Flashy Prefetching for High-Performance Flash Drives

Ahsen J. Uppal, Ron C. Chiang, and H. Howie Huang

Department of Electrical and Computer Engineering
George Washington University
{auppal, rclc, howie}@gwu.edu

***Abstract*—While hard drives hold on to the capacity advantage, flash-based solid-state drives (SSD) with high bandwidth and low latency have become good alternatives for I/O-intensive applications. Traditional data prefetching has been primarily designed to improve I/O performance on hard drives. The same techniques, if applied unchanged on flash drives, are likely to either fail to fully utilize SSDs, or interfere with application I/O requests, both of which could result in undesirable application performance. In this work, we demonstrate that data prefetching, when effectively harnessing the high performance of SSDs, can provide significant performance benefits for a wide range of data-intensive applications. The new technique, *flashy prefetching*, consists of accurate prediction of application needs in runtime and adaptive feedback-directed prefetching that scales with application needs, while being considerate to underlying storage devices. We have implemented a real system in Linux and evaluated it on four different SSDs. The results show 65-70% prefetching accuracy and an average 20% speedup on LFS, web search engine traces, BLAST, and TPC-H like benchmarks across various storage drives.**

I. Introduction

The spectrum of storage devices has expanded drastically in the last several years, thanks to the emergence of solid-state drives (SSDs) that are built upon NAND flash memory [6], [13]. As scientific and enterprise data usage continues to grow exponentially, new storage systems that leverage both high performance from SSDs and large capacity from hard drives (HDDs) will likely be in high demand to reduce the I/O performance gap. While for many data-intensive applications moving, "hot" data from HDDs to SSDs (disk swapping and data migration) can easily bring good speedups, in this paper we aim to achieve additional performance benefits from SSDs beyond the simple practice of disk replacement.

Data prefetching [37], [19] is one of, if not the most, widely-used techniques to reduce access latency, because it can load the data that are likely to soon be accessed from storage devices into main memory. Traditional prefetching techniques have been focused on rotational hard drives and are conservative with the amount of data prefetched for good reasons – because data prefetching consumes shared system resources, it is likely that aggressive data prefetching would interfere with normal access and subsequently hinder application performance. As a result, current techniques often leverage the low cost of sequential access on hard drives to read data that resides on the same and nearby tracks. Aggressive prefetching has been considered too risky by many researchers (given long seek penalties, limited HDD bandwidth, and limited system RAM), with one notable exception [36].

For flash-based SSDs, we believe that aggressive prefetching could potentially expedite data requests for many applications. However, as we will demonstrate shortly, simply prefetching as much data as possible does not provide the desired benefits for three main reasons. First, data prefetching on faster devices such as SSDs, if uncontrolled, will take shared I/O bandwidth from existing data accesses (more easily than on slower hard drives). As a side effect, useful cached data may become evicted while main memory would be filled with mispredicted (and unneeded) data while applications were waiting for useful data.

Second, not every device is the same, and this is especially true for SSDs. The performance of an SSD can vary depending on flash type (SLC/MLC), internal organization, memory management, etc. A prefetching algorithm, while reasonably aggressive for a faster drive, could become too aggressive for another drive, slowing down normal execution. Last, not every application is the same – two applications often possess different I/O requirements. A single application can also go through multiple stages, each of which has different I/O requirements. Clearly, care should be taken to avoid adverse effects from both too-conservative and too-aggressive prefetching.

In this work, we propose the technique of *flashy prefetching*[1] for emerging flash devices, which is aware of the runtime environment and can adapt to the changing requirements of both devices and applications as well as its own run-time performance.

The salient features of flashy prefetching include not only taking advantage of the high bandwidth and low latency of SSDs, but also providing inherent support for parallel data accesses and feedback-controlled aggressiveness. To demonstrate its feasibility and benefits, we have implemented a real system called *prefetchd* in Linux that dynamically controls its prefetching aggressiveness at runtime to maximize performance benefits, by making good tradeoffs between data prefetching and resource consumption. Note that we use flashy prefetching and prefetchd interchangeably in this paper. We evaluate prefetchd on four different SSDs with a wide range of data-intensive applications and benchmarks. The prototype

[1] "Flashy: momentarily dazzling." (source: Merriam-Webster) Flashy prefetching aims to effectively harness "flashy", high-performance SSDs.

978-1-4673-1747-4/12/$31.00 © 2013 IEEE

achieves average 20% speedups on LFS, web search engine traces, BLAST, and TPC-H like benchmarks across various storage drives, which we believe largely comes from the 65-70% prefetching accuracy.

The main contributions of this paper are threefold:

- We conduct a comprehensive study of the effects of conservative and aggressive prefetching in the context of heterogeneous devices and applications. The results show that adaptive prefetching is essential for taking advantage of high-performance storage devices like solid-state drives.
- We design the architecture of flashy prefetching such that it self-tunes to prefetch data in a manner that matches application needs without being so aggressive that useful pages are evicted from the cache. Measuring performance metrics in real-time and adjusting the aggressiveness accordingly significantly improves the effectiveness of this approach.
- We develop a Linux-based prototype, prefetchd, that monitors application read requests, predicts which pages are likely to be read in the near future, loads those pages into the system page cache while attempting to not evict other useful pages, monitors its success rate in time and across pages, and adjusts its aggressiveness accordingly.

Note that data prefetching on SSDs has drawn a lot of interest. For example, [46], [9] show that prefetching can be used to improve energy efficiency. Our positive results also demonstrate the performance potential of data prefetching. Another notable work, FAST [25], focuses on shortening application launch times and utilizes prefetching on SSDs for the quick start of various applications. Our approach expands on previous work along multiple dimensions, including employing a feedback-driven control mechanism, handling multiple simultaneous requests across processes, and requiring no application modifications. With a wider range of data-intensive applications in mind, the proposed prefetchd aims to improve the overall performance of generic applications.

The rest of the paper is organized as follows: Section II presents the challenges the prefetching must address and Section III describes our design principles to address those challenges. Section IV presents the architecture of prefetchd and describes each individual component. Section V discusses the implementation in detail. The evaluation is presented in Section VI and related work is discussed in Section VII. We conclude in Section VIII.

II. Three Challenges

A. Challenge #1: SSDs are different

SSDs are clearly different from HDDs in many ways. To name a few: no seek latency, excellent random read and write performance, inherent support for parallel I/O, expensive small writes, and limited erase cycles. At a high level, modern SSDs consist of several components such as NAND flash packages, controllers, and buffers. In this study, we use four different SSDs from two manufacturers (roughly covering two recent generations of SSDs): OCZ Vertex (SSD1) [34] and Vertex2 (SSD2) [35], and Intel X-25M (SSD3) [23] and 510 (SSD4) [24]. We also use a Samsung Spinpoint M7 (HDD) hard drive [39]. If we look at the specifications in the top half of Table I, the specification numbers for SSDs are close, except for the Intel 510 SSD (which comes with the SATA III support, but is limited by the SATA II interface in our test machines).

