as we show in our evaluation.

A. Admission Control Mechanism

Azor differentiates HDD blocks based on their expected importance to system performance. For this purpose, *Azor* uses a two-level block selection scheme that controls whether or not a specific cache block should be admitted to the SSD cache, according to its importance. Our design distinguishes two classes of HDD blocks: *filesystem metadata* and *filesystem data* blocks. However, we believe that an arbitrary number of other classes can be supported, if needed. The priorities between the two classes are explained in detail below.

To begin with, filesystem metadata I/Os should be given priority over plain data I/Os for two reasons. First, metadata operations represent between 50% and 80% of all the requests in a typical system [7]. Hence, their impact on performance is substantial. Second, there has been a marked increase in filesystem capacities in recent years, with the average file size remaining small [6]. This means more files and, thus, more metadata. The filesystem metadata footprint is further increased by the need for checksums at the filesystem level to achieve higher data protection [8], an approach already adopted by state-of-the-art filesystems, such as ZFS and BTRFS. Therefore, it is increasingly difficult to rely solely on DRAM for metadata caching and it makes sence to dedicate faster devices for storing filesystem metadata ([9], [10]).

In our design, differentiation between filesystem metadata and filesystem data is a straight-forward task. We modify the XFS filesystem to tag metadata requests by setting a dedicated bit in the I/O request descriptor. Then, *Azor* uses this information at the block level to categorize each HDD block. Our modification does not affect filesystem performance, can easily be implemented in other filesystems as well and only requires an additional *class* bit per SSD cache block.

Next, for the second level of our selection scheme, not all data blocks are treated as equal. For instance, in database environments indices improve query performance, by allowing fast access to specific records according to search criteria. Index requests produce frequent, small-size, and random HDD accesses, a pattern that stresses HDD performance. Moreover, given a set of queries to a database, the data tables are not usually accessed with the same intensity. In web-server environments, web pages usually exhibit temporal locality. Thus, we expect less benefit from caching web pages that have recently been accessed only sporadically. Finally, the same principle applies to mail-servers: more recent emails are more likely to be accessed again soon than older ones. Based on these observations, we cache data blocks on SSDs by differentiating them according to their access frequency.

At our second level of selection, we keep in-memory a running estimate of the accesses to each HDD block that is referenced at least once. Between any two HDD blocks, the one with the higher access count is more likely to remain in the SSD cache. This differentiation of HDD blocks overrides the selection of the "victim block" for eviction as determined by the LRU replacement policy in the fully-set-associative cache. Although workloads like TPC-C tend to have repetitive references, a good match for LRU, other workloads, such as TPC-H, rely on extensive one-time sequential scans which fill-up the cache with blocks that are not expected to be re-used any time soon. Such blocks evict others that may be accessed again soon. If we allow LRU replacement to evict blocks indiscriminately, the cache will not be effective until it is re-populated with the more commonly used blocks. This insight is also the motivation behind the ARC replacement policy [11], which keeps track of both frequently used and recently used pages and continuously adapts to the prevailing pattern in the reference stream. In our design, these per-block reference counters form an array in DRAM indexed by the HDD block number. The DRAM required for these counters increases along with the file-set size, not with the underlying HDD space. Our evaluation shows that this memory space is worth it, since differentiation improves performance overall.

Figure 1(c) shows our scheme. The control path of read hits and writes to HDD remains unaffected. On the other hand, cache fills and write hits to the SSD cache now pass through the scheme, which decides whether the write operation should actually be performed or not. If an incoming request is a metadata request, it is immediately written to the cache, since we prioritize filesystem metadata I/Os over plain data I/Os. Otherwise, the incoming request contains filesystem data and *Azor* checks whether the corresponding cache block contains filesystem metadata. If so, the cache fill is aborted, else both the incoming and the existing cache block contains data. In this case *Azor* checks which block is accessed more times, and the cache fill is performed (or aborted) accordingly.

B. Cache Associativity

The choice of associativity is mainly a tradeoff between performance and metadata footprint. Traditionally, DRAM caches use a fully-set-associative policy since their small size requires reducing capacity conflicts. SSD caches, however, are significantly larger and, thus, they may use a simpler mapping policy, without significantly increasing capacity conflicts. In this work we consider two alternatives for cache associativity: a *direct-mapped* and a *fully-set-associative* cache design.

On the one hand, a direct-mapped cache requires less metadata, hence a lower memory footprint, compared to a fully-set-associative cache. This is very important, since metadata are required for representing the state of the SSD blocks in DRAM, and DRAM space is limited. Specifically, our direct-mapped cache requires 1.28 MB of metadata per GB of SSD, needed for the address tag along with the valid and dirty bits, for each cache block. Furthermore, this cache design does not impose significant mapping overheads on the critical path and is fairly simple to implement. All these advantages are particularly important when considering offloading caching to storage controllers. However, modern filesystems employ elaborate space allocation techniques for various purposes. For instance, XFS tends to spread out space allocation over the entire free space in order to “enable utilization of all the disks backing the filesystem” [12]. Such techniques result in unnecessary conflict misses due to data placement.

On the other hand, a fully-set-associative cache requires a significantly larger metadata footprint to allow a more elaborate block replacement decision through the LRU replacement policy. However, such a design fully resolves the data placement issue, thus reducing conflict misses. Our fully-set-associative cache requires 6.04 MB of metadata per GB of SSD, 4.7$\times$ more than the direct-mapped counterpart. Metadata requirements for this design include, apart from the tag and the valid/dirty bits, pointers to the next and previous elements of the LRU list, as well as additional pointers for another data structure, explained shortly. Designing a fully-set-associative cache appears to be deceptively simple. However, our experience shows that implementing such a cache is far from trivial and it requires dealing with the following two challenges.

