Elastic Prefetching for
High-Performance Storage Devices

by Ahsen Uppal

A Thesis submitted to
The Faculty of
The School of Engineering and Applied Science
of the George Washington University in partial satisfaction
of the requirements for the degree of Master of Science

August 31, 2011

Thesis Advisor:

H. Howie Huang
Assistant Professor of Engineering and Applied Science
Acknowledgement

I would like to thank the members of the defense committee: Alex M. Li, Guru Prasad Venkataramani, and H. Howie Huang. I would like to thank my family for their love and support.
Abstract

Elastic Prefetching for High-Performance Heterogeneous Storage Devices

The spectrum of storage devices has expanded dramatically in the last several years with the increasing popularity of NAND flash memory. While hard drives hold on to the capacity advantage, flash-based solid-state drives (SSD) with high IOPS and low latencies have become good candidates for data-intensive applications. As scientific and enterprise data requirements continue to grow rapidly, high-performance storage systems will consistently be in high demand. Although commonly used to improve the I/O performance of data-intensive applications, data prefetching, if inappropriately controlled, is likely interfere with normal I/O requests and result in lower application performance. In this work, we demonstrate that good performance benefits from data prefetching can be achieved with the help of accurate prediction and an adaptive feedback-directed prefetching rate that scales with application needs and is also sensitive to varying storage device architectures. We call this combined approach elastic prefetching.

We have designed prefetchd, an elastic data prefetcher, that understands the architectural characteristics of heterogeneous storage devices and carefully prefetches data in a manner that closely matches application needs in runtime. We have implemented a Linux-based prototype that runs in user space, monitors application read requests, predicts which pages are likely to be read in the near future, and issues readahead system calls to load those pages into the system page cache, monitors its performance in time and space, and adjusts its operating parameters based on the results. We have evaluated the prototype on different SSDs, as well as SSD RAIDs, with a wide range of data-intensive applications and benchmarks. The prototype
achieves 65-70% prefetching accuracy and delivers average 20% speedups on replayed web search engine traces, BLAST, and TPC-H like benchmarks across various storage drives.
# Contents

Acknowledgement ii

Abstract iii

Contents v

List of Figures vi

List of Tables viii

1 Introduction 1

2 Background 4

   2.1 Flash Based Solid-State Drives . . . . . . . . . . . . . . . . . . 4
   2.2 Beyond Hard Disk Based Prefetching . . . . . . . . . . . . . . . 6
   2.3 New Requirements . . . . . . . . . . . . . . . . . . . . . . . . 8

3 Elastic Prefetching 11

   3.1 Trace Collection . . . . . . . . . . . . . . . . . . . . . . . . 11
   3.2 Pattern Recognition . . . . . . . . . . . . . . . . . . . . . . 13
   3.3 Block Prefetching . . . . . . . . . . . . . . . . . . . . . . . 14
   3.4 Feedback Monitoring . . . . . . . . . . . . . . . . . . . . . . 16
   3.4.1 Spatial Feedback . . . . . . . . . . . . . . . . . . . . . . 16
List of Figures

2.1 Device and Application Complexity ............................... 5
2.2 The Need for Elastic Prefetching ................................. 10
3.1 Prefetchd Architecture ........................................... 12
5.1 Zoomed-out view of block traces for a dbt3-13 query on the SSD1 and elastic prefetch operations. Blue dots in the figures represent real data access, and red arrows represent data prefetching. The x-axis represents time in seconds and the y-axis represents the 512-byte block number on disk. .................................................. 25
5.2 Zoomed-in view of block traces for a BLAST-N benchmark on the OCZ RAID and elastic prefetch operations. Blue dots in the figures represent real data access, and red arrows represent data prefetching. The x-axis represents time in seconds and the y-axis represents the 512-byte block number on disk. .................................................. 26
5.3 Prefetchd aggressiveness in runtime for a BLAST-N benchmark on the SSD1 RAID and elastic prefetch operations. The solid blue line represents I/O read operations per second over time and the dashed red line represents prefetchd scale factor i.e. aggressiveness over time. 27
LIST OF FIGURES

5.4 Prefetchd performance using elastic prefetching for different benchmarks and devices. Benchmark speedup is on the y-axis. The device models corresponding to each name are described in Table 2.1. 28

5.5 Prefetchd accuracy using elastic 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. The device models corresponding to each name are described in Table 2.1. 28

5.6 Prefetchd efficiency using elastic prefetching for different benchmarks and devices. Benchmark efficiency 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. The device models corresponding to each name are described in Table 2.1. 29

5.7 Performance of prefetchd for one and two concurrent threads for the Websearch-1 benchmark on the SSD1 RAID. The y-axis represents the measured speedup. 33

5.8 Performance of prefetchd under different replay speeds of the Websearch-1 benchmark on the SSD1 RAID with a single concurrent thread. The low, medium, and high descriptions correspond to speeds of 6x, 12x, and 18x the original Websearch-1 trace speed. The y-axis represents the measured speedup. 34
## List of Tables

2.1 Storage Device Specifications ................................. 6

3.1 Summary of prefetchd parameters .............................. 19
Chapter 1

Introduction

To provide high-performance data analysis, data-intensive applications need fast access to a vast amount of data that are stored on external storage devices. As NAND flash memory based Solid-State Drives (SSDs) provide excellent I/O throughput and energy efficiency [1][2], the spectrum of storage devices has expanded drastically in the last several years and SSDs have become commonly used for data-intensive applications. As scientific and enterprise data continue to grow exponentially, high-performance storage systems that leverage both high throughput from SSDs and high capacity from hard drives will likely be in high demand to reduce the I/O performance gap.

Data prefetching [3, 4] is one of the most widely used techniques to reduce access latency, by loading the data that are likely to soon be accessed, from the storage devices into main memory. Traditional prefetching techniques have been focused on rotational hard drives and are conservative on the amount of data prefetched – they often leverage the low cost of sequential access on hard drives to read the data that on the same and nearby tracks. Because data prefetching consumes shared system resources (e.g., I/O bandwidth, system processing, and main memory), it is likely that aggressive data prefetching would interfere with the normal access and
CHAPTER 1. INTRODUCTION

subsequently hinder application performance. As a result, aggressive prefetching has been considered too risky (given long seek penalties, limited bandwidth on hard drives, and limited system RAM) until recently [5].

For high-performance hard drives and SSDs, aggressive prefetching could potentially expedite data requests of applications to a large degree. However, as we will demonstrate, simply prefetching as much data as possible does not provide the desirable benefits for three main reasons. First, data prefetching on faster devices such as SSDs, if uncontrolled, will take the shared I/O bandwidth from existing data accesses (more easily than slower hard drives). As a side effect, the main memory would be filled with mispredicted (and unneeded) data while the applications are 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. The performance of a magnetic hard drive varies too, although it can be roughly approximated by the rotation speed. In this case, a prefetching algorithm, while reasonably aggressive for a faster drive, could potentially become too aggressive for another drive, again 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 too conservative and too aggressive prefetching.

In this work, we believe that for emerging high-performance storage devices, a smart prefetching technique should be aware of runtime environment and adapt to the changing requirements from both the devices and applications, which requires making good tradeoffs between data prefetching and resource consumption. To this end, we propose the technique of elastic prefetching and implement a prototype called prefetchd that takes into consideration both application requirements and storage device characteristics, and dynamically controls the prefetching aggressiveness at run-
CHAPTER 1. INTRODUCTION

time to maximize the performance benefits. Prefetchd 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 the success rate in time and across pages, and adjusts its aggressiveness accordingly.

We evaluate prefetchd on hard drives, SSDs, as well as SSD RAIDs, with a wide range of data-intensive applications and benchmarks. The prototype achieves 20% speedups on replayed Websearch 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 twofold:

- We conduct a comprehensive study on the effects of conservative and aggressive prefetching in the context of heterogeneous devices and applications. The results show that elastic prefetching is essential to take advantage of high-performance storage devices, e.g., solid-state drives and RAIDs.

- We design and develop a prototype, prefetchd, that self-tunes to prefetch data in a speed that matches the 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.

The rest of the paper is organized as follows. Chapter 2 describes the need for controlled aggressive prefetching. Chapter 3 presents the architecture of prefetchd and describes each individual components. Chapter 4 discusses the implementation in detail. The evaluation is presented in Chapter 5 and related work is discussed in Chapter 6. We conclude in Chapter 7.
Chapter 2

Background

2.1 Flash Based Solid-State Drives

Today most solid-state drives are built upon non-volatile NAND flash memory that consists of several components such as flash packages, controllers, and buffers. A read to flash can be completed quickly in a few microseconds, comparing to a several millisecond seek latencies on hard drives, which contributes mostly to large improvements on I/O bandwidth and throughput. SSDs do not incur the same seek penalty as hard drives where a mechanical head positioned above a platter must physically move. 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, unlike on a hard disk. The internal controllers of SSDs have already taken advantage of this inherent parallelism for high performance I/O [1], and in this work we will show that this parallelism can also be exploited from higher-level system levels.

But flash has its own drawbacks. Flash writes are slower in hundreds of microseconds, and block-level erase operations are needed before next updates. Furthermore, block erases are expensive at several milliseconds, and each cell has limited (100,000
CHAPTER 2. BACKGROUND

In this study, we use two high-performance SSDs, Intel X-25M (SSD1) [6] and OCZ Vertex (SSD2) [7], as well as a Samsung Spinpoint M7 (HDD) hard drive [8]. In addition, we evaluate a level-0 RAID that consists of two identical devices, namely SSD1 RAID, SSD2 RAID, and HDD RAID. We use software RAID for SSDs, and to 1 million) erase cycles before it wears out.
CHAPTER 2. BACKGROUND

hardware (BIOS) RAID for hard drives. Table 2.1 presents the specifications for three devices. As shown in Fig. 2.1(a), when measured under Linux, two SSDs clearly have higher bandwidth than the hard drive, that is, SSD1 and SSD2 outperform HDD by 160% and 50%, respectively. Note that two SSDs differ noticeably – their measured bandwidth is 156 and 262 MB/s.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>500GB</td>
<td>80GB</td>
<td>128GB</td>
</tr>
<tr>
<td>Buffer Size</td>
<td>8MB</td>
<td>Unknown</td>
<td>64MB</td>
</tr>
<tr>
<td>Flash Type</td>
<td>N/A</td>
<td>MLC</td>
<td>MLC</td>
</tr>
<tr>
<td>Rotational Speed</td>
<td>5,400 RPM</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Read Bandwidth</td>
<td>-</td>
<td>250MB/s (seq)</td>
<td>250 MB/s</td>
</tr>
<tr>
<td>Write Bandwidth</td>
<td>-</td>
<td>70MB/s (seq)</td>
<td>160MB/s</td>
</tr>
<tr>
<td>Latency</td>
<td>5.6ms (avg)</td>
<td>85us (Read)</td>
<td>0.4ms (avg)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>115us (Write)</td>
<td></td>
</tr>
<tr>
<td>Active Power</td>
<td>2.5W</td>
<td>0.15W</td>
<td>2W</td>
</tr>
<tr>
<td>Idle Power</td>
<td>0.85W</td>
<td>0.06W</td>
<td>0.5W</td>
</tr>
</tbody>
</table>

2.2 Beyond Hard Disk Based Prefetching

Although data-intensive applications are in dire need of high-performance data access, they tend to have different I/O requirements. Fig. 2.1(b) presents the average application throughput in IOPS for 14 applications. We will describe these benchmarks in detail in Section 5. 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.