TABLE I
Specification and Measured Performance

	SSD1 [34]	SSD2 [35]	SSD3 [23]	SSD4 [24]
Capacity	120 GB	120 GB	80 GB	120 GB
Flash Type	MLC	MLC 34nm	MLC 34nm	MLC 25nm
Controller	Indilinx	SandForce	Intel	Marvell
Read BW (Bandwidth)	250 MB/s (max)	285 MB/s (max)	250 MB/s (seq)	450 MB/s (seq)
Write BW	180 MB/s (max)	275 MB/s (max)	100 MB/s (seq)	210 MB/s (seq)
Latency	100 μs	100 μs	65 μs read	65 μs read
Measured Read BW	170 MB/s	170 MB/s	265 MB/s	215 MB/s
Measured Write BW	180 MB/s	203 MB/s	81 MB/s	212 MB/s

However, the differences between different SSDs tend to be subtle, mostly in architectural designs. Note that the same manufacturer may choose to adopt different controllers across two models, which coincidentally was the case for the SSDs chosen in this study. As shown in the bottom half in Table I, when measured under Linux, the four SSDs clearly have higher bandwidth than the hard drive (measured read bandwidth at about 90 MB/s), that is, the four SSDs outperform the hard drive by 189%, 189%, 294%, and 239%, respectively. More importantly, the four SSDs differ noticeably, especially, their measured write bandwidths range from 80 MB/s and 200 MB/s.

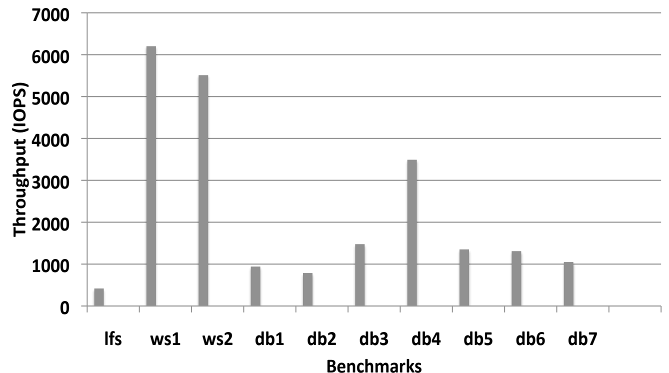
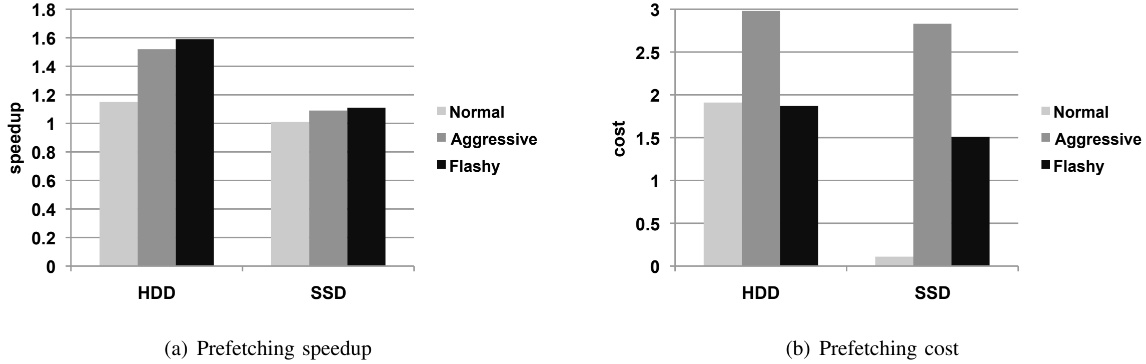


Fig. 1. Application I/O Throughputs

B. Challenge #2: Applications are different

Although data-intensive applications are in dire need of high-performance data access, they tend to have different I/O requirements. Fig. 1 presents the average application throughput in I/O operations per second (IOPS) for ten applications ranging from large file operations, search engine traces,



(a) Prefetching speedup

(b) Prefetching cost

Fig. 2. The Need for Flashy Prefetching for Benchmark dbt3-3 on SSD2

and decision support / database workloads. We will describe these benchmarks in detail in Section VI. The two replayed WebSearch traces reach the highest throughput at about 6,000 IOPS, while at the same time LFS needs an order of magnitude less throughput at 400 IOPS. Furthermore, chances are that each application will likely go through multiple stages, each of which has different I/O requirements.

C. Challenge #3: Prefetching for HDDs and SSDs is Different

Traditional disk drives can read sequential blocks quickly because the head can be stationary while the platter rotates underneath. A read from flash can be completed quickly in a few microseconds, compared to several millisecond seek latencies on hard drives. Note that data saved in SSDs does not necessarily present the same spatial locality as on hard drives. On the other hand, multiple simultaneous access requests for data on an SSD that address different flash chips can be satisfied simultaneously– a challenging task for a hard disk. The internal controllers of SSDs have already taken advantage of this inherent parallelism for high performance I/O [6], and in this work we will show that this parallelism can also be exploited from a higher system level.

Suppose that two applications simultaneously issue sequential read patterns to a hard disk; such patterns are likely to interfere with each other. To satisfy the simultaneous requests, the access patterns must occur on different platters, otherwise the disk heads might move back and forth to different tracks. An I/O scheduler will try to minimize head movements, but this movement overhead still limits the number of simultaneous prefetch operations that can occur on a traditional hard drive. In contrast, parallel I/Os in SSDs can benefit greatly from better hardware structure and organization. Nevertheless, aggressive prefetching on SSDs may not necessarily be optimal even for sequential access because SSDs cannot simply continue to read at the same track or cylinder.

To illustrate the need for going beyond traditional prefetching, we present the performance results in Fig. 2 from two different static prefetching techniques: normal and aggressive; and the proposed flashy prefetching. Here we run a database benchmark on both a HDD and SSDs, where the speedup is measured using elapsed wall-clock time and *cost* is defined as the ratio of the amount of prefetched data to the amount of data read by the application. The details of our evaluation environment can be found in Section VI. Note that it is easier to achieve higher speedups on a slower hard drive, given the larger performance gap between the drive and main memory.

It is clear that although normal prefetching (a static setting of low aggressiveness) provides a reasonable speedup for a traditional hard drive, it achieves few benefits for SSDs. While aggressive prefetching (a high static setting) helps on both devices, its cost is very low. On SSDs, aggressive prefetching loads nearly twice amount of data compared to other approaches. In contrast, normal prefetching is too conservative on SSDs, which contributes to low performance. On both devices, flashy prefetching (dynamic setting with the feedback mechanism) is able to strike a good balance between prefetching cost and speedup – it achieves over 10% performance gain while reading a modestly greater amount of data compared to the application itself.

In summary, for data prefetching, a one-size-fits-all approach cannot effectively deal with the heterogeneity and complexity that are inherent from storage devices to software applications. Simply put, without considering the architectural differences between SSDs and hard disks, data prefetching algorithms that work well on hard disks are not likely to excel on SSDs.

III. Design Principles

Designed with flash drives in mind, flashy prefetching aims to take advantage of: 1) the high I/O performance (bandwidth and throughput) and parallel I/O support that are available in solid-state drives, 2) temporal locality of the applications, and 3) the diversity of both devices and applications. Note that existing prefetching algorithms for hard drives mostly focus on application locality. We have designed flashy prefetching around three major principles.

