Computer Science Department
University of Crete

Design and Evaluation of Solid-State Drive (SSD) Caches to Improve Storage I/O Performance

Master of Science Thesis

Yannis Klonatos

February 2011

Heraklion, Greece

University of Crete
Computer Science Department

Design and Evaluation of Solid-State Drive (SSD) Caches to Improve Storage I/O Performance

Thesis submitted by
Yannis Klonatos
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Yannis Klonatos

Committee approvals: _____
Angelos Bilas
Associate Professor, Thesis Supervisor

_____
Kostas Magoutis
Professor

_____
Dimitris Nikolopoulos
Professor

Director of Graduate Studies: _____
Angelos Bilas
Associate Professor

Heraklion, February 2011

To my grandparents,
Yannis and Elsa Megagiannis,
who provided to me the best childhood
a person can hope for.

Στους παππούδες μου,
Γιάννη και Έλσα Μεγαγιάννη,
που μου έδωσαν την καλύτερη παιδική
ηλικία που κάποιος μπορεί να έχει.

Abstract

Flash-based solid state drives (SSDs) exhibit potential for solving I/O bottlenecks by offering superior performance over hard disks for several workloads. For this reason, there is recently significant work towards integrating SSD caching in the I/O stack of modern storage systems. However, the proposed solutions are usually application specific.

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 targeting on maintaining a high degree of I/O concurrency. Our main contribution is that we explore how *Azor* can differentiate HDD blocks according to their expected importance on system performance. We design and analyze a two-level block selection scheme which 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 realistic workloads and large problem sizes. We use a server-type platform and four I/O intensive workloads: TPC-H, SPECsfs2008, PostMark, and Hammerora. Our results show that as the cache size increases, our SSD-cache can enhance I/O performance by up to 14.02×, 1.63×, 1.72× and 55% for each workload respectively. In addition, our two-level block selection scheme can further enhance I/O performance compared to a typical SSD cache by up to 95%, 16%, 28%, and 34% for each workload, respectively. Not both levels of our two-level block selection scheme benefit all workloads. However, they never degrade performance, when used together or in isolation.

Supervisor Professor: Angelos Bilas

This work was performed at the Foundation for Research and Technology Hellas (FORTH), Institute of Computer Science (ICS), 100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece. The work is partially supported by the 7th European Commission Framework Programme through the IOLanes (FP7-STREP-248615), HiPEAC (FP7-NoE-004408), and HiPEAC2 (FP7-ICT-217068) projects.

Acknowledgments

I am grateful that I had Professor Angelos Bilas as supervisor of my MSc thesis. His profound knowledge about many areas of computer science has always been an excellent guide to my progress. I hope someday I can inspire and guide a student as well as he did for me.

This work wouldn't have finished if I didn't have the endless help of three people. First, I am eternally obliged to Emmanuel (*Aprosarmostos*) Marazakis for his immense (mainly psychological) help in this thesis. He really is the best, most cordial and unselfish man I have ever met. Then, i owe many thanks to my co-author Thanos Makatos, who has been not only my colleague but also my friend. Thanos was always there to console me in all these moments when my work seemed hopeless and endless. Finally, I must thank Michael D. Flouris for the countless hours he spent debugging my code. To his disappointment, I must confess that my code is not (nor ever will be) bug-free.

The friends I made in Heraklion these last 6 years have always been a sounding board for me when I was looking to complain to someone for a problem. For their patience - which I believe I exhausted - I gratefully thank Michalis Alvanos, Vangelis Kafentarakis, Manalis Nikiforos, Kostas Mousikos, Dimitris Papadeas, Zoe Sebepou and Markos (*Akatasxetos*) Fountoulakis. Their advice have always been the voice of reason in my often unreasonable complaints.

People in Computer Architecture and Systems VLSI (CARV) lab all contributed to have a perfect working environment. Many thanks to Michael Lygerakis, George Kalokairinos, Dimitris Apostolou, Yannis Manousakis, Kostas Xasapis, Stelios Mavridis, Charalampos Vatsolakis, George Tzenakis, Dimitris Tsaliagos, George Nikiforos, Shoaib Akram and our dog Azor (the lab's mascot). Because of them, for many many days work and fun in the laboratory didn't seem to abstain so much.

For reasons they know better, I sincerely thank Vaggelis Karaplios, Anne, George Payatakis, Xrisostomos Antoniadis and Danae-Christina Komitopoulou. To quote the words of W.H. Auden, "The friends who met here and embraced are gone, each to his own mistake."

Although I may be an only child, I always felt that Vasilis and Loretta Kapetanidi, as well as Afroditi Theoxari had always been my brother and sisters. They have my deepest admiration, and I have always been of the opinion that there is hope for the world as long as there are still people like them walking on it.

Finally, I really want to thank my parents, Yannis Klonatos and Eyantia Megagianni, as well as my "second" parents Yannis Antonakos and Efstathia Salamaliki. Regardless of what I am going to do next, they will always be my moral compass in life. Their support, but mostly their inexplicable faith in me, is always pushing me trying to do something better. I am here today mostly because of them.

Yannis Klonatos
Heraklion, February 2011

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is an increasing need for high-performance storage I/O in modern server systems. This is a result of mainly three reasons. First, there has lately been a significant increase in the working set sizes for most applications, causing lower DRAM hit ratios, even with the best prefetching algorithms applied. In addition, the performance of the storage subsystems has historically increased in considerably lower rates compared to those achieved by CPUs and DRAM. As a result, the percentage of time an application spends on I/O has also increaced, possibly canceling the benefits acquired by increased computational speeds [11]. Finally, new workloads and marketing approaches, such as server virtualization, have significantly increased the demands on storage, usually making the hard disk drives (HDDs) the performance bottleneck of server environments [38].

The performance characteristics of current generation NAND-Flash solid-state drives (SSDs), shown in Table 1.1, make these devices 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 com-

mon HDDs, albeit at a higher cost per GB [29]. In addition, SSDs bear complexity caveats, related to their internal organization and operational properties. For these reasons, 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).

	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

TABLE 1.1: HDD and SSD performance metrics.

To this point, there has been recently work on how to improve I/O performance using SSD caches. FlashCache [20] proposes the use of flash memory as a secondary file cache for web servers. [24] examines whether SSDs can improve the performance of transaction processing. Furthermore, [33] examines how SSDs can be used to improve check-pointing performance, while [23] examines how to use SSD 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. Where all the aforementioned approaches 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. Although an SSD cache bear similarities to the corresponding DRAM-based design, there are significant differences as well. While exploring these differences, we also investigate the

following problems:

Metadata footprint, for representing the state of the SSD cache blocks in DRAM. This footprint must be taken into account, since SSD caches are significantly larger than DRAM caches, and considering the increasing size of SSDs. However, the metadata requirements grow proportionally to the SSD capacity available, rather than the much larger HDD space; still, compacting this metadata to fit in DRAM is an important concern. 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 organization that allows more informed block replacement decisions at the cost of consuming more DRAM space for its metadata. The impact of the block mapping policy is not as clear as in smaller DRAM caches. Traditionally, DRAM caches use a fully-set-associative policy since the small cache size requires reducing capacity conflicts. However, SSD-based caches may be able to use a simpler mapping policy, thus reducing access overhead without increasing capacity conflicts. 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.