First, it requires an efficient mechanism that quickly determines the state of a cache block, without increasing latency in the I/O path. This is necessary since it is impossible to check all cache blocks in parallel for a specific tag, as a hardware implementation would do. *Azor* arranges cache blocks into a hash table-like data structure. For each HDD block processed, a bucket is selected by hashing the HDD block number using Robert Jenkins’ 32-bit integer hash function. The list of cache blocks is then traversed, looking for a match. This arrangement minimizes the number of cache blocks that must be examined for each incoming I/O request.

Second, there is a large variety of replacement algorithms typically used in CPU and DRAM caches, as well as in some SSD buffer management schemes [14], all of them prohibitively expensive for SSD caches in terms of metadata size. Moreover, some of these algorithms assume knowledge of the I/O patterns the workload exhibits, whereas *Azor* aims to be transparent. We have experimentally found that simpler replacement algorithms, such as random replacement, result in unpredictable performance. We opt for the LRU policy, since it provides a reasonable reference point for other more sophisticated policies, and we design our two-level selection scheme as a complement to the LRU replacement decision.

C. Cache Line Size

Metadata requirements for both cache associativities can be reduced by using larger cache lines. This is a result of reducing the need of per-block tag, as many blocks are now represented with the same tag. By doing so, metadata footprint can be reduced by up to 1.90$\times$ and 6.87$\times$, for the direct-mapped and the fully-set-associative cache, respectively. In addition, larger cache lines can benefit workloads that exhibit good spatial locality while smaller cache lines benefit more random workloads. A less obvious implication is that larger cache lines also benefit the flash translation layers (FTL) of SSDs. A large number of small data requests can quickly overwhelm most FTLs, since finding a relatively empty page to write to is becoming increasingly difficult. Finally, using larger cache lines has latency implications, as discussed next.

D. I/O Concurrency

Modern storage systems exhibit a high degree of I/O concurrency, having multiple outstanding I/O requests. This allows overlapping I/O with computation, effectively hiding the I/O latency. To sustain a high degree of asynchrony, *Azor* uses callback handlers instead of blocking, waiting for I/O completion. In addition, *Azor* allows concurrent accesses on the same cache line by using a form of reader-writer locks, similar to the buffer-cache mechanism. Since using a lock for each cache line prohibitively increases metadata memory footprint, *Azor* only tracks *pending* I/O requests.

Caching HDD blocks to SSDs has another implication for I/O response time: Read misses incur an *additional* write I/O to the SSD when performing a cache fill. Once the missing cache line is read from the HDD into DRAM, the buffers of the initial request are filled and *Azor* can perform the cache fill by either (a) re-using the initial application I/O buffers for the write I/O, or (b) by creating a new request and copying the filled buffers from the initial request.

Although the first approach is simpler to implement, it increases the effective I/O latency because the issuer must wait for the SSD write to complete. On the other hand, the second approach completely removes the overhead of the additional cache fill I/O, as the initial request is completed after the buffer copy and then the cache fill write request is asynchronously issued to the SSD. However, this introduces a *memory copy* in the I/O path, and requires maintaining state for each pending cache write. In our design, we adopt the second approach, as the memory throughput in our setup is an order of magnitude higher than the sustained I/O throughput. However, other SSD caching implementations, such as in storage controllers, may decide differently, based on their available hardware resources.

Handling write misses is complicated in the case of larger cache lines when only part of the cache line is modified: the missing part of the cache line must first be read from the HDD in memory, merged with the new part, and *then* written to the SSD. We have experimentally found that this approach disproportionally increases the write miss latency without providing significant hit ratio benefits. Therefore, we support *partially valid* cache lines by maintaining valid and dirty bits for each block inside the cache line.

For write requests forwarded to both HDDs and SSDs, the issuer is notified of completion when the HDDs finish with the I/O. Although this increases latency, it is unavoidable since *Azor* starts with a cold cache in case of failures. Therefore, the up-to-date blocks must *always* be located on the HDDs, to protect against data corruption.

III. Experimental Methodology

We perform our evaluation on a server-type x86-based system, equipped with a Tyan S5397 motherboard, two quad-core Intel Xeon 5400 64-bit processors running at 2 GHz, 32 GB of DDR-II DRAM, twelve 500-GB Western Digital WD5001AALS-00L3B2 SATA-II disks connected on an Areca ARC-1680D-IX-12 SAS/SATA storage controller, and four 32-GB enterprise-grade Intel X25-E (SLC NAND Flash), connected on the motherboard's SATA-II controller. The OS installed is CentOS 5.5, with the 64-bit 2.6.18-194.32.1.el5 kernel version. The storage controller's cache is set to write-through mode. Both HDDs and SSDs are arranged in a RAID-0 configurations, the first using the Areca hardware RAID, and the latter using the `MD` Linux driver with a chunk-size of 64 KB. We use the XFS filesystem with a block-size of 4KB, mounted using the *inode64*, *nobarrier* options. We do not use flash-specific filesystems like jffs2 since they assume direct access to the flash memory, and our SSDs export a SATA-II interface. Moreover, the SSD device controller implements in firmware a significant portion of the functionality of these filesystems. The database server used is MySQL 5.0.77.

We focus our evaluation on I/O-bound operating conditions, where the I/O system has to sustain high request rates. In some cases, we limit the available DRAM memory, in order to put more pressure on the I/O subsystem. For our evaluation, we use three I/O-intensive benchmarks: TPC-H, SPECsfs2008, and Hammerora, the parameters of which are discussed next.