For data prefetching, an approach of one-size-fits-all cannot effectively deal with
CHAPTER 2. BACKGROUND

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 continue to excel on SSDs.

Traditional disk drives can read sequential blocks quickly because the head can be stationary while the platter rotates underneath. 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 problem still limits the number of simultaneous prefetch operations that can occur at once on a traditional hard drive. In contrast, parallel I/Os in SSDs can benefit greatly from good support of 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 of going beyond traditional prefetching, we present the performance results in Fig. 2.2 from three different prefetching techniques, normal, aggressive, and the proposed elastic prefetching. Here we run a database benchmark (dbt3-3) on three devices, including the hard drive, an SSD, and SSD RAID. Speedup is measured using elapsed wall-clock time and efficiency is defined as the ratio of the amount of prefetched data and that of data read by the application. The details of our evaluation environment can be found in Chapter 5. It is clear that although normal prefetching provides a reasonable speedup for a traditional hard drive, it achieves few benefits for SSDs. While aggressive prefetching helps on three devices, its efficiency defined by the ratio of data read by prefetch and the application is very low. On high performance SSDs, aggressive prefetching loads nearly twice amount of data compared to other approaches. In contrast, normal prefetching is very conservative
CHAPTER 2. BACKGROUND

on SSDs, which contributes to low performance. On all three devices, the proposed elastic prefetching is able to strike a good balance between prefetching efficiency and speedup – it achieves 20 to 36% performance gain while reading a modest amount of data comparable to the application itself.

2.3 New Requirements

Designed with emerging high-performance devices in mind, prefetchd aims to take advantage of: 1) the high I/O performance (bandwidth and throughput) that are available in solid-state drives, 2) the spatial and temporal locality of the applications, and 3) the diversity of both devices and applications. Note that existing prefetching algorithms mostly focus on the application locality, ignoring the characteristics of heterogeneous devices. We believe that an elastic prefetching algorithm should possess the following capabilities:

**Control the amount of prefetching based on drive performance.** A major issue is that the total available throughput from a disk drive is limited and different disk drives have different latency and throughput characteristics. This applies to both hard disks and solid-state drives. Thus, 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. 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.

Our approach to these issues is to control the amount of prefetching by periodically evaluating whether and how much to prefetch with a small time period and then prefetching based upon a function of an application’s measured read request throughput. 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 the amount of prefetching based on prefetching performance.** Prefetchd controls the amount of prefetching by monitoring its own performance over certain time intervals. When performance speedup is observed, prefetchd will gradually increase the aggressiveness of the prefetching, that is, read more data in 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.

**Detect process context 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 concurrent prefetch operations has its own difficulties. Each simultaneous access pattern issued by an application must be detected individually. Prefetchd achieves this goal by becoming aware of the program context in which accesses occur. The context includes the information on the execution environment, e.g., process id, drive id, and block id. In prefetchd, the process context also means how much data an application accesses at a given time, and if a particular access pattern exists, stops, and changes. This knowledge is used to guide the level of data prefetching in prefetchd.
CHAPTER 2. BACKGROUND

Figure 2.2: The Need for Elastic Prefetching

(a) Prefetching speedup benchmark dbt3-3

(b) Prefetching efficiency benchmark dbt3-3
Chapter 3

Elastic Prefetching

At a high level, *prefetchd* consists of several stages: *trace collecting* which accumulates information for each application I/O request, *pattern recognition* which aims to understand the access patterns for a series of requests, *block prefetching* which moves data from the drive to the cache in the background, and *feedback monitoring* that compares old prefetch operations against actual application requests, and adjusts accordingly. Figure 3.1 shows the prefetchd architecture. Note that prefetchd is operating system agnostic, as in this work it is designed and implemented in the user space. We envision that a future implementation of prefetchd can be integrated within the I/O stacks of different operating systems.

3.1 Trace Collection

Prefetchd collects the I/O events with the help of the operating system. Typically, this information includes timestamps, the name and identifiers of process, the request type and amount. The trace collection facility accumulates every I/O request made to disk and stores it for the prefetchd pattern recognizer. All I/O requests are considered those that actually reach disk, and those made by an application that are satisfied
in the system cache. The I/O requests may come from several different applications running on multiple CPUs, and 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, sorted by time, and stored in a buffer for the later use.
3.2 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 disk events, decides whether, where, and how much to prefetch, performs the prefetch request, and sleeps for the remainder of the interval. The polling interval determines how long disk 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 0.10 seconds by default. Occasionally, a 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 disk 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 following 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 an 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. 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 blocks requested exceeds a certain threshold in 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. The default value of 0.60 means 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 on an application, it simply begins with the next block contiguous to the previous request. The stop block is set by extrapolating into the future.

### 3.3 Block Prefetching

The amount of data to prefetch once a pattern has been recognized is determined with the goal of reading data from disk into the system cache, but only those blocks that the application will actually request in the near future. In this implementation, there are two key parameters that control how much data can be potentially be prefetched:

- **Application throughput scale factor** is the most important parameter. The product of this and the polling interval is called the prefetch throughput and determines the stop block during a prefetch operation $stop\_block =$
CHAPTER 3. ELASTIC PREFETCHING

\[ \text{polling\_interval} \times \text{prefetch\_throughput} \]. The optimal value for this scale factor is application-specific and can be adjusted by feedback, but experiments show that values near 1.0 typically work well. 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, a lower value means that some portion of the application’s read requests will still be expected to go to disk.

- **Maximum throughput**: During the time interval when prefetching is occurring, prefetchd is careful to avoid saturating the available read bandwidth to the disk with prefetch requests at the expense of actual application requests. 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. The maximum prefetch throughput limits the prefetch throughput to prevent this. The value of this parameter depends on the characteristics of the drive. For our testing, we measured the raw throughput from each disk by reading a large, uncached file, and using this as the maximum.

Once the quota of number of blocks to prefetch for one application during an interval is found, prefetchd simply issues a number of readahead calls with a starting block number and the number of blocks to read for that particular stride. Multiple readahead calls may be issued in a given interval if the access pattern is not consecutive. The starting block is advanced after each operation.

The details of the cache management itself is left to the underlying operating system. Prefetchd relies on the existence of such a cache and basically fills it by reading blocks ahead of time and hoping they remain cached. This limits the amount of information available to prefetchd and requires careful control over the extent of prefetching.
CHAPTER 3. ELASTIC PREFETCHING

3.4 Feedback Monitoring

Feedback monitoring is at the heart of elastic prefetching. At the end of each polling interval, prefetchd compares the actual application disk reads against the history of recently-issued prefetch operations. When a prefetch operation is issued, it is placed in a history ring buffer. By default, this buffer keeps operations for a history time of 2.0 seconds. Since prefetchd does not have direct access to the kernel’s VM cache internals, this is a heuristic to estimate the time the pages from a prefetch operation are still stored in the system cache. (Details about how this method for generating performance metrics is discussed in detail in 4.3.1.) Before comparing the application reads against the history, prefetch operations older than the history time are purged from the buffer. For each read operation from the application, prefetchd iterates through the history buffer and finds whether the subsequent request was wholly or partially filled by a previous prefetch request.

There are two major types of feedback monitoring: spatial and temporal feedback.

3.4.1 Spatial Feedback

Spatial Feedback refers to monitoring which blocks on disk were successfully prefetched (predicted and used) in the past and which were not. The goal is to avoid prefetching from regions of the disk with a high number of mispredictions.

The entire disk is split into logical regions each 1MB in size. When a prefetch request is purged from the history, if any portion of a predicted request was satisfied by a subsequent application disk access, the predicted and used counter for that region is incremented, otherwise, only the predicted counter is incremented. Each of these regions has a single bit (green or red) indicating whether or not to prefetch from this region as well as counters for successful and unsuccessful prefetch requests.

Prefetchd maintains a bitmap for the green/red bits for all the regions. Each
CHAPTER 3. ELASTIC PREFETCHING

A single bit (green or red) for the whole region is computed by comparing the ratio of successful prefetched operations from that region to the total prefetch operations from that region. If the ratio is below the \textit{red block threshold}, the bit for that region is marked red, meaning prefetch requests from this region will be not actually be sent to disk, but will otherwise still be stored and considered in the history.

Spatial feedback was our first attempt at implementing a feedback mechanism with elastic prefetching. In practice, we found that it produces limited gains over a fixed aggressiveness, and is not as good as temporal feedback. In the rest of the paper, we discuss results from elastic prefetching with temporal feedback only.

3.4.2 Temporal Feedback

Temporal Feedback allows prefetchd to, in addition to monitoring and adjusting its operation in space, check whether it has been more or less successful in the recent past and adjust its aggressiveness accordingly. This relies on several counters that monitor recent prefetch performance.

When prefetchd iterates through the history buffer and finds whether a read request was wholly or partially filled by a previous prefetch request, it updates the accumulative record for that request. After iterating through the history, any portion of the application read that was not satisfied by prefetch operation is stored in the \textit{false negative} (unprefetched and used blocks).

When a prefetch request is purged from the recent history, prefetchd updates two counters: \textit{true positive} (prefetched and used blocks), \textit{false positive} (prefetched and unused blocks). It looks at the ratio of \textit{true positives} (prefetched and used blocks) to used blocks to see whether recent prefetching has been accurate, as well as the ratio of \textit{false positives} (prefetched and unused blocks) to the number of total prefetched blocks to see whether recent prefetching has been polluting the cache. Based upon these two ratios, it adjusts its aggressiveness higher to increase the hit rate or lower
CHAPTER 3. ELASTIC PREFETCHING

to reduce cache pollution.

(The same history buffer is also used for spatial feedback. Prefetchd keeps a history for each disk block to measure block utility, that is, whether that block was prefetched usefully. This is helpful when an application reads different blocks for multiple times. If a block was prefetched but not read by the application, it will not be prefetched again.)

Together, the true positive, false positive, and false negative counters are used to adjust the aggressiveness for the next interval. The motivation is to ramp up the aggressiveness until it becomes so high that other, useful pages are evicted from the system cache. We define two terms called accuracy and pollution based on these counters and decide whether the two terms are “good” or “bad”. The algorithm to adjust the aggressiveness considers the two to see whether prefetching is currently accurate and polluting.

Informally, prefetching is accurate when there are many true positives and few false negatives and prefetching is polluting when there are many false positives and few true positives. These ratios are compared to two parameters $\alpha$ and $\beta$ to test for accurate and polluting:

- If $\text{accuracy} = \frac{\text{truepos}}{\text{truepos} + \text{falseneg}} > \alpha$, then prefetching over the previous interval is considered accurate. We use a default value of 0.90.
- If $\text{pollution} = \frac{\text{falsepos}}{\text{falsepos} + \text{truepos}} > \beta$, then prefetching over the previous interval is considered polluting. We use a default value of 0.50.

After determining whether prefetching is currently accurate and polluting, prefetchd scales its aggressiveness accordingly:

- **Not Accurate and Not Polluting** Keep the current aggressiveness as-is, hoping to increase accuracy. This is typical when ramping up on a series of accesses.
CHAPTER 3. ELASTIC PREFETCHING

- **Accurate and Not Polluting** The current access pattern seems highly predictable, and there has not been much useless prefetching so set the aggressiveness much higher. By default, prefetchd multiplies the current scale factor by 4.00.