Second, we quantify 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.

We consider this experimental study important, since the metadata footprint in DRAM will keep increasing with SSD device capacities, thereby making high-associativity organizations less cost-effective.

Write-handling policies, for which we explore the following dimensions: *a*) write-through vs. write-back; this dimension affects not only performance, but also system reliability, *b*) write-invalidation vs. write-update in the event of cache hit for write requests, and, finally, *c*) write-allocation, in the event of cache misses for write requests. 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.

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. In addition, *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. For this reason, we compact this data structure enough so that it fits in the limited DRAM space.

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 a significant increase in the number of files and in particular, small-size files [2]; therefore, the impact of filesystem metadata accesses is expected to become even more pronounced. To differentiate metadata from data blocks we mark metadata accesses in the filesystem code (*XFS*), within the Linux kernel. 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.

We implement *Azor* as a virtual block device in the Linux kernel and evaluate our design with three realistic workloads: TPC-H, SPECsfs2008, and PostMark. 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× to 14.02×, from 1.11× to 1.63×, and from 1.12× to 1.72× for each workload, 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%, 16%, and 23% for each workload, respectively. Our block admission mechanism does not achieve performance improvement when the entire workload fits in the SSD cache, since the hit ratio is already maximized. 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 the block selection scheme, *Azor* can improve performance up to 89%.

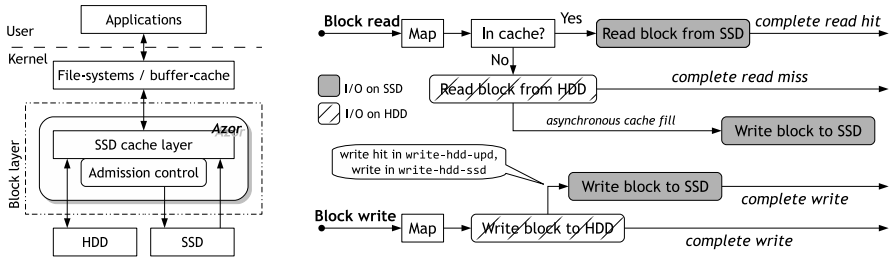
The rest of this thesis is organized as follows. Section 2 presents our design for resolving the aforementioned challenges without affecting access latency and I/O concurrency. Section 3 presents our experimental platform, representative of a current generation server for I/O intensive workloads.

Section 4 presents a detailed evaluation, based on long-running experiments with the aforementioned workloads. Section 5 discusses some further considerations and future work for *Azor* while section 6 provides a comparison with related work. Finally, Section 7 concludes the work, summarizing our main findings.

Chapter 2

System Design

The use of SSDs as I/O caches in our architecture is shown in Figure 2.1(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, effectively providing a *second level* of caching for HDD blocks. 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.



(a) System architecture (b) Read and write paths

FIGURE 2.1: *Azor* system architecture and I/O paths.

Figure 2.1(b) shows how *Azor* handles 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 initial request to the SSD (*read hit*). Otherwise, data are fetched from the HDD (*read miss*) and an asynchronous SSD write operation (*cache fill*) is scheduled.

For writes (hits or misses), *Azor* implements a *write-through* mechanism. We opted against using a *write-back* cache; a write-back cache 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 would result in data loss. Our *write-through* design avoids this issue as well. Our system 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, *Azor* can either update (*write-hdd-upd*) or invalidate (*write-hdd-inv*) the corresponding cache block. The choice of write policy has significant implications for workloads that have a fair amount of write operations, as we show in our evaluation.

2.1 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" [6]. Such techniques result in unnecessary conflict misses due to data placement, as we show in our evaluation.

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. Furthermore, such a cache organization fully resolves the data placement issue, thus reducing conflict misses.Our fully-set-associative cache requires 6.04 MB of cache metadata per GB of SSD, 4.7× more than the direct-mapped counterpart. Metadata requirements for this cache 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 software cache for SSDs appears to be deceptively simple. Our experience shows that implementing such a cache is far from trivial and it requires dealing with the following 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 would be the case in a hardware implementation. This mechanism must also be scalable enough, in order to always avoid the CPU becoming the bottleneck. *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 [17]. The list of cache blocks is then traversed, looking for a match. This arrangement minimizes the number of possible cache blocks that must be examined for each incoming I/O request. We have experimentally found that setting the number of buckets to $\frac{1}{4}$ of the number of SSD blocks provides optimal performance. Our design scales with the size of the SSD cache, and only requires the number of buckets to be increased in case more cache space becomes available.

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 [16], 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 have opted for the LRU policy, since it provides a reasonable reference point for other more sophisticated policies. Since our SSD-cache operates below the buffer-cache, we believe this was a reasonable choice for beginning the evaluation of SSD-cache mechanisms. We postpone for future work a study of the interaction between the buffer-cache and the SSD-cache, for a variety of

replacement policies.

Finally, we have tried to minimize the metadata footprint required by our data structure. This is important, since a large number of elements like 64-bit pointers may result in significant memory overheads. In our solution, we embed all collision lists, along with the LRU list, in a single table. By doing so, all pointers can be replaced with indexes in the table, thus reducing metadata footprint. With this scheme, the buckets that point to elements of the collisions lists table, are using indexes as well.

2.2 Cache Line Size

Metadata requirement for both the direct-mapped and the fully-set-associative cache designs can be reduced with the use of larger cache lines. By doing so, metadata footprint can be reduced by up to 1.90× and 6.87×, for the two cache organizations, respectively.

This is a result of reducing the need of per-block tag, as many blocks can be now represented with the same tag. 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. By using larger cache lines, we simplify this task as well. Finally, the use of larger cache lines has latency implications, as discussed in the next section.

2.3 I/O Concurrency

Modern storage systems exhibit a high degree of I/O concurrency, having multiple outstanding I/O requests. This offers opportunities for overlapping I/O with computation, effectively hiding the I/O latency. To allow for 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 to proceed in parallel 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 in the SSD cache incur an *additional* write I/O to the SSD when performing a cache fill operation. Once the missing cache line is read from the HDD into DRAM, the buffers of the initial requests 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 request to the SSD, 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 experimental 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.

Finally, one last important observation concerns the number of threads *Azor* employs for *cache fill* operations. We have experimentally found out that one such thread suffices for the degree of concurrency exhibited by current server systems and, thus, we have used only one thread in our evaluation. However, we point that, in case future systems exhibit significantly larger degrees of concurrency, *Azor* could easily be adjusted to support it, by setting a larger number of cache fill threads.