1) TPC-H [15]: is a data-warehousing benchmark that issues business analytics queries to a database with sales information. We execute queries Q1 to Q12, Q14 to Q16, Q19, and Q22 back to back and in this order. We use a 28 GB database, of which 13 GB are data files, and vary the size of the SSD cache to hold 100% (28 GB), 50% (14 GB), and 25% (7 GB) of the database, respectively. TPC-H does a negligible amount of writes, mostly consisting of updates to file-access timestamps. Thus, the choice of the write policy is not important for TPC-H, considering we start execution of the queries with a cold cache.We set the DRAM size to 4 GB, and examine how the SSD cache size affects performance.

2) SPECsfs2008 [16]: emulates the operation of an NFSv3/CIFS file server; our experiments use the CIFS protocol. In SPECsfs2008, a set of increasing *performance targets* is set, each one expressed in CIFS operations-per-second. The file set size is proportional to the performance target ($\approx$120 MB per operation/sec). SPECsfs2008 reports the number of CIFS operations-per-second actually achieved, as well as average response time per operation. For our experiments, we set the first performance target at 500 CIFS ops/sec, and then increase the load up to 15,000 CIFS ops/sec. The DRAM size is set to 32 GB. Contrary to TPC-H, SPECsfs2008 produces a significant amount of write requests, so we examine, along with associativity, the impact of the write policy on performance. We use two cache sizes, of 64 and 32 GB, respectively.

3) TPC-C [17]: is an OLTP benchmark, simulating order processing for a wholesale parts supplier and its customers. This workload issues a mix of several concurrent short transactions, both read-only and update-intensive. The performance number reported by this benchmark is New Order Transactions Per Minute (NOTPM). We use the Hammerora [18] load generator on a 155-GB database that corresponds to a TPC-C configuration with 3,000 warehouses. We run experiments with 512 virtual users, each executing 1,000 transactions. As with TPC-H, we limit system memory to 4 GB.

IV. Experimental Results

In this section we first examine how the Two-Level Block Selection Mechanism (2LBS) improves the performance of our SSD cache. Then, we analyze how four design parameters: 1) *cache associativity*, 2) *cache size* 3) *write policy*, and 4) *cache line size* affect the performance of our system.

A. Block Selection Scheme

For this case study we select cases that exhibit a fair amount of conflict misses, since that is when we expect our two-level block selection scheme to benefit performance. Thus, we do not explore trivial cases, such as having the whole workload fitting in the SSD cache, for which no additional performance benefit can be acquired. We analyze how each level of our proposed scheme separately improves performance, as well as the additional performance gains by combining them. We compare the performance of an SSD cache that uses the block selection scheme with: i) native HDDs runs, and ii) an LRU base cache. The base cache does not use neither levels of the 2LBS scheme and employs the *write-hdd-upd* write policy (the best choice as we show in Section IV-B). For the two designs (2LBS and base), we analyze the performance of both the direct-mapped and LRU-based fully-set-associative caches.

1) TPC-H: Since this benchmark performs a negligible amount of writes. both the file-set size and the number of files do not grow during workload execution. Thus, *Azor* receives a minimal amount of filesystem metadata I/Os. Consequently, pinning filesystem metadata on the SSD cache provides no performance benefit for workloads like TPC-H.

Figure 2 shows our results when using *Azor*'s 2LBS scheme for TPC-H. In all these experiments, *Azor* starts with a cold cache, using 4 GB of DRAM. Since TPC-H is very sensitive to DRAM, for our 2LBS scheme we allocate extra DRAM, as much as required. We use the medium size (14 GB) direct-mapped (DM) and fully-set-associative (FA) caches as a test case. As shown in Figure 2(a) the use of the block selection mechanism improves the performance of the direct-mapped and the fully-set-associative caches by 1.95$\times$ and 1.53$\times$, respectively. More interesting is the fact that the medium size (14 GB) direct-mapped 2LBS cache performs better than the large size (28 GB) base cache counterpart. This is because the medium-size 2LBS design caches more important data than the large size cache, for a lower hit ratio (Figure 2(b)), and for 1.9% less disk utilization (Figure 2(c)). However, the same behavior is not reproduced for the fully-set-associative cache, since this cache design employes the LRU replacement policy, which provides better performance for the larger cache.

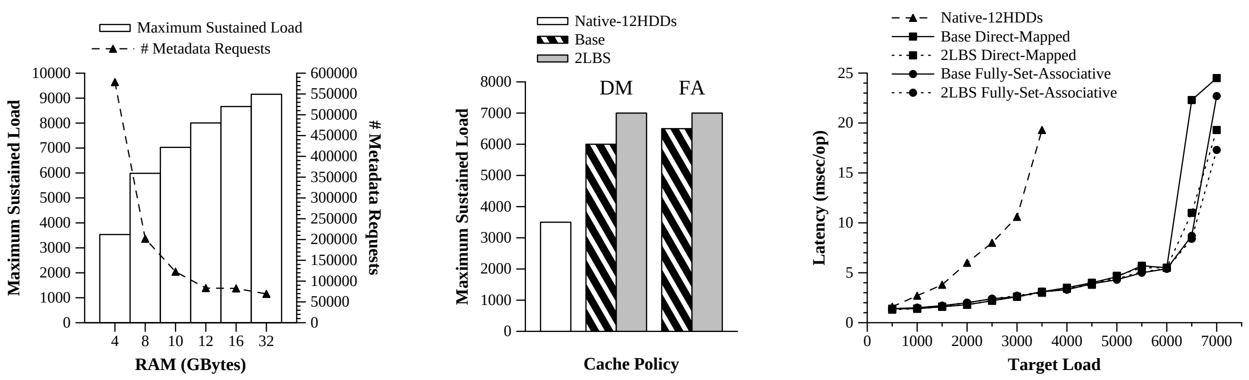


Fig. 3. SPECsfs2008 results: (a) Impact of filesystem metadata DRAM misses on performance. (b)+(c) Impact of the 2LBS scheme with 4 GB DRAM. The write policy used is the *write-hdd-upd* and *Azor* starts with a cold cache.