Control prefetching based on drive performance: Since the total available throughput from a disk drive is limited and different disk drives have different latency and throughput characteristics, prefetching must be carefully managed to prevent two problems from occurring. The first is that the entire throughput to the disk may become saturated by prefetch traffic, which is very likely to happen on SSDs since the available throughput is higher for the same amount of disk

cache in main memory. Even if such traffic is entirely useful for a particular application, reads from other applications may starve because their access patterns may not be predictable. The second problem with too much prefetching is that it can evict useful data from the cache and actually hurt performance, which also can happen easily on SSDs.

Our approach to these device-specific issues is to control the amount of prefetching by periodically (with a small time period) evaluating whether and how much to prefetch and then prefetching based upon a function of an application's measured read request throughput, up to a maximum based on the total available read throughput to the disk. This means that prefetching is always done with respect to an application's measured rate instead of as fast as possible. The duration of the polling interval timer can be varied based on the latency of the underlying disk and the throughput varied in the same way.

Control prefetching based on prefetching performance: Prefetchd controls the amount of prefetching by monitoring its own performance over certain time intervals. When a performance benefit is observed, prefetchd will gradually increase the aggressiveness of the prefetching, that is, read data at a faster speed, in order to further improve the performance. This process will be reversed when prefetchd determines that aggressive prefetching hurts (or does not help) current data accesses.

Enable prefetching for multiple simultaneous accesses: The popularity of solid-state drives comes from high demand for I/O throughput from many data-intensive applications. However, supporting parallel prefetch operations has its own difficulties. Each simultaneous access pattern issued by an application must be detected individually. We achieve this goal by letting the prefetcher become aware of the program context in which accesses occur. The context includes the information about the execution environment, e.g., process id, drive id, and block id. In flashy prefetching, the process context also means how much data an application accesses at a given time, and whether a particular access pattern exists, stops, and changes. This knowledge is used to guide the intensity of data prefetching.

IV. The Architecture of Flashy Prefetching

At a high level, flashy prefetching consists of four stages: *trace collecting* that accumulates information for each application I/O request, *pattern recognition* that aims to understand the access patterns for a series of requests, *block prefetching* that moves data from the drive to the cache in the background, and *feedback monitoring* that compares previous prefetch operations against actual application requests, and adjusts the prefetching rate accordingly. Fig. 3 shows the flashy prefetching architecture.

A. Trace Collection

Flashy prefetching collects the I/O events with the help of the operating system. Typically, this information includes a timestamp, the process name and process identifier, the request type (read or write), and amount. The trace collection facility accumulates a record for every I/O request that the application asks the operating system to perform, as well as for every I/O request that actually reaches the disk and stores them for the prefetchd pattern recognizer. Not every request by the application will actually reach the disk because some of them may be satisfied by the system cache, but prefetchd traces both application requests and those that that actually reach the disk.

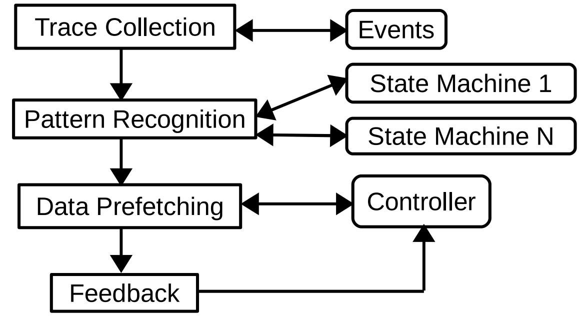


Fig. 3. Flashy Prefetching Architecture

The stored requests may come from several different applications running on multiple CPUs, and come before any I/O scheduling has occurred. A received I/O request has an associated request-type, process id, CPU number, timestamp, starting block number, and block size. The requests collected from each CPU, are sorted by time, and stored in a buffer for the later use.

B. Pattern Recognition

Internally, pattern recognition of prefetchd is designed around the idea of a *polling interval*. When a timer expires, prefetchd wakes up, looks at the accumulated I/O events, decides whether, where, and how much to prefetch, performs the prefetch request, optionally adjusts its aggressiveness based on recent prefetch performance, and sleeps for the remainder of the interval. The polling interval determines how long events accumulate in the I/O request buffer before prefetchd analyzes them. It is set once at start up and should be based on the latency of the underlying disk. If it is too small, there will not be enough accumulated events to discern a pattern. If it is too big, a pattern of accesses may already be over. This value is 0.50 seconds by default. Occasionally, large numbers of accumulated events can cause processing to take longer than the polling interval. In this case, prefetchd is careful to use the actual elapsed time since processing previously stopped to perform its calculations, but will still attempt to sleep for the same interval in the future.

A single I/O event contains several pieces of information, but prefetchd is primarily interested in the type of request (read or write), the starting block number, number of blocks in the request, and the process id of the application making the request. If a particular application makes a recognizable pattern of read accesses within a specific period of time, prefetchd begins to prefetch extrapolating the same pattern. Currently, prefetchd recognizes four major types of accesses:

sequential forward reads, sequential backward reads, strided forward reads, and strided backward reads. In this discussion a strided pattern is simply a recurring pattern with a number of blocks read and a gap where no blocks are read.

In order to perform access pattern recognition, prefetchd maintains several state machines with a front-end hash table indexed by process id, and block location on disk. The distance between subsequent block access events is compared with the previous distance. If the current request's start block is immediately where the previous request ended, the consecutive block counter is updated with the length of the current request. Similarly, if the current request's end block is immediately where the previous request started, the reverse block counter is updated. The current request may also be part of a strided pattern when the amount of jump is the same as between the previous two requests in both direction and size. In this case, the strided block counter is updated. By incrementing a counter by the request size, larger request sizes are weighted more heavily than smaller ones.

When the fraction of blocks that occurred in consecutive, reverse, or strided requests divided by the overall count of blocks read exceeds a certain threshold over the previous time interval, the state machine for that hash entry is ready to perform a prefetch during the remainder of the current time interval. *Pattern match threshold* determines which percentage of the application blocks must fit a usable pattern (sequential, reverse, or strided) before prefetchd will attempt to start prefetching. For example, the default value of 0.60 indicates that if 60 percent of the requests during a polling interval are sequential, prefetchd guesses that a sequential access is occurring and will fetch a sequential series of blocks for the next interval. When prefetchd begins prefetching on behalf of an application, it simply begins with the next block contiguous to the most recent request. The stop block is set by extrapolating into the future from the end of the last read operation.

C. Block Prefetching

The amount of data to prefetch once a pattern has been recognized is determined with the goal of reading data from an SSD into the system cache, but only those blocks that the application will actually request in the near future. For simplicity, we describe the logic for consecutive prefetching. The logic for strided and reverse prefetching is similar. In flashy prefetching, we utilize two key parameters that control how much data will be prefetched:

Application throughput scale factor is a measure of how aggressive prefetching is compared to the application's measured request rate. While we can measure the application's rate to tailor prefetching based on the application's needs, we have found that using a fixed, static scale factor does not work well.