2.4 Admission Control Mechanism

Azor differentiates HDD blocks based on their expected importance to system performance. To do so, *Azor* uses a two-level block selection scheme which controls whether or not a specific HDD block should be admitted to the SSD cache, according to its importance. We design our two-level selection scheme as a complement to the LRU replacement decision, in the spirit of more sophisticated replacement. 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 because it becomes increasingly difficult for file-systems to rely solely on DRAM for metadata caching. This is a result of a significant increase in filesystem capacities in recent years, with the average file size remaining small [2]. This problem is further amplified by the introduction of new metadata types by the file-systems, mainly to achieve higher data protection. This, for example, requires checksum integration in the filesystem [36], an approach already adopted by state-of-the art filesystems, such as ZFS and BTRFS. Thus, it makes sense to dedicate faster devices for storing filesystem metadata [10, 35], since there will be performance benefits from the decrease in latency in future metadata reads. Finally, most filesystems perform synchronous metadata writes to maintain integrity in the event of system failures. These synchronous operations can have such a marked performance penalty that several common filesystem implementations offer the option of ignoring certain update dependencies [15]. To this point, modern file-systems, such as XFS, provide journaling for file system metadata, where file system updates are first written to a serial journal in DRAM before the actual disk blocks are updated. The required performance characteristics makes logging a suitable candidate for the use of an SSD device. Our metadata marking scheme also handles this issue, since the log is marked as metadata as well.

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 flags field of the I/O request descriptor (`struct bio` in the Linux kernel). Then, *Azor* uses this information at the block level to categorize each HDD block. Our modification does not affect filesystem performance and can easily be implemented

in other filesystems as well. However, the SSD cache now needs to maintain an additional *class* bit for each SSD cache block.

Next, at a second level of the admission 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 [43]. 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 the above observations, we differentiate data blocks for SSD caching purposes, based on a running estimate of 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. Whereas 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 [27], 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 indexed by the HDD block number. The DRAM required for maintaining the reference 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, with the exception of a TPC-H workload that is sensitive to the available DRAM size.

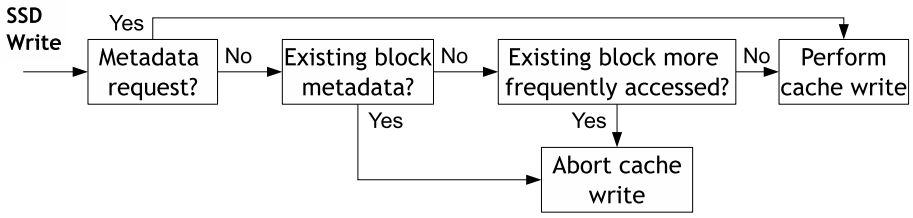


FIGURE 2.2: *Azor* admission control path.

Figure 2.2 shows how this decision is made. 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. First, *Azor* checks if the block in question is already occupied by any HDD block. If not, the cache block is empty, so the write to SSD operation can proceed as usual. Then, 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 already contains filesystem metadata. If so, the cache fill is aborted, else both the incoming request and the corresponding cache block contains data. In this case *Azor* checks

which one is accessed more times, and the cache fill is performed (or aborted) accordingly.

Chapter 3

Experimental Platform

We perform our evaluation on a server-type x86-based Linux system. Our system is 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) SSDs connected on the motherboard's SATA-II controller. The OS installed is CentOS 5.3, with the 64-bit 2.6.18-128.1.6.el5 kernel version. The storage controller's cache is set to write-through mode. We use the XFS filesystem with a block-size of 4 KB, 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 device controller in our SSDs implements in firmware a significant portion of the functionality provided by most available flash filesystems. In our setup, both HDDs and SSDs are arranged in a RAID-0 configurations, the first using the hardware RAID provided by the Areca controller, and the latter using the `MD` Linux driver with a chunk-size of 64 KB. In some cases, we

limit the available DRAM memory, in order to put more pressure on the I/O subsystem. We do so via the kernel boot option *mem=*.

We focus our evaluation on I/O-bound operating conditions, where the I/O system has to sustain high request rates. We evaluate the benefit from the SSD caching layer, as well as the trade-offs of our design. As a result, we have not used write-only workloads, as the benefit of our SSD cache is only visible when the read I/O volume comprises a fair portion of the total traffic. For our evaluation, we use four I/O-intensive benchmarks: TPC-H, SPECsfs2008, PostMark, and Hammerora. Since our experimental results are reproducible with negligible differences among subsequent runs, we have omitted standard deviation markings in all graphs. Next we briefly discuss each workload and the parameters we use.

TPC-H [45] 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. The database server used in our experiments is MySQL 5.0.77. TPC-H does a negligible amount of writes, mostly consisting of updates to file-access timestamps and other control operations. Thus, the choice of the write policy is not important for TPC-H, considering we start execution of the queries with a cold cache. For this workload, we set the DRAM size to 4 GB, and we examine how the SSD cache size affects performance.

SPECsfs2008 [40] 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. For each performance target, read/writes operations, both of data

blocks and metadata-related accesses to the filesystem, are executed over a file set of a size proportional to the performance target (≈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. For this workload we use two SSD cache sizes, of 64 and 32 GB, both built using 4 SSD devices.

TPC-C [44] 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 main performance number reported by this benchmark is New Order Transaction Per Minute (NOTPM). We use the Hammerora [39] load generator on top of a 155-GB MySQL database that corresponds to a TPC-C configuration with 3,000 warehouses. We run experiments with 512 virtual users, each one executing 100,000 transactions. As with PostMark, we limit system memory to 4 GB.

PostMark [19] is a synthetic, filesystem benchmark that simulates a mail server by creating a pool of continually changing files over the file-system, and measures transaction rates and throughput. These transactions consist of (i) a create or delete file operation and (ii) a read or append file operation. Each transaction type and its affected files are chosen randomly. When all transactions complete, the remaining files are deleted. We use version 1.51 of the benchmark and we employ a write dominated configuration with the default 35-65% read-write ratio. Since PostMark is originally single-

threaded, to acquire higher I/O rates we use eight concurrent instances of the workload. For each instance we use 10,000 mailboxes between 4 KB and 1 MB in size, resulting in a initial file set of 54 GB, and execute 100,000 transactions with 16 KB read/write accesses. We use three cache sizes, holding 100% (54 GB), 50% (37 GB), and 25% (13.5 GB) of the file-set, respectively. For these experiments, we limit system memory to 4 GB.

Chapter 4

Experimental Evaluation

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

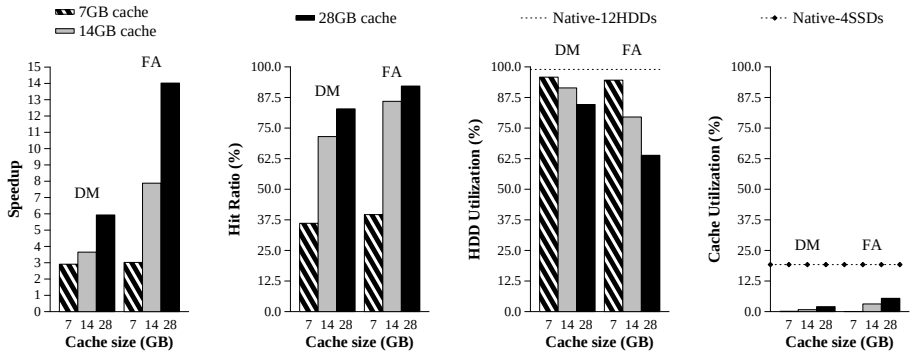
4.1 System Design Parameters

For the analysis of the parameters of design space, we perform all experiments with the 2LBS scheme turned off, so that the impact of the other parameters becomes more evident.