- **Accurate and Polluting** The current access pattern seems highly predictable, but there is too much useless prefetching, increase the aggressiveness slightly hoping to preserve accuracy. By default, prefetchd multiplies the current scale factor by 2.00.

- **Not Accurate and Polluting** The current access pattern seems highly unpredictable, and there also is too much useless prefetching. Moderating the amount prefetched would not improve the accuracy, so throttle the aggressiveness to a lower value. By default, prefetchd multiplies the scale factor by 0.75.

Table 3.1 summaries the mechanism of the proposed feedback monitoring.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
<th>Scale factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Accurate and Not Polluting</td>
<td>Maintain current state</td>
<td>-</td>
</tr>
<tr>
<td>Accurate and Not Polluting</td>
<td>Increase prefetching aggressively</td>
<td>x4</td>
</tr>
<tr>
<td>Accurate and Polluting</td>
<td>Increase prefetching</td>
<td>x2</td>
</tr>
<tr>
<td>Not Accurate and Polluting</td>
<td>Decrease prefetching</td>
<td>x0.75</td>
</tr>
</tbody>
</table>

The design of table-driven elastic prefetching as well as the values of the parameters for elastic prefetching \(\alpha, \beta\), and the adjustments to the scale factors may seem arbitrary, but they have been developed and refined over several iterations. We discuss the development of these in detail in 4.3.
Chapter 4

Implementation

We have implemented a prototype of prefetchd in Linux systems that runs in userspace and is integrated with the Linux page cache. This way, prefetchd is completely transparent to user applications so no recompilation, or re-linking is required. Another motivation is to avoid wasting physical memory for a driver-specific cache. This allows unused memory to be used for other purposes when not in use as a cache.

4.1 Event Collection

Prefetchd uses the same facility as the blktrace [9] 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 provides useful pieces of context that are not present in a traditional I/O event queue in the driver; events have timestamps and process ids and names of the originating process. Prefetchd can use this information to differentiate requests from different applications. Events can also be automatically filtered (read vs. write) with a mask before being delivered to prefetchd.
CHAPTER 4. IMPLEMENTATION

There is a timing disadvantage to the blktrace API. 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, 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. We plan to add file id in the future.

4.2 Readahead

The readahead system call in Linux [10] is designed to load pages from a particular file into the system page cache. There is one complication with the readahead call. While readahead on a block device is 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. Measurements indicated that although sustained read throughput from the buffer cache is 3x faster than to the SSD, sustained read throughput to the page cache is 10x faster. The current implementation uses a file spanning the entire disk with loopback device to take advantage of the faster page cache.

4.3 Development of Elastic Prefetching

We tried several techniques to develop elastic prefetching before settling on a table-based approach and then searching for good values for each parameter. At first, prefetchd was limited to consecutive and strided access patterns with a fixed aggressiveness. But we discovered that the implementation did not give good results except for very static workloads even when the aggressiveness was varied. Sometimes
increasing it to improve one benchmark would hurt the performance on another.

### 4.3.1 Implementing Feedback Monitoring

The first major step to improve performance was to get a better idea about the nature of the problem. At first we only had coarse aggregate statistics about the of the prefetcher, but no insight into how the performance was varying over time. Before we could come up with an improved prefetch algorithm, we had to have more information such as the hit rate and miss rate. This was challenging since prefetchd has no direct management of the VM cache – it relies on the underlying OS to do this.

The first attempt to get dynamic performance numbers was to write a cache simulator that could be run against a captured trace of the prefetchd output log. We first used a simulator with no replacement. We discovered that the hit rate with (no replacement) varies with the aggressiveness we configured at the start, but was extremely high for most benchmarks. This suggested that biggest issue could be a large number of extraneous prefetch operations. We wrote an improved cache simulator with an LRU policy and limited the size of the simulated cache to the memory size of the test machine. Running several traces through this analysis program also suggested that there were too many of extraneous prefetch operations.

An offline cache simulator is helpful, but is not suitable for operation during runtime. Even with a simulated LRU policy, the VM cache implementation differs in several major ways. The biggest is that in Linux systems, memory is dynamically adjusted between being used for the system cache and used by applications. A static size for the simulated cache would not work well. In order to get a more accurate cache performance numbers that could be collected at runtime, we tried using the Linux mincore system call. This is intended to be used to determine whether a given vector pages in process memory is currently residing in system memory. Unfortunately, the
setup costs for using mincore are large. Converting from a disk block to a process address requires making additional calls to mmap and munmap which slowed down prefetchd tremendously.

After struggling with the mincore technique, we attempted instead to use a heuristic approach to measuring cache performance. This is designed around a history buffer containing prefetch operations as described earlier. We ran several tests comparing the results of using different history depth sizes against a static analyzer and determined that 2.0 seconds was a reliable value for the depth of the buffer.

4.3.2 Implementing Adjustable Aggressiveness

Once we had a reliable monitoring mechanism, we tried several approaches to implementing feedback. Our major goal was to reduce the number of wasted prefetch operations. Our first attempt to implement feedback was with the spatial feedback technique described in 3.4.1. In practice, we found that it produces limited gains over a fixed aggressiveness. This occurs regardless of whether or not the default state is red or green. The culprit seems to still be excessive prefetching throttling the bandwidth.

We realized that the single lever of control (analogous to accuracy as used in [11]) provides only a single degree of control. Based on their approach, we developed our terms for accuracy and pollution based on what we could measure with the prefetch history mechanism and attempted to devise a table-driven technique for adjusting aggressiveness.

We tried a direct mapping between four states of accurate and polluting and fixed values of aggressiveness, but found that approach did not work well. We settled on an approach to ramp the aggressiveness up and down based on scaling the previous aggressiveness and clamping it on the high and low end. We also performed a 2-D parameter search for reasonable values for $\alpha$ and $\beta$ to test for accurate and polluting. The default values of 0.90 and 0.50 gave good results.
Chapter 5

Evaluation

5.1 Experiment Setup

5.1.1 Benchmarks

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

DBT3 (Database Test Suite) [12] is an open source implementation of the TPC-H benchmark. It is a decision support benchmark with business oriented ad-hoc queries. We create and populate the database in Postgres and evaluate a subset of 22 queries. We avoid some queries because they take a significant time to run.

BLAST (Basic Local Alignment Search Tool) [13] 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 12GB of non-redundant DNA sequences.

LFS, Sprite large file benchmark [14], performs both reads and writes on a large file, as well random and sequential read of the file. We use a file size of 100000 MB.
CHAPTER 5. EVALUATION

Figure 5.1: Zoomed-out view of block traces for a dbt3-13 query on the SSD1 and elastic prefetch operations. Blue dots in the figures represent real data access, and red arrows represent data prefetching. The x-axis represents time in seconds and the y-axis represents the 512-byte block number on disk.

and an I/O size of 1024 MB.

**Websearch** [15] contains two block-level I/O traces collected from a web search engine. We replay the traces using the replayer tool typically at 12 times normal speed and one work thread. We and report the total I/O wait time as performance metric. These come in SPC format which is a text format containing a timestamp, offset in disk, operation size, type of operation, and thread id.

### 5.1.2 Trace Replayer

In order to play the traces in SPC format 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. 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.

In addition to supporting the replaying of SPC traces, the replayer also supports
traces captured with blktrace. Our goal here was to use this when benchmarks might
be too difficult or time-consuming to run over and over. But we discovered that this
second usage has limited value. When a disk is traced using blktrace and replayed,
the I/O read sizes often conflict. The reason for this seems to be the way Linux
read calls are split and combined in the I/O scheduler. For example, a blktrace entry
indicating that a read operation had a size of 256 512-byte blocks, when replayed
Figure 5.3: Prefetchd aggressiveness in runtime for a BLAST-N benchmark on the SSD1 RAID and elastic prefetch operations. The solid blue line represents I/O read operations per second over time and the dashed red line represents prefetchd scale factor i.e. aggressiveness over time.

could result in two separate read calls of size 248 and 8. In addition, very small timing delays caused by I/O readiness are difficult to implement in userspace, resulting in somewhat unpredictable delays. All of these cause the performance of prefetchd to vary between an original application run and a replayed run. We chose to run the benchmarks normally for our tests.
CHAPTER 5. EVALUATION

Figure 5.4: Prefetchd performance using elastic prefetching for different benchmarks and devices. Benchmark speedup is on the y-axis. The device models corresponding to each name are described in Table 2.1.

Figure 5.5: Prefetchd accuracy using elastic 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. The device models corresponding to each name are described in Table 2.1.

5.1.3 Test Machine

The test system has Linux kernel 2.6.28 with an Intel Core2 Quad CPU at 2.33 GHz and 8GB RAM. We tested two SSDs and one hard drive, as listed in Table 2.1. We also created three level-0 RAIDs for two of the SSDs and the hard drive. The storage device is format 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 from the filesystem from generating metadata updates which would require writes to the device.
CHAPTER 5. EVALUATION

Figure 5.6: Prefetchd efficiency using elastic prefetching for different benchmarks and devices. Benchmark efficiency 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. The device models corresponding to each name are described in Table 2.1 and is intended to improve the I/O throughput.

5.2 Elastic Prefetching at Work

In Figures 5.1 and 5.2, we show the operation of elastic prefetching on two levels: Figure 5.1 shows the zoom out view of high level data access and the actions taken by prefetchd on a dbt3-13 benchmark running on the Samsung SSD; and 2) Figure 5.2 presents the zoom in view of a data region from the BLAST-N benchmark on a mirror RAID of two OCZ. The blue streaks 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 dots merge to form gently sloping lines. The actions of prefetch operations in response to application reads is shown by the red arrows. The horizontal position of a red 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,
CHAPTER 5. EVALUATION

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 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 need of the application.

We also measure the aggressiveness of the prefetchd against the performance of the real application. Figure 5.3 presents the numbers collected from running BLAST-N. It is clear that prefetchd is able to follow the application trend closely and adjust its aggressiveness accordingly.

5.3 Performance Speedup

We evaluate prefetchd by running all four benchmarks. As shown in Fig. 5.4, elastic 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 with prefetchd by the run time without prefetchd. 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. For the database benchmark, 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 of less aggressive prefetching.

Prefetchd does not always provide good performance benefits, e.g., when running
the query script dbt3-1, it experiences 1 or 2% slowdowns on some devices. Although it partly confirms previous belief [16, 17] that because SSDs have good random access performance, the help from data prefetching can be limited, we believe that the feedback monitoring component in our elastic prefetching can be enhanced to minimize this effect.

5.4 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. 5.5 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 elastic prefetching predicts with about 70% accuracy.

5.5 Prefetching Efficiency

We further examine prefetchd’s efficiency that is defined as the ratio of the amount of prefetched data (true and false positives) and the amount of data read by the application. A lower efficiency indicates less data preloaded by the prefetchd. On average, prefetchd reads 77% more data than the benchmarks, with 60% for the single SSD and average 90% for the two RAIDs. Being the fastest device of four,
SSD2 RAID tends to read more data and have a lower efficiency. Fig. 5.6 presents the prefetching efficiencies on all four devices.

5.6 Scalability

In this section, we want to evaluate the scalability of elastic prefetching in two dimensions, that is, where there are different number of concurrent applications, and for the applications with varied I/O requirements.