The optimal value for this scale factor is application-specific and can be adjusted by feedback (which will be described in the next section). Our experiments showed that the values near 1.0 typically work well as the starting point for the feedback mechanism. A value of 1.0 means that prefetchd for the next polling interval, prefetchd will read exactly the amount of data it expects the application to use. Intuitively, a higher value means prefetchd will read extra data that may go to waste, and a lower value means that some portion of the application's read requests will still be expected to go to the SSD.

Maximum disk throughput: This has different optimal values for each disk. During the time interval when prefetching is occurring, prefetchd is careful to avoid saturating the available read bandwidth to the disk with prefetching requests at the expense of actual application requests that may be mispredicted and have to go to the disk. If this occurred, the requested prefetch would take more than the entire allotted time interval and prefetchd would drift further and behind real application time. To prevent this, the prefetcher estimates what amount of application requests will actually reach disk because they will not be prefetched successfully and sets the prefetching throughput limit to the maximum disk throughput minus this value. For this purpose, we use the percentage of consecutive reads that is already computed in the previous stage of pattern recognition.

Since the maximum disk throughput depends on the characteristics of each the drive, we measure the raw throughput from each disk by reading a large, uncached file, and using this as the maximum.

Putting these two parameters together, the prefetcher uses the last known (read) stop block as its start block and finds the stop block as follows. It first tries to determine the linear throughput of the application by multiplying the total throughput with the percentage of consecutive reads. We consider the remainder of the total application throughput to be from random accesses. Next, the prefetcher uses the scale factor and total available bandwidth (by subtracting application random bandwidth from the maximum disk throughput) to determine the stop block for the next polling interval.

Once the quota of number of blocks to prefetch for one application during an interval is found, prefetchd simply issues a system call (e.g., readahead in Linux) with the starting block number and the number of blocks to read. (For strided access, there may be multiple readahead calls.) In this work, we decide to leave the details of the cache management itself to the underlying operating system. Prefetchd relies on the existence of such a cache and basically fills it by reading ahead of time and hoping they remain cached. This limits the amount of information available to prefetchd and thus requires careful control over the extent of prefetching.

D. Feedback Monitoring

Feedback monitoring is at the heart of flashy prefetching. Feedback monitoring classifies the stream of read operations reaching disk as linear (meaning sequential, reverse, and strided) similar to the way read requests to the operating system were classified during pattern recognition. The logic is that if there are any linear, easily predictable reads that were not prefetched, and still reached disk, then the prefetching

Algorithm 1: The Pseudocode for Flashy Prefetching

```
begin
    for each time interval T_i do
        // Gather the list of read operations that
           reached the physical disk (i.e. not
           satisfied by cache)
        for each disk read event ReadDisk_j do
            if ReadDisk_j is issued by the prefetcher then
                // Update per-disk linear and total
                   counters
                UpdateCounters(ReadDisk_j,
                prefetcher_counters);
            end
            if ReadDisk_j is issued by applications then
                UpdateCounters(ReadDisk_j,
                app_counters);
            end
        end
        // Gather the list of read operations that are
           partially satisfied by cache
        for each requested read event ReqRead_j do
            // Supports multiple processes and
               simultaneous reads
            h = Hash(process id, starting block number);
            // Update the counters for the mapped sate
               machine
            UpdateCounters(ReqRead_j,
            hash_table[h].app_counters);
        end
        for each state machine s = hash_table[h] do
            // Calculate the percentage of consecutive
               requests
            consec_pct = s.consec_blocks /
            s.total_blocks;
            if consec_pct > consec_thres then
                // Calculate prefetch throughput
                prefetch_throughput =
                scale × consec_pct ×
                s.total_blocks/timer_interval;
                // Set a prefetch ceiling
                prefetch_throughput =
                MAX(prefetch_throughput,
                max_disk_throughput);
            end
            Prefetch(startingblock,
            PredictEndBlock(prefetch_throughput, T_i));
        end
        // Adjust prefetching aggressiveness for next
           time interval
        scale = Feedback(prefetch_cost);
    end
end
```

aggressiveness should be increased. On the other hand, if there are no linear reads reaching the disk and the statistics show that the prefetching amount is more than what the applications are requesting, we decrease the aggressiveness accordingly.

In practice, not all linear application reads can be predicted so we increase the prefetch aggressiveness scale factor when the percentage of linear reads reaching disk is greater than a predefined threshold. We decrease the aggressiveness when it is clear that additional prefetching would not help. When we see that the number of linear reads reaching disk is zero and that the number of prefetched blocks reads reaching disk is greater than the number of linear reads that the application requested to the operating system, the prefetch aggressiveness will be reduced.

During each polling interval, the feedback monitor analyzes the actual performance of the prefetch operations from the last time interval and adjusts its aggressiveness accordingly. This monitoring is done by comparing the access pattern of reads that the application makes to the operating system (entering the cache) vs. the pattern of reads reaching disk (missed in the cache). As we will explain shortly in Section V, the current prototype prefetchd does not have direct access to the kernel's page cache data structures. To address this, we use this heuristic approach to estimate the prefetch accuracy in a timely manner. Note that we have also tried the alternative of keeping a history of recently-issued prefetch operations. Although we use the history to compute the overall accuracy statistics that we report, this introduced too much latency for timely feedback.

Algorithm 1 presents the high-level pseudocode of flashy prefetching.

V. Implementation

We have implemented a prototype prefetchd in Linux systems that runs in userspace and uses the Linux page cache. In this implementation, prefetchd performs the reads from the disks and implicitly relies on the in-kernel page cache to hold the data. By running in userspace, prefetchd is completely transparent to user applications, so no recompilation, or re-linking is required. Another motivation for running in userspace is to avoid wasting physical memory for a driver-specific cache, which allows unused memory to be used for other purposes when not in use as a cache. Currently, the kernel retains control over the cache eviction policy, which requires no special handling of prefetched data. As part of future work, we plan to explore a kernel-based implementation and compare with the current OS-agnostic prototype.

A. Event Collection

Prefetchd uses the same facility as the *blktrace* [7] disk block tracing utility for Linux. Blktrace uses the Linux kernel debug filesystem to trace filesystem events. Using blktrace requires calling the BLKTRACESETUP and BLKTRACESTART ioctls for a file descriptor associated with a block device. The blktrace API offers several useful pieces of context that are not present in a traditional I/O event queue in the

driver; the events have the timestamps, process ids, and names of the originating process. Prefetchd can use this information to differentiate requests from multiple applications. Also, by examining the process id, requests from prefetchd itself can be ignored when considering applications' access patterns. Events can also be automatically filtered (read vs. write) with a mask before being delivered to prefetchd.

There is a timing disadvantage to the blktrace API, i.e., there is some lag between when I/O events are buffered in the kernel and when prefetchd reads them. Since the event buffers are maintained per-CPU, the events have to be sorted by timestamp after reading. But in practice, the event lag is almost entirely dominated by prefetchd's reaction time.

In the current implementation, a process context identifies an application execution environment by using a combination of drive id and process id, and block region. The block region is used in the hash to distinguish different threads within a single process. We plan to add file id in the future.

B. Readahead

The readahead system call in Linux [3] is designed to load the pages from a particular file into the system page cache. There is one complication with using the readahead call on a block device (as opposed to a regular file). While legal, the actual effect is to populate the system buffer cache designed for caching blocks at the device driver layer, instead of the page cache designed to cache parts of files. Our measurements indicated that although sustained read throughput from the buffer cache is 3x faster than from the SSD, sustained read throughput from the page cache is 10x faster. The current implementation uses a file spanning the entire disk with a loopback device to take advantage of the faster page cache.