4.1.1 TPC-H

Figure 4.1 shows the performance impact of *Azor* when compared to the native HDD. In all TPC-H experiments, *Azor* starts with a cold cache and uses 4 GB of DRAM. Overall, as shown in Figure 4.1(a), performance improves along with larger cache sizes, both for the direct-mapped and the fully-set-associative cache designs. The maximum performance benefit gained by SSD caching is 14.02×, when all the workload fits in the SSD cache, compared to

the HDDs. Cache associativity greatly affects performance, especially when the workload does not fit entirely in the SSD cache: a medium size (14 GB) fully-set-associative cache provides better performance 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. For the smallest cache size (7 GB) the two associativities perform roughly equally.



(a) Speedup (b) Hit Ratio (c) HDD Utilization (d) SSD Util.

FIGURE 4.1: Impact of different associativities and cache sizes on TPC-H.

Generally, the fully-set-associative cache achieves higher performance due to higher hit ratio, shown in Figure 4.1(b). Specifically, the medium size (14 GB) fully-set-associative cache achieves a 14.41% and 3.08% better hit ratio than the medium and large (28 GB) direct-mapped caches, respectively. This is due to the fact that the fully-set-associative cache has significantly less conflict misses than the direct mapped. However, this benefit diminishes as the cache size decreases, a fact most evident in the two small (7 GB) caches. In this case, the 3.54% difference at hit ratio results in 3% better performance. This is because the significantly increased number of conflict misses has absorbed a large percentage of potential benefits from using an SSD cache.

Furthermore, Figure 4.1(c) shows that even the slightest decrease in

HDD utilization results in significant performance benefits. More significantly, for the medium cache (14 GB), the fully-set-associative cache reduces HDD utilization by 11.89%, resulting in a 4.23× better speedup. Generally, HDD utilization is reduced, as the percentage of workload that fits in the SSD cache increases. SSD utilization, shown in Figure 4.1(d), remains under 7% in all configurations. Finally, we must mention that in case that the whole workload was in the SSDs, then the achieved speedup is 38.81×.

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.

4.1.2 SPECsfs2008

Figures 4.2 and 4.3 shows our results for SPECsfs2008, compared to the native HDD run, when using 128 GB as SSD cache and 32 GB of DRAM. As with TPC-H, all SPECsfs2008 experiments are performed with a cold cache. However, contrary to TPC-H, we expect the choice of write policy to have a significant impact on performance for SPECsfs2008, since this workload produces a fair amount of write requests. Furthermore, we expect the effect of the spread-out mapping the direct-mapped cache exhibits to be more evident in this workload, since SPECsfs2008 produces a very large number of small files during its execution.

As shown in Figure 4.2, depending on the write policy, 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 *write-*

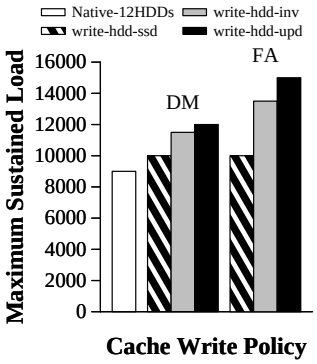
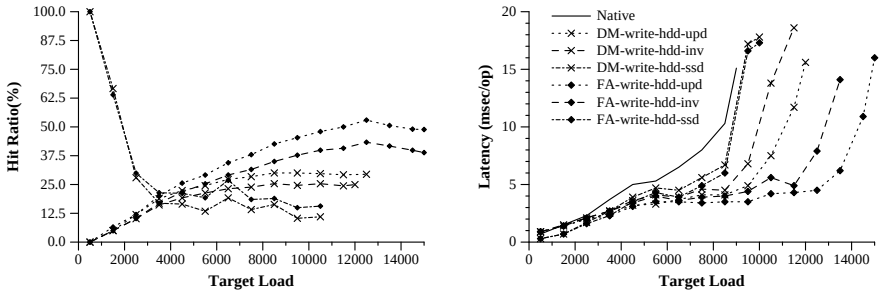


FIGURE 4.2: Impact of associativity and write policy on the maximum sustained target load in SPECsfs2008 with 128 GB cache size.

hdd-ssd write policy achieves the lowest hit ratio, shown in Figure 4.3(a), hence the lowest performance improvement. This is due to the fact that SPECsfs2008 produces a huge file-set but only access 30% of it and, 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 evict them. Cache associativity also affects performance: the best write policy (*write-hdd-upd*) for the fully-set-associative version performs 25% better than its direct-mapped counterpart, as a result of increased hit ratio.

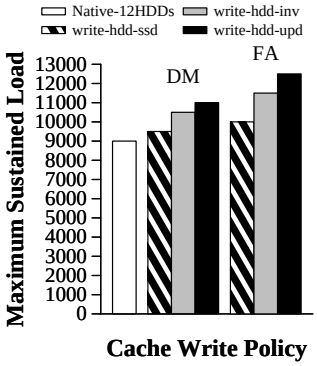
Figure 4.3(b) shows that the response time per operation also improves with higher hit ratios: the better the hit ratio, the longer it takes for the storage system to get overwhelmed and, thus, it can satisfy greater target loads. In the same figure, we see that issuing writes to both devices (*write-hdd-ssd* policy), results in a latency behavior very similar to the native HDD run. Furthermore, CPU utilization (not shown) always remains below 25%, showing that the small random writes that SPECsfs2008 exhibits make



(a) Hit Ratio (b) Latency

FIGURE 4.3: Impact of associativity and write policy on hit ratio and latency in SPECsfs2008 with 128 GB cache size.

HDDs the main performance bottleneck. HDD utilization (not shown) is always 100%, while cache utilization (not shown) remains below 25% for all configurations. Based to all above observations, we conclude that even for benchmarks, such as SPECsfs2008, that produce huge file sets and produce a fair amount of write requests, the addition of SSDs as disk caches have great potential for improving performance.

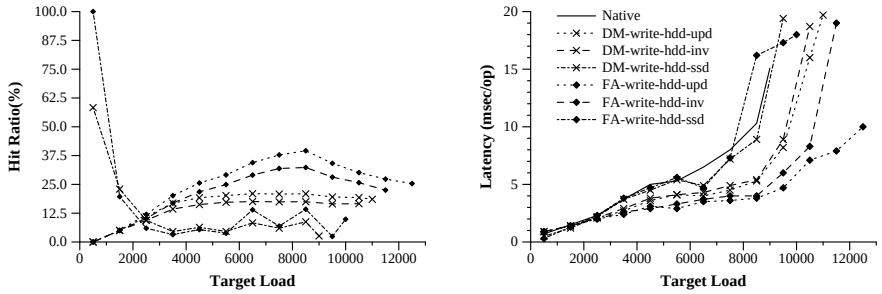


(a) Maximum Sustained Load

FIGURE 4.4: Impact of associativity and write policy on maximum sustained target load for SPECsfs2008 with 64 GB cache size.