In the first scalability test, we used the replayer to play one and two concurrent threads of the Websearch-1 trace at the same time on the SSD1 RAID, to evaluate how different prefetching techniques would scale for concurrent applications. Figure 5.7 shows the prefetching speedup for one and two instances. For one application instance, both aggressive and elastic prefetching achieve more than 10% speedup while normal prefetching has a modest 1% improvement. However, when there are two concurrent instances, both normal and aggressive prefetching suffer a great deal of performance loss of 3% and 20%, respectively. In comparison, our prefetchd scales well in this case and allows two instances to achieve 11% speedup.

In the second scalability test, we choose to adjust the replay speed of the Websearch-1 trace on the SSD1 RAID in three settings, i.e., low, medium, and high speed. Each speed setting doubles the previous speed, and there are two application instances that are running at the same time. The results are presented in Figure 5.8. Clearly, both normal and aggressive prefetching cannot scale when the application becomes more I/O intensive - in most cases they lead to various degrees of application slowdowns. In contrast, our prefetchd again scales well in this test. It delivers performance improvements of 4%, 9%, 23% for low, medium and high speed tests, respectively.
Figure 5.7: Performance of prefetchd for one and two concurrent threads for the Websearch-1 benchmark on the SSD1 RAID. The y-axis represents the measured speedup.
Figure 5.8: Performance of prefetchd under different replay speeds of the Websearch-1 benchmark on the SSD1 RAID with a single concurrent thread. The low, medium, and high descriptions correspond to speeds of 6x, 12x, and 18x the original Websearch-1 trace speed. The y-axis represents the measured speedup.
Chapter 6

Related Work

6.1 Prefetch Techniques for Disks

There exists a rich set of prior research on data prefetching on hard disks and some representative techniques include probability graph [18], data compression [19], data mining [20], address tracking [21][4], compiler support [22], and hints [23][3]. Our proposed elastic prefetching technique is orthogonal to techniques previously applied to hard disks in the sense that we work on the adaptation of prefetching aggressiveness in the runtime, which can be incorporated with the existing prefetching techniques. Further, our work focuses on emerging flash-based solid-state drives and SSD based RAIDs whose high throughput provides new opportunities and challenges for data prefetching.

Note that SSD devices are performing data prefetching in a small scale by utilizing parallel I/Os and internal memory buffer. Work has been started to measure and understand this effect [24, 2]. In comparison, our proposed prefetching is designed and implemented on 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 [25]. 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 benefits [26]. Linux developers also realize the need to have a tunable I/O size as well as the need for more aggressive prefetching [27]. Development efforts on improving prefetching performance on SSDs are ongoing, and we believe that our findings will be beneficial in this area.

Researchers have realized the importance of data prefetching on SSDs, for example, [28] shows that prefetching can be used for energy efficient sorting on SSDs. Our positive results also demonstrate the potential of data prefetching.

We would also like to point out that some researchers expressed reservations against data prefetching on solid-state drives. IotaFS (in a technical report) chooses not to implement prefetching among the file system optimizations it used for SSDs [16]. In addition, FlashVM [17] 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 regression, which we plan to further investigate in the future.

### 6.2 Prefetch Techniques from Main Memory

Prefetching techniques are common for fetching data from main memory on high-performance processors to processor caches and similar challenges about storage bandwidth and storage pollution apply. Feedback directed prefetching has been proposed for these architectures most prominently in [11].

They incorporate **accuracy**, **lateness**, and **pollution**. Directly measuring lateness and pollution is difficult because prefetchd does not directly manage the page cache. The table-driven adjustments used by prefetchd are similar to the lookup table de-
CHAPTER 6. RELATED WORK

scribed in their implementation. These three states measurements are incorporated into a 12-state transition table which can adjust its aggressiveness appropriately. Our technique uses a smaller transition table (although we tried several approaches as described earlier).

Our terms also differ slightly from theirs. We define accuracy = \( \frac{\text{true pos}}{\text{true pos} + \text{false neg}} \), while they define accuracy = \( \frac{\text{true pos}}{\text{true pos} + \text{false pos}} \).

We measure cache pollution as
\[
pollution = \frac{\text{false pos}}{\text{false pos} + \text{true pos}} = \frac{\text{amount not prefetched but read and used}}{\text{amount read and used}},
\]
whereas they define it as
\[
pollution = \frac{\text{number of demand misses caused by prefetching}}{\text{number of demand misses}}.
\]

The differing definition here is understandable since prefetchd does not directly manage the VM page cache, it cannot know which misses were caused by prefetching and which were not.

Similarly, they define lateness = \( \frac{\text{number of late prefetches}}{\text{number of useful prefetches}} \). Here again, the design of prefetchd makes lateness difficult to measure. All prefetches are scheduled before reads to a region occur, although the Linux readahead call makes this only advisory. But since there is a two-layer loopback device, it might happen that reordering of operations to the underlying disk causes an application read to occur before a prefetch occurs. This would not only waste disk bandwidth, but might cause a severe seek penalty. One way of monitoring for this could be to implement a second history buffer which stores and monitors actual reads and looks for subsequent prefetch operations to those same blocks. This may be worth investigating in the future.

6.3 Other Related Techniques

FAST is a recent program that focuses on shortening the application launch time and utilizes prefetching on SSDs for quick start of various applications [29]. 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 is the most
similar to prefetchd’s and even uses the blktrace API. However, it uses an LBA-to-
inode mapper instead of relying on a loopback device like prefetchd. Our approach
differs in that it can handle multiple simultaneous requests and includes a feedback
mechanism. With this wider range of data-intensive applications in mind, prefetchd
aims to improve the overall performance of generic applications.
Chapter 7

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 pattern and dynamically retrieves data to match both drive characteristics and application needs. Currently, the prefetcher works well for a number of I/O intensive applications that perform significant computations on data read from disk. For those applications that perform minimal processing and read data near a disk’s maximum throughput, prefetchd can be configured to avoid too aggressive prefetching. We implement a prototype in Linux and conduct a comprehensive evaluation on different hard drive, SSDs, as well as SSD RAIDs, with a wide range of data-intensive applications and benchmarks. The prototype are able to achieve 20% speedups, for Websearch engine traces, BLAST and TPC-H like benchmarks, across various storage devices. The results show that prefetchd achieves high prefetching accuracy of 98%, and low data footprint of 50%.

In the future, there are some additional features that we would like to add to improve the performance of prefetchd:

- The measured application throughput should also account for writes issued by the application, possibly weighting those more heavily since SSD random writes...
CHAPTER 7. CONCLUSIONS

are very costly in terms of time.

- In addition to a quota of the number of bytes to read during a time interval, attempt to limit the total number of readahead operations since individual operations incur a fixed overhead time cost.

- For strides where the empty space between read requests is small, it may be more effective to issue larger readahead calls instead of splitting the requests across several calls.

- Automate the profiling of disks by running a suite of tests similar to those described in our evaluation and using the results to adjust prefetchd’s configuration parameters.

- A more powerful planned feature is a history-based prefetcher. Instead of simply detecting a strided access pattern, the prefetcher could prefetch any previously-seen access trace that matches the current access pattern.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


Appendix A

Prefetchd Psuedo-Code

# Configuration parameters with default values. These can be changed with environment variables.
double interval = 0.025
double scale = 1.0
double consec_tol = 0.60
double history_time = 1.0
prefetch_adaptive = False

# Max available throughput for every prefetcher.
max_throughput = 100e6

red_block_threshold = 0.0
ratio_beta = 0.5
ratio_alpha = 0.9

def main():
    # Get environment variables that override default configuration parameters
    update_cfg_params()

    # Path to device for readahead.
    readahead_path = "/mnt/sdc1/tmp/span"

    # Path to run block trace on:
    trace_path = "/dev/loop0"
APPENDIX A. PREFETCHD PSUEDO-CODE

```python
if argc > 1:
    trace_path = argv[1]

if argc > 2:
    readahead_path = argv[2]

max_block = os.stat(readahead_path).st_size / blk_size
blk_size = 512

# rg_region has two arrays predicted_and_read and predicted
# shown here as 1:1 mapping.
rg_region = (array(max_block), array(max_block))

# Initialize hash table entries
for i in range(ht_len):
    pf_table[i] = prefetcher_state()

trace_start(trace_path)

while not exit_flag:
    think_start_time = time.time()

    # Collect trace events across all CPUs
    events = trace_read()
    for e in events:
        if e.pid == 0 or e.is_write or not e.is_queued:
            remove e

    # Sort by timestamp
    sort(events)

    for ba in events:
        hash_id = (ba.pid + (ba.start_block / 4000000) % 16851) % len(hash_table)
        pf = hash_table[hash_id]
        pf.event_cnt += 1
        pf.blk_cnt += ba.n_blocks

        start_block = ba.start_block
        n_blocks = ba.n_blocks

        if ba.t < pf.t_min:
            pf.t_min = ba.t
```

46
APPENDIX A. PREFETCHD PSUEDO-CODE

```python
if ba.t > pf.t_max:
    pf.t_max = ba.t

for pp in pf.history:
    overlap_start, overlapping_blocks, remain_start, remain_end
        = pp.reduce_overlap(ba.start_block,
                             ba.n_blocks)
    if overlapping_blocks > 0:
        start_block = remain_start
        n_blocks = remain_end - remain_start + 1

    pf.recent_unprefetch_and_used += n_blocks
    pf.recent_miss += n_blocks
    pf.recent_hit += ba.n_blocks - n_blocks

# Update consecutive, reverse, and strided counters
# based on this event
    pf.update_consec_reverse_strided(ba.start_block, ba.
                                      start_block + ba.n_blocks)

for pf in hash_table:
    if pf.event_cnt == 0:
        pass

    pf.app_throughput = pf.blk_cnt * blk_size / (pf.t_max - pf.t_min)

    if pf.consec_pct() > consec_tol:
        prefetch_throughput = max(scale * pf.app_throughput, 
                                   max_throughput)
        pf.start_block = min(pf.curr_block_hi, max_block)
        pf.stop_block = min(pf.curr_block_hi + interval * 
                             prefetch_throughput / blk_size, max_block)

        # Also check that stop block is not before the end of
        # a previous prefetch operation within some
distance.
        pf.prefetch_enable = True

    # Remove stale history starting from the oldest.
```
while pf.history[0].t + history_time < think_start_time:
    e = pf.history.pop(0)
    # And update the recent hit percentages of recent
    # prefetch operations accordingly.
    pf.recent_prefetch_and_used += e.get_used_blocks()
    pf.recent_prefetch_and_unused += e.get_unused_blocks()
    
    # Mark spatial prediction
    for j in range(e.n_blocks):
        if e.used_array[j]:
            rg_region[e.start_block + j].predicted_and_read += 1
        else:
            rg_region[e.start_block + j].predicted += 1

if prefetch_adaptive:
    scale = adjust_aggressiveness(
        pf.recent_prefetch_and_used,
        pf.recent_prefetch_and_unused,
        pf.recent_unprefetch_and_used,
        scale)

    # Create a new history entry
    pf.history.append((think_start_time, pf.start_block, pf.
        stop_block - pf.start_block + 1))

    # Check the spatial percentage
    if rg_region[(pf.start_block + pf.stop_block) / 2].
        predicted_and_read / (predicted_and_read + predicted)
        >= red_block_threshold:
        perform_readahead(readahead_path, pf.start_block, pf.
            stop_block)