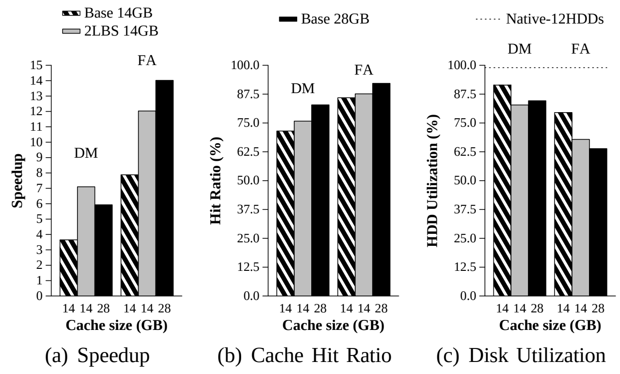


Fig. 2. Impact of block selection scheme on TPC-H, for both the direct-mapped (DM) and fully-set-associative (FA) caches.

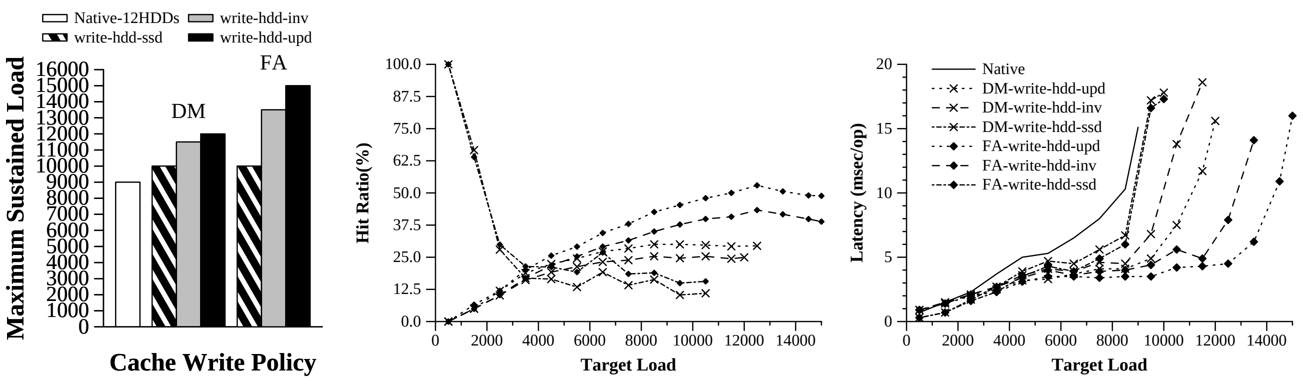
2) SPECsfs2008: Contrary to TPC-H, SPECsfs2008 equally accesses filesystem data blocks and thus, using running estimates of blocks accesses can not further improve performance. On the other hand, the file-set produced by SPECsfs2008 continuously increases during workload execution and thus, the metadata footprint continuously increases as well. Consequently, we argue that system performance is affected by the filesystem metadata DRAM hit ratio. To validate this assumption, we run SPECsfs2008 on the native 12 HDDs setup, while varying the available DRAM size. Figure 3(a) shows that, as the DRAM size increases, the number of metadata I/Os that reach *Azor* significantly decreases, providing substantial performance gains. This is evident when moving from 4 GB to 8 GB of DRAM; a 186% reduction in metadata requests results in 71% better performance. Thus, we expect to gain significant performance improvements for SPECsfs2008 by pinning filesystem metadata on SSDs.

For our experiments, we choose the worst-case scenario with 4 GB DRAM, using the best write policy (*write-hdd-upd*), and starting with an 128 GB cold cache. Since SPECsfs2008 is less sensitive to DRAM for filesystem data caching, we do not allocate further memory for the 2LBS scheme. Figure 3(b) shows that even the base version of *Azor* significantly improves performance, achieving a speedup of 1.71× and 1.85× for the direct-mapped and fully-set-associative caches, respectively. Furthremore, by pinning filesystem metadata on SSDs, performance further improves by 16% and 7% for the two associativities, respectively. These improvents are accompanied by a significant decrease in latency. Figure 3(c) shows that *Azor* supports roughly 3,000 more operations per second for the same latency, compared to the native 12 HDDs run. In addition, there is a 21% and 26% decrease in latency for the direct-mapped and fully-set-associative cache designs, respectively, when comparing the base with the 2LBS version of *Azor* at the last sustainable target load (7000 ops/sec). This, however, is not a result of an increase in hit ratio (not shown), but only of pinning filesystem metadata on SSDs.

3) Hammerora: Finally, we examine how our two-level block selection scheme performs when faced with a black-box workload. For this purpose, we use Hammerora, 4 GB of DRAM, and a cold cache, large enough to hold half the TPC-C database (77.5 GB). Since Hammerora is an OLTP workload, we expect *Azor* to receive a significant amount of write requests, hence we choose our best write policy (*write-hdd-upd*) for our experiments. Our results show that even the base version of *Azor* improves performance by 20% and 55%, for the direct-mapped and fully-set-associative cache designs, respectively. In addition, with the 2LBS scheme performance *further* improves by 31% and 34% for the two associativities, respectively. Not both levels of the 2LBS scheme equally benefit Hammerora: when the two levels are applied individually on the fully-set-associative cache, there is 9% and 24% performance improvement respectively, compared to the base version. As with SPECsfs2008, although there is no change in the hit ratio between the base and the 2LBS versions for both associativities, the performance benefits are a result of which HDD blocks are cached. For this workload, disk utilization is at least 97%, while cache utilization remains under 7% for all configurations. These results reveal that SSD caches can greatly improve OLTP workloads, especially when a large percentage of the database fits in the cache.