Finally, we examine how reducing cache size affects performance. We execute again the same experiments, but this time we use a 64 GB SSD cache. Figure 4.4 shows that, although the behavior of the write-policies

remains the same as before, there are less performance benefits from the SSD cache. The *write-hdd-upd* write policy remains the best choice, resulting in a 22% and 38% better performance than the native HDD run for the direct-mapped and the fully-set-associative caches, respectively. As shown in Figure 4.5(b), *Azor* becomes saturated earlier in the execution of the workload than the previous case, resulting in the hit ratio (Figure 4.5(a)) starting to drop at earlier target loads than before.



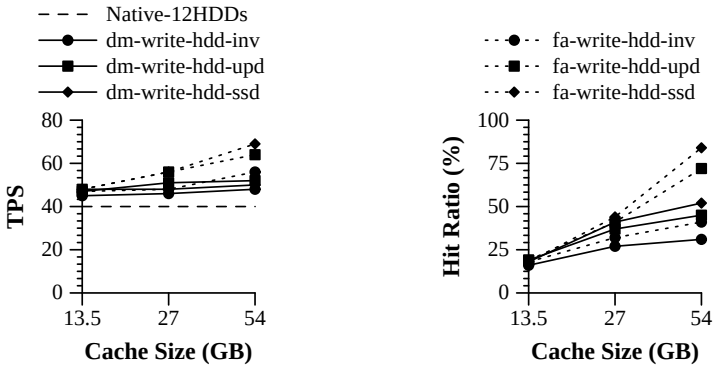
(a) Hit Ratio (b) Latency

FIGURE 4.5: Impact of associativity and write policy on hit ratio and latency for SPECsfs2008 with 64 GB cache size.

4.1.3 PostMark

Figure 4.6 shows our results for PostMark using eight concurrent instances of the workload, 4 GB of RAM and starting with a cold cache. Figure 4.6(a) shows that *Azor* improves performance by up to 72% compared to the native 12 HDDs configuration. Using SSD caches improve performance even when a small percentage of the workload fits in the cache, up to 20% for this cache size (13.5 GB). Furthermore, the fully-set associative version of *Azor* performs better than the direct-mapped counterpart, a result of the improved hit ratio, shown in Figure 4.6(b). Increasing the available cache size benefits the fully-set-associative cache significantly more than the direct-mapped cache. We believe this is, as with TPC-H, due to the spread-out mapping

the direct-mapped cache exhibits, which results in a significant increase in conflict misses. However, as the available cache size is decreased, both associativities perform equally. The same observation applies to the write policies as well.



(a) Transactions per Second

(b) Hit Ratio

FIGURE 4.6: PostMark using eight concurrent instances.

Finally, HDD utilization (not shown) remains constantly to 100%, while cache utilization (not shown) is always under 12% even when writes are propagated to both devices (*write-hdd-ssd* policy). As with SPECsfs2008, this fact shows that the addition of SSDs as HDD caches have great potential for improving performance.

4.1.4 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 2.2 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).

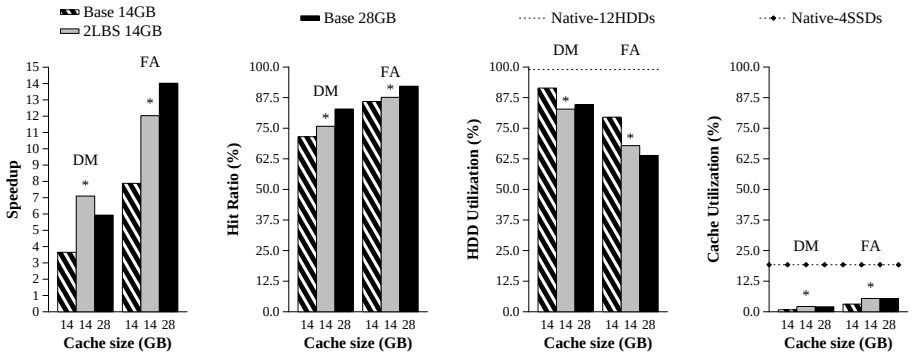
4.2 Block Selection Filter

In this section, we examine how the use of our two-level Block Selection Mechanism (2LBS) improves performance of our SSD cache. For this case study we have selected cases where there are 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 how much further performance can be improved by combining them. We compare the performance of an SSD cache that uses the block selection scheme with: i) the native HDDs runs, and with ii) a base cache, which does not use the 2LBS scheme, employs the *write-hdd-upd* write policy (the best choice as we show in Section 4.1), while it may have its size fixed. For the two designs (2LBS and base), we analyze the performance of both the direct-mapped and LRU-based fully-set-associative caches.

4.2.1 TPC-H

TPC-H does a negligible amount of writes. As a result, both the file-set size and the number of files don't grow during workload execution. Thus, *Azor* receives a minimal amount of I/Os containing filesystem metadata. Consequently, the mechanism concerning filesystem metadata pinning on the SSD cache provides no performance benefit for workloads, such as TPC-H.

Figure 4.7 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 two-level mecha-



(a) Speedup (b) Cache Hit Ratio (c) Disk Utilization (d) Cache Utilization

FIGURE 4.7: Impact of block selection scheme on TPC-H, for both the direct-mapped (DM) and fully-set-associative (FA) caches. Bars with a star on top of them are the 2LBS versions of *Azor*.

nism 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 4.7(a) the use of the block selection mechanism improves the performance of the direct-mapped and the fully-set-associative caches by 1.95× and 1.53×, 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 due to the fact that, although the adaptive version does not produce a better hit ratio (shown in Figure 4.7(b)), it still manages to cache more important data than the large size base cache version. This results in 1.9% less disk utilization, shown in Figure 4.7(c). However, the same behavior is not reproduced to the fully-set-associative cache. This is due to the LRU replacement policy this cache design employs, and which also provides better performance for the larger cache.

4.2.2 SPECsfs2008

Contrary to TPC-H, the file-set produced by SPECsfs2008 continuously increases during the workload execution. As a result, we expect the filesystem metadata footprint to continuously increase as well, and the performance of the workload to be consequently affected by the filesystem metadata DRAM hit ratio. Furthermore, SPECsfs2008 equally accesses filesystem data blocks and thus, the use of the running estimate of blocks accesses does not further improve performance. To validate our assumption about the significance of metadata DRAM hit ratio on performance, we run SPECsfs2008 on the native 12 HDDs setup, while varying the available DRAM size. Our results are shown in Figure 4.8. As the DRAM size increases, the number of metadata I/Os that reach *Azor* significantly decreases, resulting in substantial performance gains. This becomes more evident when moving from 4 GB to 8 GB of DRAM; a 186% reduction in the number of metadata requests results in 71% better performance. With these observations, we expect significant performance improvement in SPECsfs2008 when employing our filesystem metadata pinning mechanism.

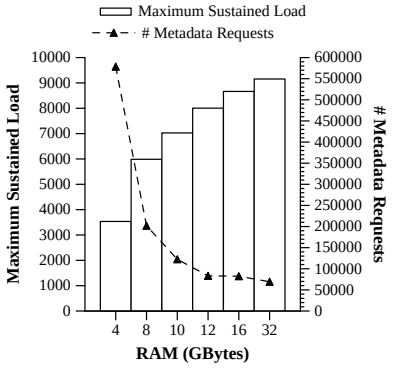


FIGURE 4.8: Impact of filesystem metadata DRAM misses on performance for SPECsfs2008.