    # Reset all interval counters
    (pf.recent_prefetch_and_used, pf.
        recent_prefetch_and_unused, pf.
        recent_unprefetch_and_used, pf.recent_miss, pf.
        recent_hit) = (0, 0, 0, 0, 0)
    
    (pf->event_cnt, pf.blk_cnt, pf.consec_blk_cnt, pf.
        reverse_blk_cnt, pf.strided_blk_cnt) = (0, 0, 0, 0, 0)
    think_end_time = time.time()
APPENDIX A. PREFETCHD PSUEDO-CODE

```python
sleep_time = interval - (think_end_time - think_start_time)
sleep(sleep_time)

def adjust_aggressiveness(true_pos, false_neg, false_pos, old_scale):
pct = 0.

if true_pos + false_neg > 0:
pct = true_pos / (float)(true_pos + false_neg)
accurate = False

if pct > ratio_alpha:
    accurate = True

polluting = False

pol_pct = false_pos / (float)(false_pos + true_pos)

if pol_pct > ratio_beta:
polluting = True

new_scale = old_scale

if accurate == False and polluting == False:
    new_scale = 1.00 * old_scale
elif accurate == True and polluting == False:
    new_scale = 4.00 * old_scale
elif accurate == True and polluting == True:
    new_scale = 2.00 * old_scale
elif accurate == False and polluting == True:
    new_scale = 0.75 * old_scale

if new_scale < 1.00:
    new_scale = 1.00

if new_scale > 8.00:
    new_scale = 8.00

return new_scale
```

49
Appendix B
Prefetchd C Source Code

#define FILE_OFFSET_BITS 64
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#define USE_GNU /* For readahead() */
define _XOPEN_SOURCE 600 /* posix_fadvise */
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>
#include <math.h>
#include <assert.h>
#include <signal.h>
#include <string.h>
#include <sys/ioct1.h>
#include <poll.h>
#include <errno.h>
#include <sched.h>
#include "bitarray.h"
#include "blktrace_api.h"
#include "cache-sim.h"

double ratio_beta = 0.5;
double ratio_alpha = 0.9;

double adjust_aggressiveness(
  int true_pos,
  int false_neg,
  int false_pos,
  double old_scale,
  FILE *fp_trace
)
{
  double pct = 0.0;
  if (true_pos + false_neg) {
    pct = true_pos
    / (double) (true_pos + false_neg);
  }
  int accurate = 0;
  if (pct > ratio_alpha)
    accurate = 1;
  int polluting = 0;
  double pol_pct = false_pos / (double) (false_pos + true_pos);
  if (pol_pct > ratio_beta) {
    polluting = 1;
  }
APPENDIX B. PREFETCHD C SOURCE CODE

```c
printf(fp_trace, "pct=%lf\alpha=%lf\beta=%lf\n\", pct, ratio_alpha, ratio_beta);

double new_scale = old_scale;
if (accurate == 0 && polluting == 0)
    new_scale = 1.00 * old_scale;
else if (accurate == 1 && polluting == 0)
    new_scale = 4.00 * old_scale;
else if (accurate == 1 && polluting == 1)
    new_scale = 2.00 * old_scale;
else if (accurate == 0 && polluting == 1)
    new_scale = 0.75 * old_scale;
if (new_scale < 1.00)
    new_scale = 1.00;
if (new_scale > 8.00)
    new_scale = 8.00;
return new_scale;
}
#endif
#define BUF_SIZE (512 * 1024)
#define BUF_NR (4)
#define MAX_CPUS (16)
static int exit_flag;
static int act_mask = 0U;
static unsigned long buf_size = BUF_SIZE;
static unsigned long buf_nr = BUF_NR;
static char buts_name[32];
static double red_block_threshold = 0.0;
static void sighandler(int sig)
{
    exit_flag = 1;
}
static int start_trace(int fd)
{
    struct blk_user_trace_setup buts;
    memset(&buts, 0, sizeof(buts));
    buts.buf_size = buf_size;
    buts.buf_nr = buf_nr;
    buts.act_mask = act_mask;
    if (ioctl(fd, BLKTRACESETUP, &buts) < 0) {
        perror("BLKTRACESETUP");
        return -1;
    }
    if (ioctl(fd, BLKTRACESTART) < 0) {
        perror("BLKTRACESTART");
        return -1;
    }
    memcpy(buts_name, buts_name, sizeof(buts_name));
    return 0;
}
static void stop_trace(int fd)
{
    if (fd <= 0)
    return;
    /* should be stopped, just don’t complain if it isn’t */
    ioctl(fd, BLKTRACESTOP);
    if (ioctl(fd, BLKTRACETEARDOWN) < 0)
        perror("BLKTRACETEARDOWN");
}
#endif
#define MAXPATHLEN (2048)
static int get_dropped_count(const char *buts_name)
{
    int fd;
    char tmp[MAXPATHLEN + 64];
    static char default_debfs_path[] = "/sys/kernel/debug";
    ...
```

51
APPENDIX B. PREFETCHD C SOURCE CODE

snprintf(tmp, sizeof(tmp), "%s/block/%s/dropped", default_debugfs_path, buts_name);
fd = open(tmp, O_RDONLY);
if (fd < 0) {
    /* this may be ok, if the kernel doesn’t support dropped counts */
    if (errno == ENOENT)
        return 0;
    printf(stderr, " Couldn’t_open_dropped_file_%s\n", tmp);
    return -1;
}
if (read(fd, tmp, sizeof(tmp)) < 0) {
    perror(tmp);
    close(fd);
    return -1;
}
close(fd);
return atoi(tmp);

static int event_cmp(const void *a, const void *b)
{
    const struct blk_io_trace *e0 = *(const struct blk_io_trace **a);
    const struct blk_io_trace *e1 = *(const struct blk_io_trace **b);
    if (e0->time < e1->time) {
        return -1;
    } else if (e0->time > e1->time) {
        return 1;
    }
    return 0;
}

double gettime_double()
{
    struct timespec tp;
    if (clock_gettime(CLOCK_REALTIME, &tp) < 0) {
        perror("clock_gettime");
    }
    return tp.tv_sec + tp.tv_nsec * 1e-9;
}

int sleep_double(double t)
{
    struct timespec tp;
    tp.tv_sec = (time_t) t;
    tp.tv_nsec = (t - tp.tv_sec) * 1e9;
    return nanosleep(&tp, NULL);
}

struct timespec double_to_timespec(double t)
{
    assert(t >= 0.0);
    struct timespec tp;
    tp.tv_sec = (time_t) t;
    tp.tv_nsec = (t - tp.tv_sec) * 1e9;
    return tp;
}

double timespec_to_double(struct timespec tp)
{
    return tp.tv_sec + tp.tv_nsec * 1e-9;
}

static char default_debugfs_path[] = "/sys/kernel/debug";
static int max_events = 65536;

struct blk_watch {
    int fd;
    int ncpus;
    struct pollfd trace_fd[MAX_CPUS];
    char read_buf[MAX_CPUS];
    int used_bytes[MAX_CPUS];
    int processed_bytes[MAX_CPUS];
}
APPENDIX B. PREFETCHD C SOURCE CODE

```c
struct blk_io_trace *
unsigned int event_cnt;
/* Replay mode */
int replay;
double t_prev;
int peek_cnt;
FILE *p_trace;
int trace_enable;
};
/* Forward declarations. */
int blkwatch_close(struct blk_watch *bw);
int blkwatch_init(struct blk_watch *bw, const char *path)
{
    int fd;
    int ncpus;
    FILE *p_trace;
    struct stats st;
    int rc;
    memset(bw, 0, sizeof(*bw));
    rc = stat(path, &st);
    if (rc < 0) {
        perror("stat");
        return -1;
    }
    bw->event_cnt = 0;
    if (!S_ISBLK(st.st_mode)) {
        /* Run trace on block device. */
        fd = open(path, O_RDONLY | O_NONBLOCK);
        if (fd < 0) {
            blkwatch_close(bw);
            return -1;
        }
        ncpus = sysconf(_SC_NPROCESSORS_ONLN);
        if (ncpus < 0) {
            printf(stderr, "sysconf(_SC_NPROCESSORS_ONLN) failed\n");
            blkwatch_close(bw);
            return -1;
        }
        if (ncpus > MAX_CPUS) {
            printf(stderr, "ncpus=%d > max=%d\n", ncpus, MAX_CPUS);
            blkwatch_close(bw);
            return -1;
        }
        if (start_trace(fd) < 0) {
            blkwatch_close(bw);
            return -1;
        }
        int i;
        for (i=0; i<ncpus; i++) {
            char buf[80];
            snprintf(buf, sizeof(buf), "%s/block/%s/trace%d",
                default_debugfs, path, buts_name, i);
            bw->trace_fd[i].fd = open(buf, O_RDONLY | O_NONBLOCK);
            if (bw->trace_fd[i].fd < 0) {
                perror(buf);
                break;
            }
            bw->trace_fd[i].events = POLLIN;
        }
        if (i != ncpus) {
            blkwatch_close(bw);
            return -1;
        }
    }
    return 0;
}
```

53
for (i=0; i<ncpus; i++) {
    bw->read_buf[i] = malloc(max_events * sizeof(struct blk_io_trace));
    if (bw->read_buf[i] == NULL) {
        blkwatch_close(bw);
        return -1;
    }
    bw->used_bytes[i] = 0;
}

bw->event = malloc(max_events * sizeof(struct blk_io_trace *));
if (bw->event == NULL) {
    blkwatch_close(bw);
    return -1;
}
else {
    /* Replay trace from ordinary file. */
    fprintf(stderr, "Ordinary file replay not implemented.\n");
    blkwatch_close(bw);
    return -1;
}

fp_trace = fopen("prefetch.trace", "wb");
if (!fp_trace) {
    blkwatch_close(bw);
    return -1;
}

bw->fd = fd;
bw->ncpus = ncpus;
bw->fp_trace = fp_trace;
bw->trace_enable = 1;
return 0;
}

int blkwatch_close(struct blk_watch *bw)
{
    int i;
    free(bw->event);
    for (i=0; i<ncpus; i++) {
        if (bw->read_buf[i] == NULL) break;
        free(bw->read_buf[i]);
    }
    get_dropped_count(buts_name);
    for (i=0; i<ncpus; i++) {
        if (bw->trace_fd[i].fd <= 0) break;
        close(bw->trace_fd[i].fd);
    }
    stop_trace(bw->fd);
    if (bw->fd > 0) close(bw->fd);
    if (bw->fp_trace)
        fclose(bw->fp_trace);
    return 0;
}

struct pred_linear {
    double sx;
    double sy;
    double ss;
    double ss2;
    double sy2;
    int n;