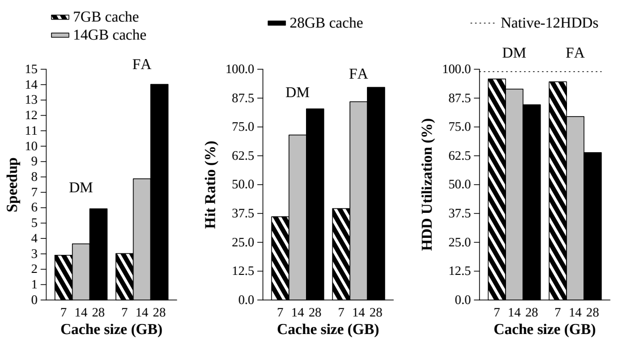
B. System Design Parameters

In this section we analyze how *cache associativity*, *cache size*, and the *write policy* affect the performance of *Azor*. Then, we present our observations on using *larger cache lines*. We



(a) Maximum Sustained Load (b) Hit Ratio (c) Latency

Fig. 5. Impact of associativity and write policy on SPECsfs2008 with 128 GB cache size.



(a) Speedup (b) Hit Ratio (c) HDD Utilization

Fig. 4. Impact of different associativities and cache sizes on TPC-H.

perform these experiments without the 2LBS scheme, so that the impact of these parameters becomes more evident.

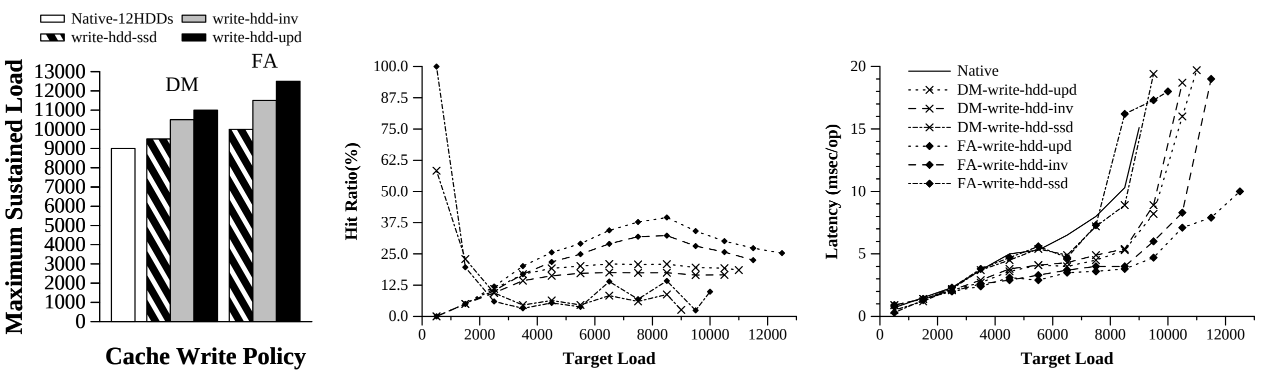
1) TPC-H: Figure 4 shows the performance gains of *Azor*, compared to the native HDDs. In all experiments, *Azor* starts with a cold cache and uses 4 GB of DRAM. Figure 4(a) shows that performance improves along with larger cache sizes, both for the direct-mapped and the fully-set-associative cache. The maximum performance benefit gained is 14.02×, when all the workload fits in the SSD cache, compared to the HDDs.

Cache associativity greatly affects performance; when the workload does not fit entirely in the SSD cache, a medium size (14 GB) fully-set-associative cache performs better than all of the direct-mapped counterparts (7, 14, and 28 GB), by giving a 2.71×, 2.16× and 1.33× higher performance, respectively. Generally, the fully-set-associative cache performs better due to higher hit ratio, shown in Figure 4(b). This is because the fully-set-associative cache has significantly less conflict misses that the direct mapped counter-part, due to the spread-out mapping the latter exhibits. This benefit, however, diminishes as the cache size decreases, evident by the fact that for the smallest cache size (7 GB) the two associativities perform roughly equally. In this case, the 3.54% difference at hit ratio results in 3% better performance, because the significantly increased number of conflict misses has absorbed a large percentage of potential benefits from using an SSD cache.

Furthermore, Figure 4(c) shows that even the slightest decrease in HDD utilization results in significant performance benefits. For instance, the fully-set-associative medium size cache (14 GB) has 11.89% less HDD utilization than the small size (7 GB) counterpart, resulting in a 4.23× better speedup. Generally, HDD utilization is reduced, as the percentage of workload that fits in the cache increases. SSD utilization remains under 7% in all configurations. Moreover, we must mention that the native SSD run achieves a 38.81× speedup, compared to HDDs. Finally, TPC-H is very sensitive to the DRAM size. Performance is exponentially improved, as the percentage of the workload that fits in DRAM is increased. For instance, in case the whole workload fits in DRAM, the achieved speedup is 168.8×. By combining all the above observations, we conclude that the choice of a proper DRAM size along with enough SSD space can lead to optimal performance gains for archival database benchmarks, such as TPC-H.

2) SPECsfs2008: For this workload we compare the performance of *Azor* with the native 12 HDD run, using 32 GB DRAM, and performing all experiments starting with a cold cache. We expect the choice of write policy to significantly affect performance, since this workload produces a fair amount of write requests. Furthermore, since SPECsfs2008 produces a very large number of small files during its execution, we expect the effect of the spread-out mapping the direct-mapped cache exhibits to be more evident in this workload.