For our experiments with *Azor*, we choose the worst-case scenario with 4

GB of DRAM, using the best write policy (*write-hdd-upd*), and we examine how performance is improved, comparing the base version with the 2LBS version of our SSD cache, and with the native 12 HDDs run. As with all the previous tests, *Azor* runs starting with a cold cache. Furthermore, SPECsfs2008 is less sensitive to DRAM for filesystem data caching, so we do not allocate further memory for our 2LBS scheme.

Figure 4.9(a) 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. The same figure shows that employing the filesystem metadata pinning mechanism further improves performance by 16% and 7% for the two cache associativities, respectively. Figure 4.9(b) shows that the improved performance is accompanied by a sig-

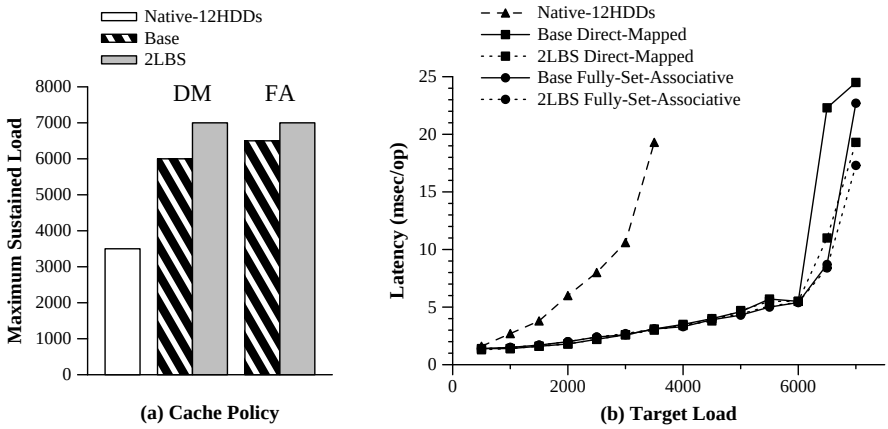


FIGURE 4.9: Impact of the two-level block selection scheme for SPECsfs2008 with 4 GB DRAM. The write policy used is the *write-hdd-upd*.

nificant decrease in latency: *Azor* supports roughly 3,000 more operations per second for the same latency when compared to the native 12 HDDs run. Furthermore, when comparing the base with the 2LBS versions of *Azor* at the last sustainable target load (7000 ops/sec), we see that there is a 21% and 26% decrease in latency for the direct-mapped and fully-set-associative

cache designs, respectively. This is not, however, a result of the increased hit ratio (not shown), equal for the base and the 2LBS cache designs, but a result of only the *type* of blocks cached (filesystem metadata).

4.2.3 PostMark

For PostMark we expect *Azor* to receive a small number of filesystem metadata I/Os. This is because the workload creates the file-set at the beginning, and only appends data to files during the transaction phase. Consequently, as with TPC-H, the filesystem metadata pinning mechanism provides negligible performance benefits.

In these experiments, we present our results of the base and 2LBS cache designs for the eight instance PostMark. We run our experiments starting with a cold cache, using the small (13.5 GB) and medium (27 GB) SSD cache, and we examine how our adaptive mechanism improves performance. As with SPECsfs2008, we do not use any extra DRAM for our mechanism, since PostMark is not sensitive to DRAM size. We have also chosen the best write policy (*write-hdd-upd*).

As shown in Figure 4.10(a), the 2LBS scheme provides further performance improvement by 16% for the small cache (13.5 GB) for both associativities, and by 21% and 28% for the medium cache (27 GB) both for the direct-mapped cache and fully-set-associative caches, respectively. The hit ratio (not shown) is the same for both versions, hence the performance improvement of the 2LBS version only comes from the type of blocks cached.

4.2.4 Hammerora

Finally, we examine how our 2LBS scheme performs when faced with an "unknown" workload, Hammerora, using 4 GB of DRAM. *Azor* starts with

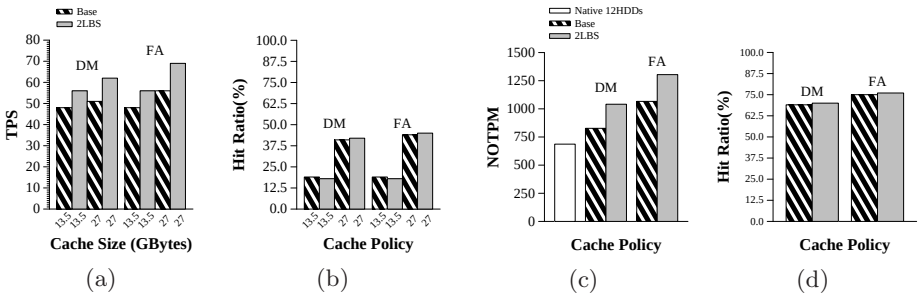


FIGURE 4.10: Impact of the 2LBS scheme on eight instance PostMark ((a)+(b)) and Hammerora workloads ((c)+(d)).

a cold cache, and large enough to hold half the 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*).

Our results for both the base and 2LBS versions of *Azor* is shown in Figure 4.10(b). We see that the base version of *Azor* improves performance by 20% and 55%, for the direct-mapped and fully-set-associative cache designs, respectively. In addition, using the 2LBS scheme, performance further improves by 31% and 34%, for the two cache associativities, respectively. Not both levels of the two-level block selection scheme equally benefit Hammerora: two test runs for the fully-set-associative cache revealed that the two levels executed individually gave 9% and 24% performance improvement respectively, compared to the base version. Furthermore, like SPECsfs2008 and PostMark, although the hit ratio (not shown) between the base and the 2LBS versions for both associativities does not change, the performance benefits are again a result of which HDD blocks are cached. For this workload, disk utilization (not shown) is at least 97%, while cache utilization (not shown) remains under 7% for all configurations. As with TPC-H, these observations reveal that SSDs hold great performance potential for OLTP

workloads, such as Hammerora, especially when a large percentage of the database fits in the cache. Next, we explore how other system parameters affect performance of our SSD-cache design.

Chapter 5

Discussion & Future Work

In this chapter we examine issues that appear to be interesting as a future study as well as some remaining design considerations.

Metadata memory footprint: The DRAM space required by *Azor* in each case is shown in Table 5.1. We see that, at the cost of needing a significant 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 moder-

	Base Cache Metadata Footprint		2LBS total additional metadata	Maximum Performance gain	
	DM	FA		Base Cache	2LBS
TPC-H	1.28MB / SSD GB	6.03MB / SSD GB	28 MByte	14.02×	95% (DM)
SPECsfs			No overhead	63%	16% (DM)
PostMark			1.5GByte	72%	28% (FA)
Hammerora			56 MByte	55%	34% (FA)

TABLE 5.1: Trading off DRAM space for performance in *Azor*. The performance improvements reported by the 2LBS cache are additional to the base version of *Azor*. For the base cache, the maximum performance gain is achieved by using the LRU-based fully-set-associative cache. For the 2LBS scheme, the best associativity is reported in parenthesis.

ate. However, if DRAM requirements are an issue for some systems, *Azor* can trade DRAM space with performance, by using larger cache lines as we mentioned in Section 4.1.4.