Note that the userspace readahead call should not be confused with the file readahead algorithm in the Linux kernel [17] which anticipates that a sequential portion of a file will soon be needed and speculatively reads it. This is very similar to prefetching, but the amount of data loaded in this way is much smaller than the amount that the proposed prefetchd reads.

VI. Evaluation

A. Experiment Setup

High-performance storage systems are needed in many different types of data-intensive applications. To evaluate the performance of flashy prefetching technique, we choose a wide variety of benchmarks, including database applications, web servers, file servers, and scientific computing.

Database test suite (DBT3) is an open source database benchmark [2] that implements the TPC-H benchmark. DB is a decision support benchmark with business oriented adhoc queries. We create and populate the database in Postgres and evaluate a subset of 22 queries. We remove some queries because they take a too long or too little time to run.

BLAST (Basic Local Alignment Search Tool) [1] is a widely used algorithm for identifying local similarity between different biological sequences. We pick the NIH implementation for searching nucleotide queries in nucleotide database. The input database is obtained from NCBI and has 12 GB of non-redundant DNA sequences.

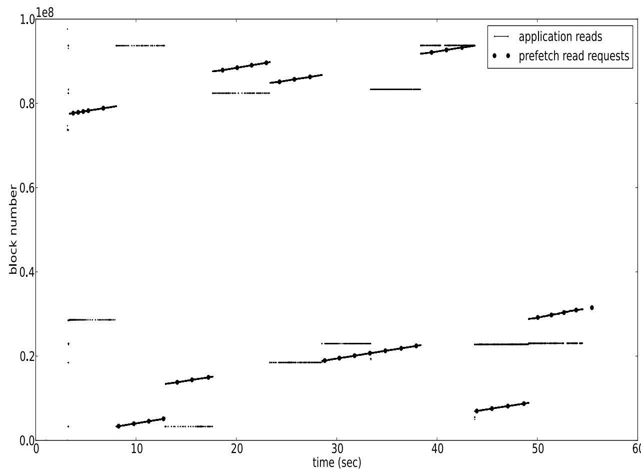
LFS, Sprite large file benchmark [38], performs both reads and writes on a large file, as well random and sequential read of the file. We use a file size of 100 GB.

PostMark is a widely used benchmark for email server type of workloads, which unlike LFS deals with a large number of small files [28].

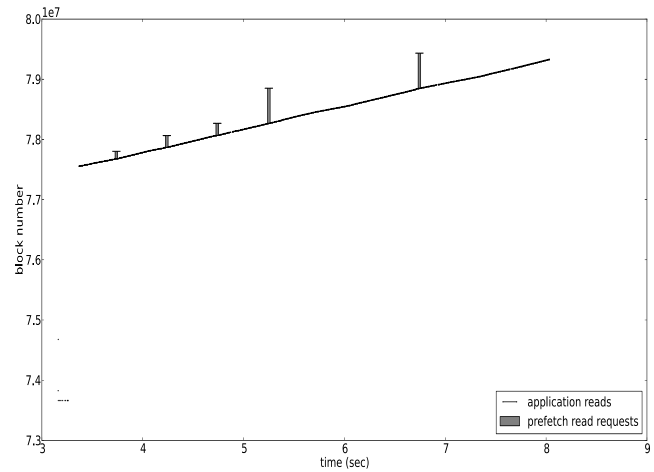
WebSearch [45] contains two block-level I/O traces collected from a web search engine. These traces come in SPC format which is a text file containing a timestamp, offset in disk, operation size, type of operation, and thread id. In order to play the traces and also have a test bed for re-running application traces, we developed a trace replayer that can play back a series of read operations at a desired speed and with a desired number of worker processes, typically at six times normal speed and one worker thread. We report the total I/O wait time as the performance metric. Note that there is some difficulty here when using total elapsed time as a metric when using replayed traces. The original captured SPC timestamps include time spent waiting for I/O to complete as well as idle. If a trace is just replayed and prefetching improves I/O performance, the replayer will spend less time waiting and more time idle – but the total elapsed time will still be the same. To avoid this problem, we consider the total time spent waiting for I/O operations to complete when running these benchmarks and measure speedup using these times.

SlowThink is a synthetic benchmark that we developed to simulate a CPU-intensive application. This application reads from a file in 1 MB records and performs several iterations of difference computations on the read bytes. By varying the number of iterations performed, we can control the applications run time and I/O throughput. We can vary the interval time, application throughput scale factor, and examine the behavior of prefetchd. This benchmark helps guide us to find the specific cases where the prefetching can be helpful for SSDs.

The test system has Linux kernel 2.6.28 with an Intel Core2 Quad CPU at 2.33 GHz and 8 GB RAM. We tested four SSDs and one hard drive, as listed in Table I. Each storage device is formatted with an ext2 filesystem, mounted with the *noatime* option and filled with one large file which was connected to a loopback device. The loopback device is then formatted with an ext3 filesystem and also mounted with the *noatime* option for running the benchmarks. The *noatime* option prevents read operations to the filesystem from generating metadata updates which would require writes to the device and is intended to improve the I/O throughput. The drives used for testing (especially the hard drive) are not especially fast. We plan on evaluating our approach with more diverse and faster storage devices in the future.



(a) Zoomed-out view



(b) Zoomed-in view

Fig. 4. Zoomed-out and -in views of block traces. The X-axis represents time in seconds and the Y-axis represents the 512-byte block number on SSD. Lines in the figures represent real data access, and dots and arrows represent data prefetching.

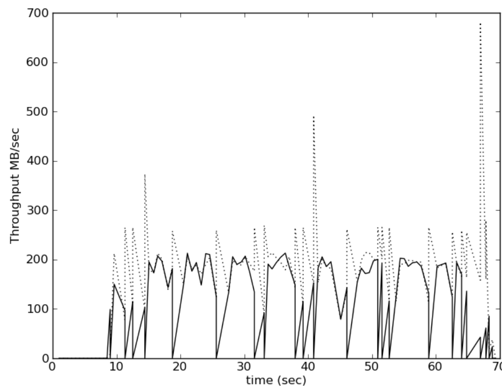


Fig. 5. Prefetchd aggressiveness on the SSD3. The dotted line represents I/O read operations per second over time from the benchmark and the solid line represents prefetchd throughput, i.e., aggressiveness over time.

B. Flashy Prefetching at Work

Fig. 4 presents the zoomed-out and -in views of a data region from the BLAST benchmark. The lines are formed from a scatter plot of the block id numbers read by the application as a function of time in seconds. The Y-axis is the block id number based on 512-byte blocks. Since most of these reads are sequential, the reads form gently sloping lines. The actions of prefetch operations in response to application reads are shown as the dots in the zoomed-out view of Fig. 4(a), and as the arrows in the zoomed-in view of Fig. 4(b). The horizontal position of an arrow indicates the time a prefetch operation is requested and its vertical extent shows the amount of data that is prefetched.