APPENDIX B. PREFETCHD C SOURCE CODE
APPENDIX B. PREFETCHD C SOURCE CODE

double x_min;
double y_min;
double x_max;
double y_max;
double slope;
double intercept;
};

void pred_linear_init(struct pred_linear *p)
{
    p->sx = 0.;
    p->sy = 0.;
    p->sxy = 0.;
    p->sx2 = 0.;
    p->sy2 = 0.;
    p->n = 0;
    p->sx_min = 10e37;
    p->sx_max = -10e37;
    p->sy_min = 10e37;
    p->sy_max = -10e37;
    p->slope = 0.;
    p->intercept = 0.;
}

void pred_linear_point(struct pred_linear *p, double x, double y)
{
    p->sx += x;
    p->sy += y;
    p->sxy += x * y;
    p->sx2 += x * x;
    p->sy2 += y * y;
    p->n++;
    if (x < p->sx_min)
    {
        p->sx_min = x;
    }
    if (x > p->sx_max)
    {
        p->sx_max = x;
    }
    if (y < p->sy_min)
    {
        p->sy_min = y;
    }
    if (y > p->sy_max)
    {
        p->sy_max = y;
    }
}

double pred_linear_score(struct pred_linear *p)
{
    int n = p->n;
    double sx = p->sx;
    double sy = p->sy;
    double sx2 = p->sx2;
    double sy2 = p->sy2;
    double sxy = p->sxy;
    double cov = sxy / n - (sx / n) * (sy / n);
    double stdx = sqrt(sx2 / n - (sx / n) * (sx / n));
    double stdy = sqrt(sy2 / n - (sy / n) * (sy / n));
    if (n == 0 || n == 1)
    {
        p->slope = 0.;
        p->intercept = 0.;
        return 0.;
    }
    /* Least-squares regression. */
    double m = (sy * sx - n * sxy) / (sx * sx - n * sx2);
    double h = (sx * sxy - sy * sx2) / (sx * sx - n * sx2);
    p->slope = m;
    p->intercept = h;
    return cov / (stdx * stdy);
}

static int set_sched(int yes)
{
    if (yes) {

55
APPENDIX B. PREFETCHD C SOURCE CODE

```c
struct sched_param sp;
memset(&sp, 0, sizeof(sp));
errno = 0;
sp.sched_priority = sched_get_priority_max(SCHED_FIFO);
if (sp.sched_priority < 0 && errno != 0) {
    perror("sched_get_priority");
    return -1;
}
if (sched_setscheduler(0, /* use our pid */,
    SCHED_FIFO, &sp) < 0) {
    perror("sched_setscheduler");
    return -1;
}
return 0;
}
#define MAX_PREFETCH_HISTORY (512)
struct prefetch_operation {
    double t;
    off_t start_block;
    size_t n_blocks;
    unsigned char *used_array;
};
#define MIN(a, b) ((a) < (b) ? a : b)
size_t prefetch_operation_get_used_blocks(struct prefetch_operation *pp) {
    size_t i, cnt = 0;
    for (i = 0; i < pp->n_blocks; i++) {
        if (pp->used_array[i])
            cnt++;
    }
    return cnt;
}
void reduce_overlap(struct prefetch_operation *pp,
    off_t start_block, size_t n_blocks,
    off_t *overlapping_start,
    off_t *remain_start,
    off_t *remain_end) {
    off_t a, b, c, d;
    off_t overlap_end;
    a = start_block;
    b = start_block + n_blocks - 1;
    c = pp->start_block;
    d = pp->start_block + pp->n_blocks - 1;
    *overlapping_start = 0;
    *overlapping_blocks = 0;
    *remain_start = 0;
    *remain_end = 0;
    if (b < c || d < a) {
        /* No overlap */
        return;
    } else if (a >= c) {
        /* Partial overlap */
        a = MIN(a, b);
        a = MIN(a, d);
        *overlapping_start = a;
        overlap_end = MIN(b, d);
```
APPENDIX B. PREFETCHD C SOURCE CODE

```c
*overlapping_blocks = overlap_end - overlap_start + 1;
if (b > d) {
    *remain_start = d + 1;
    *remain_end = b;
} else {
    /* Partial overlap
     * a==b
     * c==d
     */
    overlap_start = c;
    overlap_end = MIN(b, d);
    *overlapping_blocks = overlap_end - overlap_start + 1;
    *remain_start = a;
    *remain_end = c - 1;
}
/* Mark overlapping */
for (i=0; i<overlapping_blocks; i++) {
    pp->used_array[*overlap_start - pp->start_block + i]++;
}
}

typedef struct circ_buf_t {
    int head;
    int tail;
    unsigned int count;
    unsigned int len;
    unsigned int size;
    char *buf;
} circ_buf_t;

int circ_init(circ_buf_t *b, unsigned int len, unsigned int size)
{
    b->buf = malloc((len + 1) * size);
    if (!b->buf) {
        return -1;
    }
    b->len = (len + 1);
    b->size = size;
    b->head = 0;
    b->tail = 0;
    b->count = 0;
    return 0;
}

int circ_enq(circ_buf_t *b, const void *elm)
{
    int head = (b->head + 1) % b->len;
    if (head == b->tail) {
        return -1;
    }
    memcpy(b->buf + b->head * b->size, elm, b->size);
    b->head = head;
    b->count++;
    return 0;
}

int circ_deq(circ_buf_t *b, void *elm)
{
    if (b->head == b->tail) {
        return -1;
    }
    if (elm) {
        memcpy(elm, &b->buf[b->head * b->size], b->size);
    }
    b->tail = (b->tail + 1) % b->len;
    b->count--;
    return 0;
}

void *circ.peek(circ_buf_t *b, int index)
```
APPENDIX B. PREFETCHD C SOURCE CODE

\{
    if (index > b->count)
        return NULL;
    int i = (b->tail + index) % b->len;
    return &b->buf[i * b->size];
\}

unsigned int cinct(circ_buf_t *b)
\{
    return b->count;
\}

void ciclean(circ_buf_t *b)
\{
    if (b)
    {
        free(b->buf);
    }
\}

struct rg_region {
    size_t bytes_per_region;
    off_t max_id;
    unsigned int *predicted_and_read;
    unsigned int *predicted;
};

int rg_region_init(struct rg_region *rg, size_t bytes_per_region, off_t max_bytes)
\{
    rg->bytes_per_region = bytes_per_region;
    rg->max_id = max_bytes / bytes_per_region;
    rg->predicted_and_read = calloc(rg->max_id, sizeof(rg->predicted_and_read[0]));
    rg->predicted = calloc(rg->max_id, sizeof(rg->predicted[0]));
    return 0;
\}

void rg_region_predict(struct rg_region *rg, int read, off_t byte_offset, size_t n_bytes)
\{
    off_t id = byte_offset / rg->bytes_per_region;
    if (id >= rg->max_id) {
        printf(stderr, "rg_region_predict: bad offset %lu\n", byte_offset);
        return;
    }
    if (read) {
        rg->predicted_and_read[id] += n_bytes;
    } else {
        rg->predicted[id] += n_bytes;
    }
\}

double rg_pct(struct rg_region *rg, off_t byte_offset)
\{
    off_t id = byte_offset / rg->bytes_per_region;
    double pct = 1.0;
    if (id >= rg->max_id) {
        printf(stderr, "rg_region_pct: bad offset %lu\n", byte_offset);
        return 0.0;
    }
    if (rg->predicted_and_read[id] + rg->predicted[id]) {
        pct = (double) (rg->predicted_and_read[id] + rg->predicted[id]) / (double) (rg->predicted_and_read[id] + rg->predicted[id]);
    }
    return pct;
\}

int main(int argc, char *argv[])
\{
    set_sched(1);
    double interval = 0.025;
    double scale = 1.0;
    double consec_tol = 0.60;
    double history_time = 1.0;
    int prefetch_adaptive = 0;
    /* Max available throughput for every prefetcher. */
double max_throughput = 100e6;

/* Use readahead or post_advise to prefetch. The readahead call blocks until complete and the time loop tracks and accounts for this, so its performance is slightly better. */
int use_readahead = 1;

if (getenv("MAX_THROUGHPUT")) {
    max_throughput = strtod(getenv("MAX_THROUGHPUT"), NULL);
    printf(stderr, "Set max_throughput = %lf\n", max_throughput);
}

if (getenv("PREFETCH_ADAPTIVE")) {
    prefetch_adaptive = strtol(getenv("PREFETCH_ADAPTIVE"), NULL, 0);
    printf(stderr, "Set prefetch_adaptive = %ld\n", prefetch_adaptive);
}

if (getenv("RATIO_BETA")) {
    ratio_beta = strtod(getenv("RATIO_BETA"), NULL);
    printf(stderr, "Set ratio_beta = %lf\n", ratio_beta);
}

if (getenv("RATIO_ALPHA")) {
    ratio_alpha = strtod(getenv("RATIO_ALPHA"), NULL);
    printf(stderr, "Set ratio_alpha = %lf\n", ratio_alpha);
}

if (getenv("RED_BLOCK_THRESHOLD")) {
    red_block_threshold = strtod(getenv("RED_BLOCK_THRESHOLD"), NULL);
    printf(stderr, "Set red_block_threshold = %lf\n", red_block_threshold);
}

if (getenv("SCALE")) {
    scale = strtod(getenv("SCALE"), NULL);
    printf(stderr, "Set scale = %lf\n", scale);
}

if (getenv("INTERVAL")) {
    interval = strtod(getenv("INTERVAL"), NULL);
    printf(stderr, "Set interval = %lf\n", interval);
}

if (getenv("CONSEC_TOL")) {
    consec_tol = strtod(getenv("CONSEC_TOL"), NULL);
    printf(stderr, "Set consec_tol = %lf\n", consec_tol);
}

int disable_prefetch = 0;

/* Path to device for readahead. */
char *readahead_path = "/mnt/sdcl/tmp/span";

/* Path to run block trace on. */
char *trace_path = "/dev/loop0";
struct stat st;
int fd;
int rc;

if (argc > 1) {
    trace_path = argv[1];
}
APPENDIX B. PREFETCHD C SOURCE CODE

```c
if (argc > 2) {
    readahead_path = argv[2];
}
if (argc > 3) {
    disable_prefetch = strtol(argv[3], NULL, 0);
    if (disable_prefetch)
        fprintf(stderr, "Warning: prefetching disabled\n");
}
fd = open(readahead_path, O_RDONLY | O_NONBLOCK);
if (fd < 0) {
    perror("open");
    goto bad0;
}
rc = fstat(fd, &st);
if (rc < 0) {
    perror("fstat");
    goto bad1;
}
/* offset is signed on this system, so comparisons with 0 are
   meaningless! */
offset_blk_size = 512;
offset_max_block = st.st_size / blk_size;
struct rg_region red_green;
rg_region_init(&red_green, 1048576, st.st_size);
struct blk_watch bw;
struct timespec overall_start_time_tp;
if (clock_gettime(CLOCK_REALTIME, &overall_start_time_tp) < 0) {
    perror("clock_gettime");
}
unsigned long long overall_start_timestamp =
    overall_start_time_tp.tv_sec * 1000000000ull +
    overall_start_time_tp.tv_nsec;
double overall_start_time = timespec_to_double(overall_start_time_tp);
if (blkwatch_init(&bw, trace_path)) {
    perror("blkwatch_init");
    return 1;
}
signal(SIGINT, sighandler);
signal(SIGUSR1, sighandler);
signal(SIGUSR2, sighandler);
signal(SIGALRM, sighandler);
signal(SIGSEGV, sighandler);
signal(SIGBUS, sighandler);
int pref_event_cnt = 0, pref_blk_cnt = 0;
double think_start_time, think_end_time;
double sleep_time = interval;
double elapsed_time = 0;
pid_t our_pid = getpid();
fprintf(bw.fp_trace, "pid=%ld\n", our_pid);
fprintf(bw.fp_trace, "overall_start_time=%llu\n",
    overall_start_timestamp);
fflush(bw.fp_trace);
unsigned long long pref_read_bytes = 0;
size_t tot_blk_cnt = 0;
unsigned long long tot_recent_hit = 0;
unsigned long long tot_recent_miss = 0;
unsigned long long tot_cache_hit = 0;
unsigned long long tot_cache_miss = 0;
unsigned long long tot_false_pos = 0;
```