Finally, concerning the cost/benefit trade-off between DRAM size and SSD capacity, we argue that this tradeoff 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 thesis. 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 4.9(a).

Applicability of *Azor* behind disk controllers and standard storage protocols: *Azor*'s 2LBS scheme is feasible behind 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, our SSD cache management mechanism will be implemented in the storage controller (target in a networked environment) by using per-block access counters. The main issue is a standardization one, whether it makes sense to push hints from higher layers to lower: In our view, and as our work shows, there is merit to such an approach (cf. [8],[22]).

Using *Azor* 2LBS scheme with other file-systems: An important implication of our work is that it requires modifications at the file-system level. So far we presented results with our modification in the XFS filesystem. We argue that such modifications can easily be applied to other filesystems as well, since each filesystem must have a way to differentiate metadata from data blocks, in order to carry out specific operations (e.g.

repair). However, as future work, we plan to explore possible mechanisms for automatically detecting filesystem metadata without any modifications to the filesystem itself. We already have an approach to this direction using the filesystem metadata magic numbers for this purpose.

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 guarantee metadata consistency after a failure: in this case *Azor* assumes that a failure occurred and 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, thus exploiting the non-volatile properties of SSDs.

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. Furthermore, we cannot directly influence the FTL's block allocation and re-mapping policies. 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. Finally, the choice of write policy may significantly affect SSD performance and wear-leveling related issues, but we chose to leave such analysis for future work.

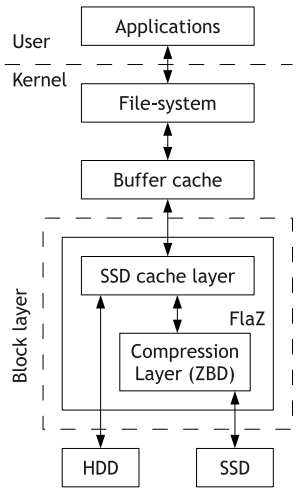


FIGURE 5.1: Compressed Caching System Architecture

Increasing the effective cache size Given the current high cost per gigabyte for SSDs [29], it is important to examine techniques that can increase their cost-efficiency. One such technique is to use transparent online data compression in the I/O path. Data compression [25] reduces the space required to store a piece of data, e.g., a file, block, or other data segment, by storing it in compressed form. The original data can then be reconstructed by decompressing the transformed piece of data. We employ compression along with SSD-based I/O caches in a larger system, called *Flaz*, that uses SSDs as compressed cached in the I/O path. *Flaz* internally consists of two layers, one that achieves transparent compression (*Azor*) and one that uses SSDs as an I/O cache.

We find that even modest increases in the amount of available SSD capacity improve I/O performance, for a variety of server workloads. Transparent block-level compression allows such improvements by increasing the effective SSD capacity, at the cost of increased CPU utilization. Overall, although compression at the block-level introduces significant complexity, our work

shows that online data compression is a promising technique for improving the performance of I/O subsystems.

Flash-Specific File System: During the evaluation of *Azor*, a specific file-system had to be chosen for the experiments. As mentioned in Section 3, we have opted not to use flash-specific filesystems like jffs2 since they assume direct access to the flash memory, and they implement a significant portion of the functionality provided by the firmware of our SSD devices. Furthermore, their current implementations seem to have issues with the large SSD device size, as these file-systems usually target small-sized flash devices, such as USB sticks. To this point, there is currently no flash-specific file system for large-scale SSD devices. Developing such a file-system raises some interesting questions about: 1. Whether such a targeted design can perform better than the current generic-purpose file-systems when used on top of flash memory, 2. The interactions of the file-system with the flash translation layers in SSDs, 3. Whether or not these interactions affect performance, but most importantly 4. The possibility of moving the functionality implemented in firmware to the file-system level, thus canceling the "black box nature" of solid state disks.

SSD caching power analysis: There has been the case of using SSD based I/O caches as a mean to achieve power reduction in large scale clusters. However, in our work we have observed, that even when using the larger cache size available in *Azor*, the HDD utilization is not decreased and always remains equal to 100%. Thus, we believe a quantitive power analysis is necessary to anwser the question whether or not the perfromance benefits acquired are actually traded with increased power consumption in the long term.

Chapter 6

Related Work

The authors in [29] examine whether SSDs can fully replace traditional disks 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 is given in [25], starting from a more performance-oriented point of view. Although studies of SSD internals and their performance properties [9, 3] show promise for improvements in upcoming SSDs, we still expect mixed-device storage environments to become increasingly common.

A recent development in the Linux kernel is the `bcache` block-caching subsystem [41], which is still under development. Similar to our work, `bcache` is transparent to applications, operating below the filesystem. The cache space is organized as a collection of buckets, indexed via a B-Tree data structure. Buckets are intended to match the physical device's erase-blocks, to avoid small random writes on the SSDs. Currently `bcache` enforces no admission control, which is the main focus of this thesis. Similar goals to `bcache` are addressed by a number of commercial products: HotZone [13], and MaxIQ [1] are two recent examples. The `ReadyBoost` feature [28] aims

to optimize performance by caching all HDD writes. In combination with prefetching (`Superfetch` feature) and static file pre-loading tools, it improves the application-perceived performance. In contrast, *Azor* dynamically adapts to the workload, by tracking the block access frequency, rather than relying on prefetching and static pre-loading.

In addition, there are several flash-specific filesystem implementations available for the Linux kernel [46, 4, 12], 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 [14], with a careful division of labor between systems and SSDs, an approach discussed in [37]. Several related implementation challenges, are shown in [18].

`FlaZ` [26] transparently compresses cached blocks in a direct-mapped SSD-cache, presenting techniques for hiding the CPU overhead from compression, for workloads with multiple in-flight I/O operations. 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 promising approach to this problem appears to be a dynamic scheme for partitioning the available SSD-cache space among the competing workload classes [5].

Flash-based caching has started to appear in enterprise-grade storage ar-

rays. EMC's FAST-Cache [7] offers the capability to utilize SSD devices as a transparent caching layer. Similarly to our design, FAST-Cache is a LRU cache that serves both reads and writes, in 64KB extent units. However, contrary to *Azor*, writes are not directly written to the cache, while policies are system-defined and cannot be changed by the user. Next, the differentiation between filesystem data and metadata blocks is present in NetApp's Performance Acceleration Module (PAM) [42]. As with *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.

There has also been extensive work on cache replacement policies for storage systems [16], more recently focusing on SSD-specific complications. BPLRU [21] 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 deliberately 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 [32] 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. LRU-K also introduces the concept of *aging*, by considering the last K references to a page. ARC [27] maintains more state information than LRU (4 lists instead of one), which can become a concern with ever increasing storage system capacities. In this work, we consider how to augment the

LRU replacement policy with a two-level selection scheme which can be seen as rewarding or penalizing blocks based on the expected benefit from their continued residence in the SSD-cache. This is a notion similar to the *marginal gain* notion introduced for database buffer allocation in [30, 31]. In our work, we apply selection criteria at the time of block eviction from the SSD cache.