Figure 5(a) presents our results using 128 GB of SSD cache. We notice that, depending on the write policy chosen, the speedup gained by *Azor* varies from 11% to 33% and from 10% to 63%, for the direct-mapped and fully-set-associative cache designs, respectively. The performance gains are directly dependent on the hit ratio, shown in Figure 5(b), achieved by each write policy. The *write-hdd-ssd* write policy achieves the lowest hit ratio, hence the lowest performance improvement. This is because SPECsfs2008 produces a huge file-set but only access 30% of it. Thus, useful data blocks are evicted, overwritten by blocks that are never be read. Furthermore, because SPECsfs2008 exhibits a modify-read access pattern, the *write-hdd-upd* write policy exhibits better hit ratio than *write-hdd-inv*, since the first will update the corresponding blocks present in the SSD cache, while the latter will essentially



(a) Maximum Sustained Load (b) Hit Ratio (c) Latency

Fig. 6. Impact of associativity and write policy on SPECsfs2008 with 64 GB cache size.

evict them. Cache associativity also affects performance: the best write policy (*write-hdd-upd*) for the fully-set-associative cache performs 25% better than its direct-mapped counterpart, a result of the increased hit ratio.

Figure 5(c) shows that the response time per operation also improves with higher hit ratios: the better the hit ratio, the longer it takes for the storage system to get overwhelmed and, thus, it can satisfy greater target loads. Furthermore, CPU utilization (not shown) always remains below 25%, showing that the small random writes that SPECsfs2008 exhibits make HDDs the main performance bottleneck. HDD utilization is always 100%, while cache utilization remains below 25% for all configurations. Based on these observations, we conclude that even for write-intensive benchmarks, such as SPECsfs2008, that produce huge file sets the addition of SSDs as HDD caches holds great performance potential.

Finally, we examine how reducing the cache size affects performance. We run again our experiments, this time using 64 GB of SSD cache. We notice that, although the behavior of the write-policies remain the same (Figure 6(a)), *Azor* now becomes saturated earlier in the execution of the workload. This is due to the increased latencies (Figure 6(c)) observed, and the fact that the hit ratio (Figure 6(b)) starts to drop in earlier target loads. Still, there is a 22% and 38% performance improvement, compared to the native HDD run, for the direct-mapped and the fully-set-associative caches, respectively.

3) Impact of Cache Line Size on Performance: Our I/O workloads generally exhibit poor spatial locality, hence cache lines larger than one block (4 KB) result in lower hit ratio. Thus, the benefits described in II-C are not enough to amortize the impact on performance of this lost hit ratio, hence performance *always* degrades. However, we believe that larger cache lines may eventually compensate the lost performance in the long term due to better interaction with the SSD's metadata management techniques in their flash translation layers (FTL).

V. Discussion

A. Metadata memory footprint

The DRAM space required by *Azor* in each case is shown in Table II. The *base cache* design is the LRU-based fully-set-associative cache. The 2LBS scheme offers additional gains to the base cache, and the best associativity in this case is shown in parenthesis. We see that, at the cost of consuming a considerable amount of DRAM in some cases, *Azor* provides significant performance improvement. Furthermore, the DRAM space required scales with the size of the SSD cache size, not with the capacity of the underlying HDDs. Thus, we expect the DRAM space requirements for metadata to remain moderate. However, if DRAM requirements are an issue for some systems, *Azor* can trade DRAM space with performance, by using larger cache lines as described in Section IV-B3.

TABLE II
Trading off DRAM space for performance in *Azor*.

	TPC-H	SPECsfs2008	Hammerora
Base Cache Metadata Footprint	1.28 (DM) / 6.03 (FSA) MB / GB of SSD		
Base Cache Max. Performance Gain	14.02×	63%	55%
Additional Total Metadata for 2LBS	28 MB	No overhead	56 MB
Max. Performance Gain with 2LBS	95% (DM)	16% (DM)	34% (FSA)

Finally, we argue that the cost/benefit trade-off between DRAM size and SSD capacity, only affects workloads sensitive to DRAM, such as TPC-H. On the contrary, for workloads like TPC-C, additional DRAM has less impact as we observed in experiments not reported in this paper. These experiments show that DRAM hit ratio remains below 4.7%, even if DRAM size is quadrupled to 16 GB. Similarly, for SPECsfs2008, additional DRAM serves only to improve the hit ratio for filesystem metadata, as shown in Figure 3(a).

B. Using Azor *within disk controllers and storage protocols*

Azor's 2LBS scheme is feasible within disk controllers by embedding *Azor*'s metadata flag within the (network) storage protocol (e.g. SCSI) command packets transmitted from storage initiators to storage targets. Storage protocols have unused fields/commands that can carry this information. Then, *Azor* will be implemented in the storage controller (target in a networked environment) by using per-block access counters. The main issue in this case is standardization of storage protocols, and whether it makes sense to push hints from higher layers to lower. As our work shows, there is merit to such an approach ([19], [20]).

C. *Using* Azor *2LBS scheme with other filesystems*

Our 2LBS scheme requires modifying the filesystem implementation. We have modifed XFS to mark metadata elements. However, transparent detection of metadata is needed is some setups, e.g. in virtual machines where the hypervisor cannot have access to block identity information. We have developed a mechanism for automatically detecting filesystem metadata without any modifications to the filesystem itself, using the metadata magic numbers for this purpose. Preliminary results with the benchmarks used in this paper show that this mechanism adds negligible overheads to the common I/O path.