60
APPENDIX B. PREFETCHD C SOURCE CODE

unsigned long long read_counter = 0;
unsigned long long tot_prefetch_and_unused = 0;
unsigned long long tot_prefetch_and_used = 0;
unsigned long long tot_unprefetch_and_used = 0;
int tot_prefetch_enabled = 0;
#define HT_LEN (83)

struct prefetcher_state {
    pid_t pid;
    int event_cnt;
    int blk_cnt;
    off_t min_seen_block;
    off_t max_seen_block;
    off_t curr_block_lo;
    off_t curr_block_hi;
    off_t prev_block_lo;
    off_t prev_block_hi;
    int consec_blk_cnt;
    int reverse_blk_cnt;
    struct pred_linear pl;
    double app_throughput;
    int prefetch_enable;
    int blk_dir;
    off_t start_block;
    off_t stop_block;
    off_t blocks_on;
    off_t blocks_off;
    int curr_gap_dir;
    int curr_gap_req_len;
    off_t curr_gap;
    off_t prev_gap;
    int stride onBlur;
}
pf_table [HT_LEN];

int i;
for (i=0; i<HT_LEN; i++) {
    struct prefetcher_state *pf = &pf_table [i];
pf->pid = 0;
pf->event_cnt = 0;
pf->blk_cnt = 0;
pf->min_seen_block = max_block;
pf->max_seen_block = 0;
pf->curr_block_lo = 0;
pf->curr_block_hi = 0;
pf->prev_block_lo = 0;
pf->prev_block_hi = 0;
pf->consec_blk_cnt = 0;
pf->reverse_blk_cnt = 0;
pf->app_throughput = 0;
pf->prefetch_enable = 0;
pf->prev_gap = 0;
pf->stride onBlur = 0;
pf->read_bytes = 0;
pred_linear_init(&pf->pl);
}
APPENDIX B. PREFETCHC C SOURCE CODE

circ_init(&pf->prefetch_history,
MAX_PREFETCH_HISTORY,
sizeof(struct prefetch_operation));

pf->recent_miss = 0;
pf->recent_hit = 0;
pf->recent_prefetch_and_used = 0;
pf->recent_prefetch_and_unused = 0;
pf->recent_unprefetch_and_used = 0;
pf->cache_miss = 0;
pf->cache_hit = 0;
pf->false_pos = 0;
pf->curr_gap = 0;
}

printf(bw.fp_trace, "getpagesize=%d
", getpagesize());

off_t max_offset = 12000000000ull;
double cache_mem = 600e6;
int pagesize = 4096;
int blocks_per_page = pagesize / 512;

struct cache_state *sim_cache = cache_init(
max_offset / pagesize,
cache_mem / pagesize);

unsigned long long initial_timestamp = 0;

while (!exit_flag) {
    think_start_time = gettime_double();
    /* Pretend poll succeeded */
    for (i=0; i<bw.ncpus; i++) {
        bw.trace_fd[i].revents = POLLIN;
    }

    for (i=0; i<bw.ncpus; i++) {
        int unused_bytes = max_events * sizeof(struct blkio_trace);
        if (bw.trace_fd[i].revents & POLLIN || exit_flag) {
            char *dst = bw.read_buf[i] + bw.used_bytes[i];
            ssize_t rc = read(bw.trace_fd[i].fd,
                              dst,
                              unused_bytes);
            if (rc < 0 && errno != EAGAIN) {
                perror("read");
                exit_flag = 1;
            }
            /* Reads from this device always seem to return 0, so
             * this may not be needed.
             */
            if (rc < 0 && errno == EAGAIN) {
                rc = 0;
            }
            bw.used_bytes[i] += rc;
            unused_bytes -= rc;
            dst += rc;
            if (unused_bytes == 0) {
                printf(stderr,
"Event buffer overflow\n");
            }
        }
    }
}

double read_end_time = gettime_double();

printf(bw.fp_trace,
"trace_read_time_is=%lf\n", read_end_time - think_start_time);
APPENDIX B. PREFETCHD C SOURCE CODE

read_end_time - think_start_time);
fflush(bw.fp_trace);

/* Find events in each buffer */
bw.event_cnt = 0;
memset(bw.processed_bytes, 0, sizeof(bw.processed_bytes));
for (i=0; i<bw.ncpus; i++) {
    int used = bw.used_bytes[i];
    char *blk_c = bw.read_buf[i];
    while (blk_c < &bw.read_buf[i][used]) {
        struct blk_io_trace *blk = (struct blk_io_trace *) blk_c;
        blk_c += sizeof(struct blk_io_trace);
        if (blk_c > &bw.read_buf[i][used]) break;
        bw.processed_bytes[i] += sizeof(struct blk_io_trace);
        __u32 magic = blk->magic;
        if ((magic & 0xffffffff) != BLK_IO_TRACE_MAGIC) {
            fprintf(stderr, "Bad magic 0x%x\n", magic);
        }
        blk_c += blk->pdu_len;
        if (blk_c > &bw.read_buf[i][used]) break;
        bw.processed_bytes[i] += blk->pdu_len;
        if (initial_timestamp == 0) {
            initial_timestamp = blk->time;
            fprintf(bw.fp_trace, "Setting initial_timestamp to %llu\n",
                    initial_timestamp);
        }
        /* Convert length to block count. */
        blk->bytes /= blk->size;
        /* Filter as needed. */
        if (((blk->action & 0xffff) != BLK_TAQUEUE) continue;
        if (((blk->action & BLK_TC_ACT(BLK_TC_READ)) == 0) continue;
        /* Ignore pid 0 — kernel stuff */
        if (blk->pid == 0) continue;
        if (blk->pid == our_pid) {
            continue;
        }
        bw.event[bw.event_cnt] = blk;
        bw.event_cnt++;
    }
}
/* Sort by timestamp. */
qsort(bw.event,
     bw.event_cnt,
     sizeof(struct blk_io_trace *),
     event_cmp);
if (initial_timestamp == 0 && bw.event_cnt > 0) {
    initial_timestamp = bw.event[0]->time;
}
for (i=0; i<bw.event_cnt; i++) {
    struct blk_io_trace *ba = bw.event[i];
    ba->time -= initial_timestamp;
    if (bw.trace_enable) {
APPENDIX B. PREFETCHHD C SOURCE CODE

```c
fprintf(bw.fp.trace, 
    "Actu,%lu %lu %lu %lu %lu
", 
    ba->time, 1e-9, 
    ba->sector, 
    ba->bytes, 
    ba->pid 
    ba->sequence 
); 
fflush(bw.fp.trace);
}

pid_t hash = (ba->pid + (ba->sector / 4000000) ) * 16851 % HT_LEN;
struct prefetcher_state *pf = &pf_table[hash];
pf->pid = ba->pid;
/* Weight a multiple block request N times. */
int i;
for (i=0; i<ba->bytes / 8; i++) {
    pred_linear_point(&pf->pl, 
        ba->time + 1e-9, 
        ba->sector);
}
pf->event_cnt++; 
pf->blk_cnt += ba->bytes; 
tot_blk_cnt += ba->bytes; 
/* Compute cache hit or miss */
off_t page_start = ba->sector / blocks_per_page; 
int page_len = ceil(ba->bytes / blocks_per_page);
/* Find in history */
struct prefetch_operation *pp; 
off_t start_block = ba->sector; 
off_t n_blocks = ba->bytes; 
for (i=0; i<circ_cnt(&pf->prefetch_history); i++) {
    pp = circ_peek(&pf->prefetch_history, 
        i); 
    if (!pp) {
        break;
    }
    if (pp->t + history.time >= elapsed_time) {
        off_t overlap_start; 
        size_t overlapping_blocks; 
        off_t remain_start; 
        off_t remain_end; 
        reduce_overlap(pp, 
            start_block, 
            n_blocks, 
            &overlapping_blocks, &overlap_start, 
            &remain_start, &remain_end);
        if (overlapping_blocks > 0) {
            start_block = remain_start; 
            n_blocks = remain_end - remain_start + 1;
        }
    }
}
pf->recent_unprefetch_and_used += n_blocks; 
pf->recent_miss += n_blocks; 
pf->recent_hit += ba->bytes - n_blocks; 
if (ba->sector < pf->min_seen_block) {
    pf->min_seen_block = ba->sector;
}
```

64
APPENDIX B. PREFETCHD C SOURCE CODE

if ( (ba->sector > pf->max_seen_block) ) {
    pf->max_seen_block = ba->sector;
}

pf->prev_gap = pf->curr_gap;
pf->prev_block_lo = pf->curr_block_lo;
pf->prev_block_hi = pf->curr_block_hi;

if (pf->prev_block_hi == pf->curr_block_lo) {
    pf->consec_blk_cnt += ba->bytes;
pf->curr_consec_block_hi = pf->curr_block_hi;
}

if (pf->prev_block_lo == pf->curr_block_hi) {
    pf->reverse_blk_cnt += ba->bytes;
}

if (pf->prev_block_hi < pf->curr_block_lo) {
    pf->curr_gap = pf->curr_block_lo - pf->prev_block_hi;
pf->curr_gap_dir = 1;
pf->curr_gap_num = ba->bytes;
}

if (pf->prev_block_lo > pf->curr_block_hi) {
    pf->curr_gap = pf->curr_block_hi - pf->prev_block_lo;
pf->curr_gap_dir = -1;
pf->curr_gap_num = ba->bytes;
}

if (pf->curr_gap == pf->prev_gap) {
    /* Should we check the prev req len? */
    pf->strided_blk_cnt += ba->bytes;
}

sleep_time = interval;
elapsed_time = think_start_time - overall_start_time;
for (i=0; i<HTLEN; i++)
{
    struct prefetcher_state *pf = &pf_table[i];
    if (pf->event_cnt == 0) {
        continue;
    }

double r = pred_linear_score(&pf->pl);
double x_min = 0.; x_max = 0.;
if (pf->event_cnt > 0) {
    x_min = pf->pl.x_min;
    x_max = pf->pl.x_max;
}