Clearly, the application does not read the data entire sequentially on the device – it goes through different stages that consist of sequential reads, seeks, random reads, etc. In addition to the gaps that exist between data accesses, the varying slopes show that the throughput available from the device and obtained by the application is not entirely constant.

Data prefetching, presented by upwards arrows in the Figure, shows that the prefetching occurs just before those blocks are accessed by the application, except for the gaps where prefetchd mispredicts the next blocks. The changing sizes of the arrows indicate that prefetchd adapts the speed of data prefetching in runtime to match the needs of the application.

We also measure the aggressiveness of the prefetchd against the performance of the real application. Fig. 5 presents the numbers collected from running BLAST. It is clear that prefetchd is able to follow the application trend closely and adjust its aggressiveness accordingly.

C. Performance Speedup

We evaluate prefetchd by running all four benchmarks. As shown in Fig. 6, flashy prefetching performs well on all the benchmarks – prefetchd achieves average 31%, 22%, 10%, and 28% speedup on the hard drive, solid-state drive, and two SSD RAIDs, respectively. Speedup was measured by dividing the run time without prefetchd by the run time with prefetchd. In the base case without prefetchd, the default I/O configuration for the Linux system is used, including some limited readahead by the kernel, normal file caching in the page cache, use of the buffer cache, etc. Note that while all benchmarks already run much faster on solid-state drives, prefetchd is still able to achieve a significant amount of improvements of 20% on average. Prefetchd provides the best performance speedups on the LFS benchmark, that is, 3.44, 2.9, 1.09, and 1.97 times on four tested devices. Note that the speedup over two times is not shown in the figure. For the database benchmarks, prefetchd delivers on average 9%, 13%, and 15% improvements on the single SSD, and two SSD RAIDs. For the hard drive, some database scripts result in small performance slowdowns, indicating the need for less aggressive prefetching.

In addition, we evaluate prefetchd by running a number of DB scripts on all four SSDs. The results are presented in Fig. 7, where most scripts can expect a speedup of up to 6% on average on four SSDs. In some cases, e.g., DB script

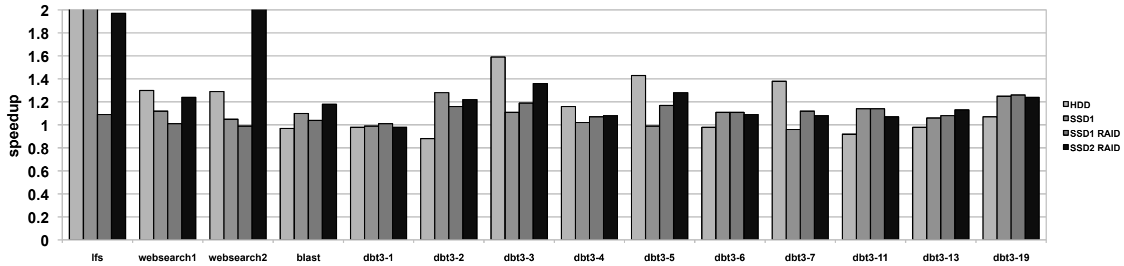


Fig. 6. Prefetchd performance using flashy prefetching for different benchmarks and devices. Benchmark speedup is on the y-axis. Values above 2.0 are omitted (including for websearch2 at 2.02) to more clearly show variation between the other benchmarks.

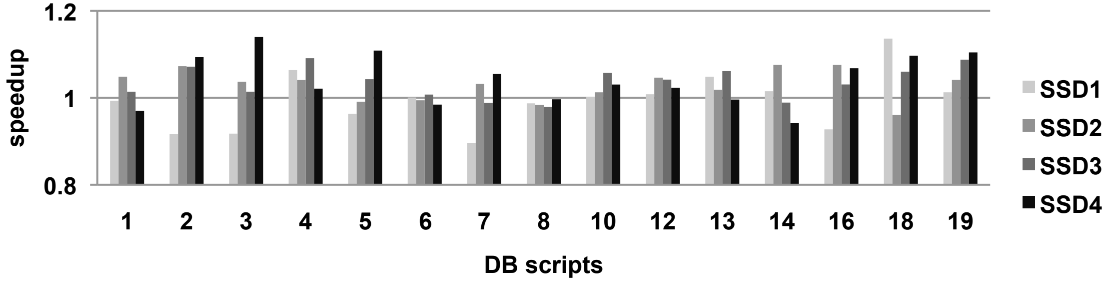


Fig. 7. Prefetchd performance using flashy prefetching for different SSDs

19 on SSD4, prefetchd achieves a more than 10% speedup. Prefetchd does not always provide good performance benefits, e.g., when running the query script 6, it experiences small slowdowns on some devices. We think that there are two reasons for this. First, some SSDs have difficulty dealing with the many writes in some of the database queries. Second, this partly confirms previous belief [14], [40] that because SSDs have good random access performance, the help from data prefetching may be limited. However, we believe that the feedback monitoring component in our flashy prefetching can be enhanced to minimize this effect, which we will explore as part of future work.

Also note that in many instances the speedups achieved on the HDD exceed those on an SSD, especially for benchmarks performing large, contiguous reads such as LFS. It is harder to achieve the same speedup on an already-fast device since a smaller fraction of the workload is spent in I/O. On a real system, the situation is more complicated due to multiple simultaneous reads and non-linear access patterns, but the basic idea described above limits the speedup of simple benchmarks on fast devices.

D. Prefetching Accuracy

In this section, we evaluate the prediction accuracy of our prefetching algorithm. The accuracy is calculated by dividing the amount of prefetched and subsequently used data by the total used data. The word *used* here means read by the application. Fig. 8 presents the accuracy for different benchmarks on various devices. On average, prefetchd achieves more than 60% accuracy for all the benchmarks. Prefetchd achieves over 70% accuracy for most database benchmarks. The average accuracy for database benchmarks is 68% for the hard drive, and about 72% for SSD and SSD RAIDs. The only exception is the two WebSearch benchmarks, which we suspect is caused by the existence of the large amount of random accesses. Although the prediction has low accuracy for the WebSearch traces, prefetchd provides a good 25% average improvement on four devices. If not counting the WebSearch benchmarks, our proposed flashy prefetching predicts with about 70% accuracy.

E. Prefetching Cost

We further examine prefetchd's *cost* that is defined as the ratio of the amount of prefetched data (true and false positives) to the amount of data read by the application. A lower cost indicates less data preloaded by the prefetchd. On average, prefetchd reads 77% more data than the benchmarks, with 60% for the single SSD and an average of 90% for the two RAIDs. The fastest device of four, SSD2 RAID tends to read more data and have a lower cost. Fig. 9 presents the prefetching costs on all four devices. In a few cases like LFS and DBT3-1, prefetching may incur cost as high as four times depending on the type of the device.