D. SSD cache persistence

Azor makes extensive use of metadata to keep track of block placement. Our system, like most traditional block-level systems, does not update metadata in the common I/O path, thus avoiding the necessary additional synchronous I/O. *Azor* does not guarantees metadata consistency after a failure: in this case *Azor* starts with a cold cache. This is possible because our *write-through* policy ensures that all data have their latest copy in the HDD. If the SSD cache has to survive failures, this would require trading-off higher performance with consistency to execute the required synchronous I/O in the common path. However, we choose to optimize the common path at the expense of starting with a cold cache after a failure.

E. FTL and wear-leveling

Given that SSD controllers currently do not expose any block state information, we rely on the flash translation layer (FTL) implementation within the SSD for wear-leveling. Designing block-level drivers and file-systems in a manner cooperative to SSD FTLs which improves wear-leveling and reduces FTL overhead is an important direction, especially while raw access to SSDs is not provided by vendors to system software. Our write policies may significantly affect wear-leveling, however, we leave such an analysis for future work.

VI. Related Work

The authors in [1] examine whether SSDs can fully replace HDDs in data-center environments. They conclude that SSDs are not a cost-effective technology for this purpose, yet. Given current tradeoffs, mixed SSD and HDD environments are more attractive. A similar recommendation, from a more performance-oriented point of view, is given in [21]. Although studies of SSD internals and their performance properties [22], [23] show promise for improved SSDs, we still expect mixed-device storage environments to become increasingly common.

`FlaZ` [24] transparently compresses cached blocks in a direct-mapped SSD-cache, presenting techniques for hiding the CPU overhead from compression. In this work, we take the view that mixed-device storage environments will become common. However, we argue that, beyond any benefits from increasing the effective cache size, the admission and replacement policies will have a critical impact on application performance. Furthermore, we believe that such policies will become even more prominent when dealing with mixed workloads running on the same server.

A recent development in the Linux kernel is the `bcache` block-caching subsystem. Similar to our work, `bcache` is transparent to applications, operating below the filesystem. However, it does not enforce any admission control, which is the main focus of this paper. The `ReadyBoost` feature [25] aims to optimize application-perceived performance with prefetching (`Superfetch` feature) and static file preloading. In contrast, *Azor* dynamically and transparently adapts to the workload, by tracking the block access frequency.

In addition, there are several flash-specific filesystem implementations available for the Linux kernel (JFFS, LogFS, YAFFS) that are mostly oriented to embedded systems. Server workloads require much larger device sizes and therefore a mixed-device (SSDs and HDDs) storage environment is more appropriate. In addition, it is important to address resource consumption issues, such as in-memory metadata footprint, and to sustain much higher degrees of I/O concurrency. These issues point towards tuning filesystem design to the properties of high-performance SSDs, such as PCI-Express devices [26], with a careful division of labor between systems and SSDs, an approach discussed in ([27], [28]).

Flash-based caching has started to appear in enterprise-grade storage arrays. HotZone [29], and MaxIQ [30] are two recent examples. EMC's FAST-Cache [31] utilizes SSD devices as a transparent caching layer. As with our work, FAST-Cache is a LRU cache that serves both reads and writes. However, contrary to *Azor*, writes are not directly written to the cache, while policies are system-defined and cannot be changed by the user. L2ARC [21] is a SSD-based cache for the `ZFS` filesystem, operating below the DRAM-based cache. L2ARC amortizes the cost of SSD write over large I/O, by speculatively pushing out blocks from the DRAM-cache. Similarly to *Azor*, L2ARC takes into account the requirement for in-memory book-keeping metadata. Next, the differentiation between filesystem data and metadata blocks is present in NetApp's Performance Acceleration Module (PAM) [32]. Like *Azor*, PAM aims to accelerate reads, and can be configured to accept only filesystem metadata (as marked by NetApp's proprietary WAFL filesystem). However, PAM requires specialized hardware, while *Azor* is a software layer.

Finally, there has been extensive work on cache replacement policies for storage systems [14], more recently focusing on SSD-specific complications. BPLRU [33] attempts to establish a desirable write pattern for SSDs, via RAM buffering. The LRU list is dynamically adjusted for this purpose, taking into consideration the erase-block size. CFLRU [34] keeps a certain amount of dirty pages in the page cache to reduce the number of flash write operations. BPLRU and CFLRU show the benefit from adjusting LRU-based eviction decisions based on run-time conditions. However, they do not explicitly track properties of the reference stream. LRU-k [35] discriminates between frequently referenced and infrequently referenced pages, by keeping page access history even after page eviction. This is a key insight, allowing adaptation to the prevailing patterns in the reference stream, but comes at the cost of potentially unbound memory space consumption. In this work, we consider how to augment the LRU replacement policy with a two-level selection scheme which rewards or penalizes blocks based on the expected benefit from their continued residence in the SSD-cache. This is a notion similar to that of *marginal gain* used in database buffer allocation in [36].

VII. Conclusions

In this work we examine how SSDs can be used in the I/O path to increase storage performance. We present the design and implementation of *Azor*, a system that transparently caches data in dedicated SSD partitions, as they flow between DRAM and HDDs. Our base design provides various choices for associativity, write and cache line policies, while maintaining a high degree of I/O concurrency. Our main contribution concerns exploring differentiation of HDD blocks according to their expected importance on system performance. For this purpose, we design and analyze a two-level block selection scheme that dynamically differentiates HDD blocks before placing them in the SSD cache.