pf->app_throughput = pf->blk_cnt * blk.size / (x_max - x_min);
double measured_prefetch_throughput = pf->blk_cnt * blk.size / interval;
double consec_pct = 0.;
double reverse_pct = 0.;
double strided_pct = 0.;
if (pf->event_cnt > 0)
    consec_pct = (double) pf->consec_blk_cnt / pf->blk_cnt;
if (pf->event_cnt > 0)
    reverse_pct = (double) pf->reverse_blk_cnt / pf->blk_cnt;
if (pf->event_cnt > 0)
    strided_pct = (double) pf->strided_blk_cnt / pf->blk_cnt;
int attempt_enable_prefetch = 0;
if (consec_pct > consec_tol ) {
    double prefetch_throughput = scale * pf->app_throughput;
    if (prefetch_throughput > max_throughput) {
        prefetch_throughput = max_throughput;
}
APPENDIX B. PREFETCHD C SOURCE CODE

```c
} // end of function

fprintf(bw.fp_trace, "prefetch_throughput, pf->app_throughput); pf->start_block = pf->curr_block_hi;
pf->stop_block = pf->curr_block_hi + (interval * prefetch_throughput / blk_size);

if (pf->start_block < pf->prev_end_block && (pf->prev_end_block - pf->start_block) * blk_size / pf->app_throughput < 4 * interval)
    pf->start_block = pf->prev_end_block + 1;
if (pf->stop_block < pf->start_block)
    pf->stop_block = pf->start_block + (interval * prefetch_throughput / blk_size);

attempt_enable.prefetch = 1;

if (attempt_enable.prefetch) {
    if (pf->start_block < 0) {
pf->start_block = 0;
    }
    if (pf->stop_block < 0) {
pf->stop_block = 0;
    }
    if (pf->start_block > max_block) {
pf->start_block = max_block;
    }
    if (pf->stop_block > max_block) {
pf->stop_block = max_block;
    }
    if (pf->start_block <= pf->stop_block) {
        pf->prefetch_enable = 1;
        tot_prefetch_enabled++;
    }
}

double lag = elapsed_time - x_min;
printf(bw.fp_trace, "Elp %2.2lf[\%d]%ld
%.3lf
elapsed_time, pf->pid, pf->prefetch_enable, pf->event_cnt, pref_event_cnt, x_min, x_max, 100. * pf->app_throughput / max_throughput, 100. * measured_prefetch_throughput / max_throughput, 100. * measured_throughput / max_throughput, 100. * consec_pct, 100. * reverse_pct, 100. * strided_pct, pf->min_seen_block, pf->max_seen_block, pf->curr_gap_dir, pf->curr_gap, lag
"flflush(bw.fp_trace);

double pct = 0.;
```

APPENDIX B. PREFETCHD C SOURCE CODE

if (pf->recent_hit + pf->recent_miss) {
    pct = pf->recent_hit
    / (double) (pf->recent_hit + pf->recent_miss);
}
printf(bw.fp_trace,
    "Recent hit/miss rate\n\npf->recent_hit, pf->recent_miss, pct\n"f);
fflush(bw.fp_trace);
pct = 0.;
if (pf->cache_hit + pf->cache_miss) {
    pct = pf->cache_hit
    / (double) (pf->cache_hit + pf->cache_miss);
}
printf(bw.fp_trace,
    "Cache hit/miss rate\npf->cache_hit, pf->cache_miss, pct\n"f);
fflush(bw.fp_trace);
if (pf->prefetch_enable) {
    off_t pref_blk_len = pf->stop_block - pf->start_block + 1;
    fprintf(bw.fp_trace,
        "Prefetchเหลือง blocksเหลือง total_for_pidเหลือง the_Block
pf->start_block, pf->stop_block, pref_blk_len, pf->pid,\n{pf->stop_block - pf->start_block} / interval\n"f);
    fflush(bw.fp_trace);
    ssize_t rc;
    double read_start = gettimeofday();
    double read_end = read_start;
    if (bw.trace_enable) {
        fprintf(bw.fp_trace,
            "Pref\n\nelapsed_time, pf->start_block, pref_blk_len\n"f);
        fprintf(bw.fp_trace,
            "Histเหลือง\n\ncirc.cent(&pf->prefetch_history)\n"f);
        fflush(bw.fp_trace);
    } // Add prefetched blocks to the cache
    /*
    off_t page_start = pf->start_block / blocks_per_page;
    int page_len = ceil(pref_blk_len / blocks_per_page);
    /* Remove stale history
    * starting from the oldest. *
    */
    fprintf(bw.fp_trace,
        "Purge_history\n"f);
    fflush(bw.fp_trace);
    struct prefetch_operation *pp;
    do {
        pp = circ.peek()
        &pf->prefetch_history, 0);
        if (!pp || pp->t + history_time
           >= elapsed_time)
struct prefetch_operation pop;
circ_deq(&pf->prefetch_history, &pop);

size_t used = prefetch_operation_get_used_blocks(&pop);
size_t unused = pop.n_blocks - used;

pf->recent_prefetch_and_used += used;
pf->recent_prefetch_and_unused += unused;

/* mark spatial prediction */
int j;
for (j=0; j<pop.n_blocks; j++) {
    int read = 0;
    if (pop.used_array[j]) {
        read = 1;
    }
    rg_region_predicted(kred_green, read,
                        (pop.start_block + j) * blk_size, blk_size);
}
free(pop.used_array);

while (pp);
printf(bw.fp_trace, "Done purging history\n");

printf(bw.fp_trace, "Recent_true_pos false_false_pos_false_pos\n", pf->recent_hit, pf->recent_miss, pf->false_pos);
fflush(bw.fp_trace);

if (prefetch_adaptive) {
    scale = adjust_aggressiveness(
        pf->recent_prefetch_and_used,
        pf->recent_prefetch_and_unused,
        pf->recent_unprefetch_and_used,
        scale,
        bw.fp_trace);
}

printf(bw.fp_trace, "adjust_scale to\n", scale);

struct prefetch_operation p;
p.t = elapsed_time;
p.start_block = pf->start_block;
p.n_blocks = pref_blk_len;
p.used_array = calloc(p.n_blocks, sizeof(unsigned char));
/*/ end points to just after prefetch op ends */
circ_enq(&pf->prefetch_history, &p);

int color = 0;
if (rg_pct(kred_green, ((pf->start_block + pf->stop_block) / 2) * blk_size) < red_block_threshold)
    color = 1;
printf(bw.fp_trace, "call_readahead\n");
fflush(bw.fp_trace);

if (disable_prefetch || color) {
break;
}
APPENDIX B. PREFETCHD C SOURCE CODE

```c
red_counter += pref_blk_len;
rc = 0;
} else {
  if (use_readahead) {
    rc = readahead(
      fd,     
      pf->start_block + blk_size, 
      pref_blk_len + blk_size);
  } else {
    rc = posix_fadvise(
      fd,     
      pf->start_block + blk_size, 
      pref_blk_len + blk_size, 
      POSIX_FADV_WILLNEED 
    );
  }
}

if (rc) {
  perror("readahead");
}
pref_event_cnt++;  
pref_blk_cnt += pref_blk_len;
read_end = gettime_double();
size_t n_bytes_read = 0;
  n_bytes_read += blk_size * pref_blk_len;
  fprintf(bw.fp_trace, 
    "Read time: %lf %g Bps\n", read_end - read_start,
    n_bytes_read / (read_end - read_start));
  flush(bw.fp_trace);
pf->prev_end_block = pf->stop_block;
pf->prefetch_enable = 0;
}
pf->read_bytes += pf->blk_cnt + blk_size;
pf->read_bytes += pref_blk_cnt + blk_size;
tot_recent_miss += pf->recent_miss;
tot_recent_hit += pf->recent_hit;
tot_cache_miss += pf->cache_miss;
tot_cache_hit += pf->cache_hit;
tot_false_pos += pf->false_pos;
tot_prefetch_and_used += pf->recent_prefetch_and_used;
tot_unprefetch_and_used += pf->recent_unprefetch_and_used;
pf->recent_miss = 0;
pf->recent_hit = 0;
pf->cache_miss = 0;
pf->cache_hit = 0;
pf->false_pos = 0;
pf->event_cnt = 0;
pf->blk_cnt = 0;
pf->event_cnt = 0;
pf->blk_cnt = 0;
pred_linear_init(&pf->pl);
pf->consec_blk_cnt = 0;
pf->reverse_blk_cnt = 0;
pf->strided_blk_cnt = 0;
pf->min_seen_block = max_block;
pf->max_seen_block = 0;
}
think_end_time = gettime_double();
tot_prefetch_enabled = 0;
```

APPENDIX B. PREFETCHD C SOURCE CODE

sleep_time = think_end_time - think_start_time;

printf(bw_fp_trace ,
"Thinkook\%lf now, sleep %lf\n", 
think_end_time - think_start_time , 
sleep_time); 
flush(bw_fp_trace);

/* Reset counters */
bw_event_cnt = 0;
for (i = 0; i < bw.ncpus; i++) {
    bw_used_bytes[i] = 0;
}

ssize_t poll_rc = -1;

struct timespec timeout;

if (sleep_time < 0.) {
    timeout.tv_sec = 0;
    timeout.tv_nsec = 0;
} else {
    timeout = double_to_timespec(sleep_time);
}

if (!exit_flag 
    && (poll_rc = ppoll(bw.trace_fd , 
bw.ncpus, 
&timeout , 
NULL /* sigmask */
)) < 0 
    && errno != EINTR)
{
    perror("poll");
    exit_flag = 1;
}

double pct = 0.;

if (tot_recent_hit + tot_recent_miss) {
    pct = tot_recent_hit 
        / (double) (tot_recent_hit + tot_recent_miss);
}

printf(bw_fp_trace ,
"Recent %llu hits %llu misses %llu pct (%llu)\n", 
tot_recent_hit , 
tot_recent_miss , 
pct , 
tot_false_pos 
);

if (tot_recent_hit + tot_recent_miss) {
    pct = tot_recent_hit 
        / (double) (tot_recent_hit + tot_recent_miss);
}

printf(bw_fp_trace ,
"Recent %llu pref and used %llu pref and unused %llu unpref and used\n", 
tot_prefetch_and_used , 
tot_prefetch_and_unused , 
tot_unprefetch_and_used 
);

printf(bw_fp_trace ,
"prefetch_and_used / tot_used = %llu\n", 
(double) tot_prefetch_and_used / (tot_blk_cnt) ,
(double) tot_prefetch_and_used / (tot_prefetch_and_used + tot_unprefetch_and_unused) 
);

cache.clear(sim_cache);
pct = 0.;

if (tot_cache_hit + tot_cache_miss) {
    pct = tot_cache_hit 
        / (double) (tot_cache_hit + tot_cache_miss);
}

printf(bw_fp_trace ,
"Cache %llu hits %llu misses %llu pct\n", 
tot_cache_hit , 

APPENDIX B. PREFETCHD C SOURCE CODE

```c

    tot_cache_miss, pct); 
    
    fprintf(bw.fp_trace, 
        "true_pos \{\text{prefetched and used}\}: %llu\n" 
        "false_neg \{\text{not prefetched and used}\}: %llu\n" 
        "false_pos \{\text{prefetched and not used}\}: %llu\n", 
        sim_cache->true_pos, 
        sim_cache->false_neg, 
        sim_cache->false_pos); 
    
    fprintf(bw.fp_trace, 
        "Read \{\text{prefetch bytes}\}: \%3.0fMB\n", 
        tot_blk_cnt * blk_size, 
        le-6 * tot_blk_cnt * blk_size / elapsed_time); 
    
    fprintf(bw.fp_trace, 
        "Read \{\text{prefetch bytes}\}: \%3.0fMB\n", 
        pref_read_bytes, 
        le-6 * pref_read_bytes / elapsed_time); 
    
    fflush(bw.fp_trace); 
    fprintf(stderr, "prefetchd closing trace device\n"); 
    fflush(bw.fp_trace); 
    cache_free(sim_cache); 
    blkwatch_close(&bw); 
    close(fd); 
    fprintf(stderr, "prefetchd_exit\n"); 
    return 0; 
}

bad1: 
    close(fd); 
bad0: 
    return 1; 
```