VII. Related Work

There has been a rich set of prior research on data prefetching on hard disks, which we cannot possibly enumerate. Some representative techniques include probability graph [20], data compression [15], data mining [31], semantics-aware [11], [43], address tracking [18], [19], compiler support [33], [10], off-line information [26], [27], and hints [12], [37]. Data prefetching can also be done at both block level (e.g., [31], [16]) and file level (e.g., [42], [30], [50], [47]), and has been closely studied with caching [48], [50], [19], [8], [51]. In

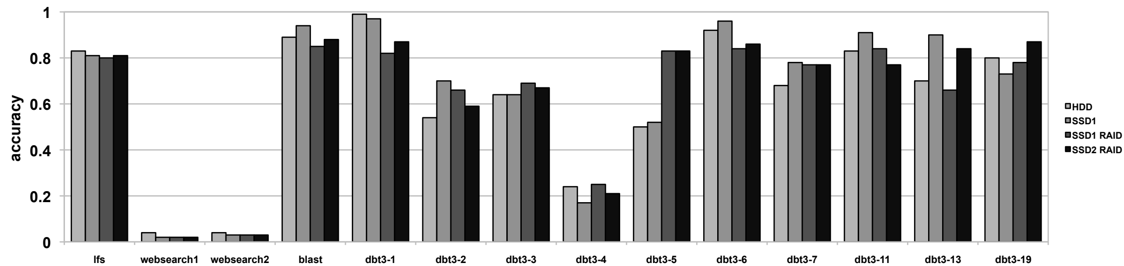


Fig. 8. Prefetchd accuracy using flashy prefetching for different benchmarks and devices. Benchmark accuracy is on the y-axis, measured as the amount of prefetched and used data divided by total used data.

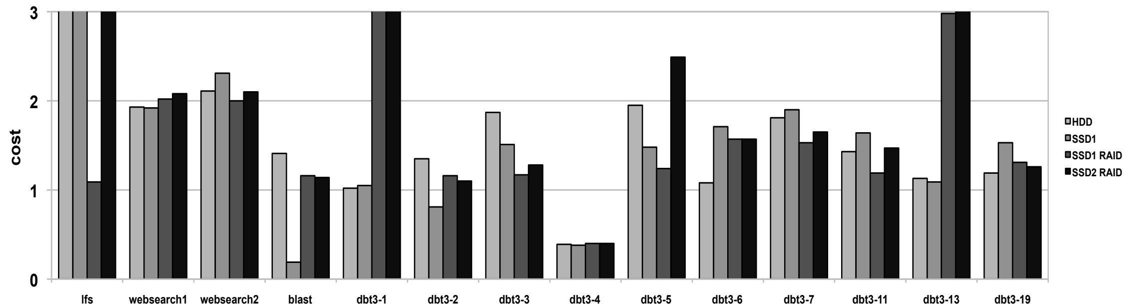


Fig. 9. Prefetchd cost using flashy prefetching for different benchmarks and devices. Prefetchd cost is on the y-axis, defined as the ratio of the amount of prefetched data (true and false positives) and the amount of data read by the application. Values above 3.0 are omitted to more clearly show variation between the other benchmarks.

addition, prefetching techniques are common for fetching data from main memory on high-performance processors into processor caches where similar challenges about I/O bandwidth and pollution apply, notably [44], [21].

Our proposed flashy prefetching is orthogonal to techniques previously applied to hard disks in the sense that we focus on emerging flash-based solid-state drives and SSD based RAIDs whose high throughput provides new opportunities and challenges for data prefetching. In particular, sharing the self-monitoring and self-adapting approach as in [41], we work on the adaptation of prefetching aggressiveness in runtime to meet the needs from applications and stress SSDs within a reasonable range. In essence, our technique is also similar to freeblock scheduling [32] that utilizes free background I/O bandwidth in a hard drive. We believe that our technique can be potentially combined with a few existing prefetching techniques, e.g., [12], [50], [47].

Note that SSD devices are performing data prefetching on a small scale by utilizing parallel I/Os and an internal memory buffer. Work has been started to measure and understand this effect [13], [29], [22]. In comparison, our proposed prefetching is designed and implemented in the software layer, which can be used to complement the hardware-based approach.

Current operating systems do not have a good support for data prefetching on solid-state drives. For example, Windows 7 recommends computer systems with SSDs not use features such as Superfetch, ReadyBoost, boot prefetching, and application launch prefetching, and by default turns them off for most SSDs [4]. The key reason is that such features were designed with traditional hard drives in mind. It has been shown that enabling them provides little performance benefit [5]. Linux developers also realize the need to have a tunable I/O size as well as the need for more aggressive prefetching [49]. Development efforts on improving prefetching performance on SSDs are ongoing, and we believe that our findings will be beneficial in this area.

FAST is a recent work that focuses on shortening the application launch time and utilizes prefetching on SSDs for quick start of various applications [25]. It takes advantage of the nearly identical block-level accesses from run to run and the tendency of these reads to be interspersed with CPU computations. This approach even uses the blktrace API with an LBA-to-inode mapper instead of using a loopback device like prefetchd. A similar work to FAST is C-Miner [31], which discovers block correlations to predict which blocks will be accessed. This approach can cope with a wider variety of access patterns while prefetchd is limited to simpler strided forward and backward patterns. Our approach differs from these two in that it can handle request streams from multiple simultaneous applications and includes an aggressiveness-adjusting feedback mechanism. We believe that incorporating block correlations would improve prefetchd's accuracy in some cases and plan to investigate this approach in the future.

We would also like to point out that some researchers have expressed reservations against data prefetching on solid-state drives. IotaFS chooses not to implement prefetching among the file system optimizations it used for SSDs [14]. In addition, FlashVM [40] found out that disabling prefetching can be beneficial to some benchmarks. As we have discussed

before, prefetchd is not always helpful – for some benchmarks, prefetchd has limited benefits and may even lead to some modest regressions, which we plan to further investigate in the future.

VIII. Conclusions

We have designed and implemented a data prefetcher for emerging high-performance storage devices, including flash-based solid-state drives that detects application access patterns, retrieves data to match both drive characteristics and application needs, and dynamically controls its aggressiveness with feedback. Currently, the prefetcher works well for a number of I/O intensive applications. We implement a prototype in Linux and conduct a comprehensive evaluation on hard drive, SSDs, as well as SSD RAIDs, with a wide range of data-intensive applications and benchmarks, where the prototype is able to achieve a 20% speedup and a 65-70% prefetching accuracy on average.

IX. Acknowledgments

The authors are grateful to the anonymous reviewers for their feedback and suggestions. This work is in part supported by the National Science Foundation under grants OCI-0937875 and IOS-1124813.