Finally, L2ARC [25] is a SSD-based cache for the `ZFS` filesystem, operating below the DRAM-based cache. L2ARC speculatively pushes out blocks from the DRAM-cache, to amortize the cost of SSD write over large write I/Os. L2ARC takes into account the requirement for in-memory bookkeeping metadata, a concern which has been a major motivation for our work.

Chapter 7

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. We evaluate our approach using four I/O intensive benchmarks: TPC-H, SPECsfs2008, PostMark and Hammerora.

Our base design provides various choices for associativity, write and cache line policies, while targeting on maintaining a high degree of I/O concurrency. 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.

Our main contribution concerns exploring differentiation of HDD blocks according to their expected importance on system performance. For that purpose, we design and analyze a two-level block selection scheme which dynamically differentiates HDD blocks before placing them in the SSD cache. We find that when there is a significant number of conflict misses, our scheme can significantly improve performance workload, up to 95%. Our proposed

mechanism may consume more DRAM in some cases, however it trades this cost with significant performance benefits. Not both levels of this scheme benefit all workloads, however, they never degrade performance. Overall, our work shows that differentiation of blocks is a promising technique for improving SSD-based I/O caches.

Bibliography

[1] Adaptec Inc. MaxIQ SSD cache performance kit. http://www.adaptec.com/en-US/products/Cloud-Computing/MaxIQ/SSD-Cache-Performance, 2009.

[2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *FAST '07*, pages 3–3, Berkeley, CA, USA, 2007. USENIX Association.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *ATC'08: USENIX 2008 Annual Technical Conference*, pages 57–70, 2008.

[4] Aleph One. Ltd, Embedded Debian: Yaffs: A NAND-Flash Filesystem. www.yaffs.net, 2002.

[5] K. P. Brown, M. Mehta, M. J. Carey, and M. Livny. Towards automated performance tuning for complex workloads. In *VLDB '94*, pages 72–84, San Francisco, CA, USA, 1994. Morgan Kaufmann Inc.

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

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

[8] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *USENIX Annual Technical Conference*, pages 177–190, Berkeley, CA, USA, 2002. USENIX Association.

[9] 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*, pages 279–289. ACM, 2009.

[10] I. H. Doh, H. J. Lee, Y. J. Moon, E. Kim, J. Choi, D. Lee, and S. H. Noh. Impact of NVRAM write cache for file system metadata on I/O performance in embedded systems. In *SAC '09*, pages 1658–1663, New York, NY, USA, 2009. ACM.

[11] E. Jassaud, W. Szoecs. High performance storage for the engineering workflow. http://www.dynamore.de/dokumente/papers-1/2010-deutsches-forum/papers/N-I-03.pdf, 2010.

[12] J. Engel and R. Mertens. LogFS - finally a scalable flash file system. http://logfs.org/git/, 2010.

[13] FalconStor. FalconStor HotZone - Maximize the performance of your SAN (North American Systems International, Inc.). http://www.nasi.com/hotZone.php.

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

[15] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 5, Berkeley, CA, USA, 1994.

[16] R. Gramacy, M. Warmuth, S. A. Brandt, and I. Ari. Adaptive caching by refetching. In *Advances in Neural Information Processing Systems 15*, pages 1465–1472, Dec. 2003.

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

[18] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A File System for Virtualized Flash Storage. In *Proc. USENIX FAST*, pages 85–100, 2010.

[19] J. Katcher. PostMark: A New File System Benchmark. *NetAPP TR3022*, 1997. http://www.netapp.com/ tech_library/3022.html.

[20] T. Kgil and M. Trevor. FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers. In *CASES '06*, pages 103–112. ACM, 2006.

[21] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *FAST'08*, pages 1–14. USENIX Association, 2008.

[22] N. Kirsch. Isilon's onefs operating system white paper. http://www.isilon.com/onefs-operating-system, August 2010.

[23] H. J. Lee, K. H. Lee, and S. H. Noh. Augmenting raid with an ssd for energy relief. In *HotPower'08*, pages 12–12, Berkeley, CA, USA, 2008. USENIX Association.

[24] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *Proc. of SIGMOD '08*, pages 1075–1086. ACM, 2008.

[25] A. Leventhal. Flash storage memory. *Commun. ACM*, 51(7):47–51, 2008.

[26] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas. Using Transparent Compression to Improve SSD-based I/O Caches. In *Proc. EuroSys*, pages 1–14, 2010.

[27] N. Megiddo and D. S. Modha. ARC: A self-tuning, lowoverhead replacement cache. In *Proc. of FAST'03*, pages 115–130. USENIX Association, 2003.

[28] Microsoft. Corporation, Explore the features: Windows ReadyBoost. www.microsoft.com/windows/windows-vista/features/readyboost.aspx, 2006.

[29] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: analysis of tradeoffs. In *EuroSys '09*, pages 145–158. ACM, 2009.

[30] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. *SIGMOD Rec.*, 20(2):387–396, 1991.

[31] R. Ng, C. Faloutsos, and T. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers*, 44:546–560, 1995.

[32] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *SIGMOD Rec.*, 22(2):297–306, 1993.

[33] X. Ouyang, S. Marcarelli, and D. K. Panda. Enhancing Checkpoint Performance with Staging I/O and SSD. *SNAPI '10*, 0:13–20, 2010.

[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*, pages 234–241, New York, NY, USA, 2006. ACM.

[35] J. Piernas, T. Cortes, and J. M. García. DualFS: a new journaling file system without meta-data duplication. In *ICS '02*, pages 137–146, New York, NY, USA, 2002. ACM.

[36] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. *SIGOPS Oper. Syst. Rev.*, 39(5):206–220, 2005.

[37] A. Rajimwale, V. Prabhakaran, and J. D. Davis. Block Management in Solid-State Devices. *9th USENIX Annual Technical Conference*, June 2009.

[38] Russ Fellows. Evaluating storage technologies for virtual server environment. http://www.3par.com/SiteObjects/D4CEA9FD82D18A27FB9F8C1F20-B36A07/EGI_3PAR_block_for_virtual_environments_wp_final.pdf, 2010.

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

[40] SPEC. SPECsfs2008: SPEC's benchmark designed to evaluate the speed and request-handling capabilities of file servers utilizing the NFSv3 and CIFS protocols. http://www.spec.org/sfs2008/, 2008.

[41] W. Stearns and K. Overstreet. Bcache: Caching beyond just RAM. http://lwn.net/Articles/394672, July 2010.

[42] D. Tanis, N. Patel, and P. Updike. The netapp performance acceleration module. http://www.netapp.com/us/communities/tech-ontap/pam.html, September 2008.

[43] I. Tatarinov, A. Rousskov, and V. Soloviev. Static Caching in Web Servers, 1997.

[44] TPC. TPC-C, an on-line transaction processing benchmark. http://www.tpc.org/tpcc/, 1992.

[45] TPC. TPC-H: an ad-hoc, decision support benchmark. http://www.tpc.org/tpch/, 1993.

[46] D. Woodhouse. JFFS: The Journalling Flash File System. http://sourceware.org/jffs2/jffs2-html, 2001.