We evaluate *Azor* using three I/O intensive benchmarks: TPC-H, SPECsfs2008, and Hammerora. We show that at the cost of additional metadata footprint, performance of SSD caching improves when moving to higher way associativities, while the proper choice of the write policy can make up to 50% difference in performance. Furthermore, when there is a significant number of conflict misses, our scheme can significantly improve workload performance, up to 95%. Our mechanism may consume more DRAM in some cases, but results in significant performance benefits. Not both levels of this scheme benefit all workloads. However, they never degrade performance, when used together or in isolation. Overall, our work shows that differentiation of blocks is a promising technique for improving SSD-based I/O caches.

Acknowledgments

Yannis Klonatos thanks Dimitrios Apostolou for his help with naming the proposed architecture. We also thankfully acknowledge the support of the European Commission under the 6th and 7th Framework Programs through the IOLanes (FP7-STREP-248615), HiPEAC (NoE-004408), and HiPEAC2 (FP7-ICT-217068) projects.

References

[1] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to SSDs: analysis of tradeoffs," in *EuroSys '09*. ACM, pp. 145–158.

[2] T. Kgil and T. Mudge, "FlashCache: a NAND flash memory file cache for low power web servers," in *CASES '06*. ACM, pp. 103–112.

[3] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *SIGMOD '08*. ACM, pp. 1075–1086.

[4] X. Ouyang, S. Marcarelli, and D. K. Panda, "Enhancing Checkpoint Performance with Staging IO and SSD," in *IEEE SNAPI'10*, pp. 13–20.

[5] H. J. Lee, K. H. Lee, and S. H. Noh, "Augmenting RAID with an SSD for energy relief," in *HotPower '08*. USENIX Association, pp. 12–12.

[6] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," *Transactions on Storage*, vol. 3, 2007.

[7] D. Roselli and T. E. Anderson, "A comparison of file system workloads," in *ATC '00*. USENIX Association, pp. 41–54.

[8] V. Prabhakaran et al., "IRON file systems," in *SOSP '05*. ACM.

[9] I. H. Doh et al., "Impact of NVRAM write cache for file system metadata on I/O performance in embedded systems," in *SAC '09*. ACM.

[10] J. Piernas, T. Cortes, and J. M. García, "DualFS: a new journaling file system without meta-data duplication," in *ICS '02*. ACM, pp. 137–146.

[11] N. Megiddo and D. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *FAST'03*. USENIX Association, pp. 115–130.

[12] D. Chinner, "Details of space allocation in the XFS filesystem (private communication)," June 2010.

[13] R. Jenkins, "32 bit integer hash function," http://www.concentric .net/~Ttwang/tech/inthash.htm, 2007.

[14] R. B. Gramacy, M. K. Warmuth, S. A. Brandt, and I. Ari, "Adaptive Caching by Refetching," in *In Advances in Neural Information Processing Systems*. MIT Press, 2003, pp. 1465–1472.

[15] Transaction Processing Performance Council, "TPC-H: an ad-hoc, decision support benchmark," 1993.

[16] Standard Performance Evaluation Corporation, "SPECsfs2008 benchmark suite measuring file server throughput and response time."

[17] Transaction Processing Performance Council, "TPC-C, an on-line transaction processing benchmark," 1992.

[18] S. Shaw, "Hammerora: the open source oracle load test tool," http://hammerora.sourceforge.net/index.html, 2010.

[19] T. E. Denehy, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Bridging the Information Gap in Storage Protocol Stacks," in *ATC' 02*. USENIX Association, pp. 177–190.

[20] N. Kirsch, "Isilon's OneFS Operating System white paper," http://www.isilon.com/onefs-operating-system, August 2010.

[21] A. Leventhal, "Flash storage memory," *Commun. ACM*, vol. 51, no. 7, pp. 47–51, July 2008.

[22] 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 *ISCA '09*. ACM, pp. 279–289.

[23] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *ATC'08*. USENIX Association, pp. 57–70.

[24] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas, "Using Transparent Compression to Improve SSD-based I/O Caches," in *EuroSys '10*. ACM, pp. 1–14.

[25] Microsoft Corp., "Explore the features: Windows ReadyBoost," www.microsoft.com/windows/windows-vista/features/readyboost.aspx.

[26] Fusion-io, "Solid State Storage – A New Standard for Enterprise-Class Reliability," http://www.dpie.com/manuals/storage/ fusionio/Whitepaper_Solidstatestorage2.pdf.

[27] A. Rajimwale, V. Prabhakaran, and J. D. Davis, "Block management in solid-state devices," in *ATC '09*. USENIX Association.

[28] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, "DFS: A File System for Virtualized Flash Storage," in *USENIX FAST '10*.

[29] FalconStor, "HotZone - Maximize the performance of your SAN," http://www.nasi.com/hotZone.php.

[30] Adaptec Inc., "MaxIQ SSD cache performance kit," 2009.

[31] EMC corp, "EMC CLARiiON and Celerra Unified FAST Cache," http://www.emc.com/collateral/software/white-papers/h8046-clariion-celerra-unified-fast-cache-wp.pdf, 2010.

[32] D. Tanis, N. Patel, and P. Updike, "The NetApp Performance Acceleration Module," http://www.netapp.com/us/communities/tech-ontap/pam.html, 2008.

[33] H. Kim and S. Ahn, "BPLRU: a buffer management scheme for improving random writes in flash storage," in *FAST'08*. USENIX Association, pp. 1–14.

[34] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "CFLRU: a replacement algorithm for flash memory," in *CASES '06*. ACM, pp. 234–241.

[35] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *SIGMOD '93*. ACM, pp. 297–306.

[36] R. Ng, C. Faloutsos, and T. Sellis, "Flexible and Adaptable Buffer Management Techniques for Database Management Systems," *Transactions on Computers*, vol. 44, pp. 546–560, 1995.