References

[1] "Basic local alignment search tool," www.ncbi.nlm.nih.gov/BLAST/.
[2] "Database test suite," http://osdldbt.sourceforge.net/.
[3] Linux man page for readahead system call. [Online]. Available: http://www.kernel.org/doc/man-pages/online/pages/man2/readahead.2.html
[4] MSDN blogs. engineering windows 7.support and q&a for solid-state drives. [Online]. Available: http://blogs.msdn.com/b/e7/archive/2009/05/05/support-and-q-a-for-solid-state-drives-and.aspx
[5] Super-fast ssds: Four rules for how to treat them right. [Online]. Available: http://itexpertvoice.com/home/super-fast-ssds-four-rules-for-how-to-treat-them-right/
[6] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX Annual Technical Conference*, 2008, pp. 57–70.
[7] J. Axboe and A. D. Brunelle. (2007) blktrace user guide.
[8] S. H. Baek and K. H. Park, "Prefetching with adaptive cache culling for striped disk arrays," in *USENIX Annual Technical Conference*, 2008, pp. 363–376.
[9] A. Beckmann, U. Meyer, P. Sanders, and J. Singler, "Energy-efficient sorting using solid state disks," in *2010 International Green Computing Conference*. IEEE, 2010, pp. 191–202.
[10] A. D. Brown, T. C. Mowry, and O. Krieger, "Compiler-based I/O prefetching for out-of-core applications," *ACM Trans. Comput. Syst.*, vol. 19, no. 2, pp. 111–170, 2001.
[11] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Exploiting gray-box knowledge of buffer-cache management," in *USENIX Annual Technical Conference*, 2002, pp. 29–44.
[12] F. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in *Proceedings of the third symposium on Operating Systems Design and Implementation*. USENIX Association, 1999, pp. 1–14.
[13] F. Chen, D. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the eleventh international joint conference on measurement and modeling of computer systems*, 2009, pp. 181–192.
[14] H. Cook, J. Ellithorpe, L. Keys, and A. Waterman. Iotafs: Exploring file system optimizations for ssds. [Online]. Available: http://www.stanford.edu/~jdellit/default_files/iotafs.pdf
[15] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical prefetching via data compression," in *Proceedings of the ACM SIGMOD international conference on management of data*, 1993, pp. 257–266.
[16] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "Diskseen: exploiting disk layout and access history to enhance i/o prefetch," in *USENIX Annual Technical Conference*, 2007, pp. 20:1–20:14.
[17] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the design of a new linux readahead framework," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 75–84, July 2008.
[18] B. S. Gill and D. S. Modha, "SARC: sequential prefetching in adaptive replacement cache," in *USENIX Annual Technical Conference*, 2005.
[19] B. S. Gill and L. A. D. Bathen, "AMP: adaptive multi-stream prefetching in a shared cache," in *Proceedings of the 5th USENIX conference on File and Storage Technologies*, 2007.
[20] J. Griffioen, "Performance measurements of automatic prefetching," *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pp. 165—170, 1995.
[21] Y. Guo, P. Narayanan, M. A. Bennaser, S. Chheda, and C. A. Moritz, "Energy-efficient hardware data prefetching," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 19, no. 2, pp. 250–263, Feb. 2011.
[22] H. Huang, S. Li, A. Szalay, and A. Terzis, "Performance modeling and analysis of flash-based storage devices," in *IEEE 27th Symposium on Mass Storage Systems and Technologies*, 2011, pp. 1–11.
[23] Intel, "Intel X-25M SSD Specification," http://download.intel.com/design/flash/nand/mainstream/mainstream-sata-ssd-datasheet.pdf, 2009.
[24] ——, "Intel 510 SSD Specification," http://download.intel.com/pressroom/kits/ssd/pdf/Intel_SSD_510_Series_Product_Specification.pdf, 2011.
[25] Y. Joo, J. Ryu, S. Park, and K. Shin, "FAST: quick application launch on solid-state drives," in *Proceedings of the 9th USENIX conference on File and Storage Technologies*, 2011, pp. 19–19.
[26] M. Kallahalla and P. J. Varman, "Optimal prefetching and caching for parallel i/o sytems," in *Proceedings of the thirteenth annual ACM symposium on parallel algorithms and architectures*, 2001, pp. 219–228.
[27] ——, "Pc-opt: Optimal offline prefetching and caching for parallel i/o systems," *IEEE Trans. Comput.*, vol. 51, no. 11, pp. 1333–1344, Nov. 2002.
[28] J. Katcher, "Postmark: a new file system benchmark." Network Appliance Tech Report TR3022, Oct. 1997.
[29] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh, "Parameter-aware i/o management for solid state disks (SSDs)," *IEEE Transactions on Computers*, vol. 99, 2011.
[30] T. M. Kroeger and D. D. E. Long, "Design and implementation of a predictive file prefetching algorithm," in *USENIX Annual Technical Conference*, 2001, pp. 105–118.
[31] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-Miner: mining block correlations in storage systems," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004, pp. 173–186.
[32] C. R. Lumb, J. Schindler, and G. R. Ganger, "Freeblock scheduling outside of disk firmware," in *Proceedings of the Conference on File and Storage Technologies*, 2002, pp. 275–288.
[33] T. C. Mowry, A. K. Demke, and O. Krieger, "Automatic compiler-inserted i/o prefetching for out-of-core applications," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. SI, pp. 3–17, Oct. 1996.
[34] OCZ, "OCZ Vertex SSD Specification," http://www.ocztechnology.com/products/flash_drives/ocz_vertex_series_sata_ii_2_5-ssd, 2009.
[35] ——, "OCZ Vertex 2 SSD Specification," http://www.ocztechnology.com/ocz-vertex-2-sata-ii-2-5-ssd.html, 2011.
[36] A. E. Papathanasiou and M. L. Scott, "Aggressive prefetching: an idea whose time has come," in *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*, 2005, pp. 6–11.
[37] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 79–95, 1995.
[38] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.
[39] Samsung, "Spinpoint m7 hard disk specification," http://www.samsung.com/global/system/business/hdd/prdmodel/2009/1/13/728799m7_sheet_0.5.pdf, 2009.
[40] M. Saxena and M. M. Swift, "FlashVM: revisiting the virtual memory hierarchy," in *Proceedings of the 12th conference on Hot Topics in Operating Systems*, 2009, pp. 13–13.
[41] M. Seltzer and C. Small, "Self-monitoring and self-adapting operating systems," in *The Sixth Workshop on Hot Topics in Operating Systems*, May 1997, pp. 124 –129.
[42] E. Shriver, C. Small, and K. A. Smith, "Why does file system prefetching work?" in *USENIX Annual Technical Conference*, 1999, pp. 6–35.

[43] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Semantically-smart disk systems," in *Proceedings of the 2nd USENIX conference on File and Storage Technologies*, 2003, pp. 6–22.
[44] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 63–74.
[45] UMass, "Umass trace repository," http://traces.cs.umass.edu, 2007.
[46] L. Useche, J. Guerra, M. Bhadkamkar, M. Alarcon, and R. Rangaswami, "Exces: External caching in energy saving storage systems," in *IEEE 14th International Symposium on High Performance Computer Architecture*, Feb. 2008, pp. 89–100.
[47] G. Whittle, J.-F. Pâris, A. Amer, D. Long, and R. Burns, "Using multiple predictors to improve the accuracy of file access predictions," in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, Apr. 2003, pp. 230–240.
[48] T. M. Wong and J. Wilkes, "My cache or yours? making storage more exclusive," in *USENIX Annual Technical Conference*, 2002, pp. 161–175.
[49] F. Wu, "Sequential File Prefetching in Linux," *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*, p. 218, 2010.
[50] C. Yang, T. Mitra, and T. Chiueh, "A decoupled architecture for application-specific file prefetching," in *USENIX Annual Technical Conference, FREENIX Track*, 2002, pp. 157–170.
[51] Z. Zhang, A. Kulkarni, X. Ma, and Y. Zhou, "Memory resource allocation for file system prefetching: from a supply chain management perspective," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 75–88.