An Adaptive IO Prefetching Approach for Virtualized Data Centers

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

Abstract—Cloud and data center applications often make heavy use of virtualized servers, where flash-based solid-state drives (SSDs) have become popular alternatives over hard drives for data-intensive applications. Traditional data prefetching focuses on applications running on bare metal systems using hard drives. In contrast, virtualized systems using SSDs present different challenges for data prefetching. Most existing prefetching techniques, if applied unchanged in such environments, are likely to fail to fully utilize SSDs, interfere with virtual machine I/O requests, or cause too much overhead if run in every virtualized instance. In this work, we demonstrate that data prefetching, when run in a virtualization-friendly manner can provide significant performance benefits for a wide range of data-intensive applications. We have designed and developed VIO-prefetching, consisting of accurate prediction of application needs in runtime and adaptive feedback-directed prefetching that scales with application needs, while being considerate to underlying storage devices and host systems. We have implemented a real system in Linux and evaluated it on different storage devices with the virtualization layer. Our comprehensive study provides insights of VIO-prefetching’s behavior at various virtualization system configurations, e.g., the number of VMs, in-guest processes, application types, etc. The proposed method improves virtual I/O performance up to 43% with the average of 14% for 1 to 12 VMs while running various applications on a Xen virtualization system.

Index Terms—Data storage systems, platform virtualization, operating systems.

1 INTRODUCTION

Cloud service providers have adapted flash-based solid-state drives (SSDs) in data centers for high throughput and low energy consumption [1], [8]. For example, Amazon is using SSDs for the DynamoDB application in Amazon Web Services (AWS) that offers virtual machines (VMs), virtual storages and computing services [29]. Data prefetching is one of, if not the most, widely-used techniques to reduce access latency, because it can load data that are likely to soon be accessed from storage devices into main memory [14], [33]. Future data centers certainly need novel prefetching technologies to meet the needs for virtualization and new storage devices. In this paper, we aim to achieve performance benefits in a virtualized environment. More specifically, our goal is to serve data-intensive applications in VMs with better performance via virtual I/O prefetching.

Cloud service providers heavily rely on virtualization technology to provide flexible task management and efficient resource utilization. Nonetheless, such advantage comes with the cost of I/O performance. Operating systems are traditionally designed to optimize disk I/O under an assumption that they hold the exclusive control over storage drives. Nowadays, such assumption of exclusivity does not exist anymore because virtualization puts operating systems into guest VMs and hosts many VMs on one physical machine. In addition, storage devices are now shared among numerous guest VMs. Unfortunately, the I/O stacks inside guest VMs still try to optimize the sequential I/O patterns on an imaginarily exclusively owned disk. By the time the I/O requests hit the physical storage devices, they are not sequential anymore. This effect is called the virtual I/O blurring [36]. The more VMs involved in the blending, the more obvious the effect. These blurred I/O patterns are not just because of multiple concurrent I/O processes, but also the lack of understanding between the guest VMs and the host environment.

Besides the new challenges introduced by virtualization, emerging SSDs in data centers also significantly affect the design of a new prefetching method. Traditional prefetching focuses on rotational hard drives and is conservative with the amount of data prefetched for good reasons—because data prefetching consumes shared system resources. It is likely that aggressive data prefetching would interfere with normal access and subsequently hinder application performance. As a result, current techniques often leverage the low cost of sequential access on hard drives to read data that reside on the same and nearby tracks. Aggressive prefetching has been considered too risky by many researchers (given long seek penalties, limited HDD bandwidth, and limited system RAM), with one notable exception [32].

For SSDs, we believe that aggressive prefetching could potentially expedite data requests for many applications. However, as we will demonstrate shortly, simply prefetching as much data as possible does not provide the desired benefits for several 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 on slower hard drives).
As a side effect, useful cached data may be evicted while main memory would be filled with mispredicted (and unneeded) data and applications were waiting for useful data [14]. Second, not every device has the same performance characteristics, and this is especially true for SSDs. The performance of an SSD can vary depending on the flash type (SLC/MLC), internal organization, memory management, etc. A prefetching algorithm, while reasonably aggressive for a faster drive, could become too aggressive for another drive and hinder applications’ performance.

Our work [41] has shown that adaptive prefetching techniques can be used on SSDs to avoid adverse effects from both too-conservative and too-aggressive prefetching. In this paper, we propose a technique called VIO-prefetching for virtualized servers. VIO-prefetching is aware of the runtime environment and can adapt to the changing requirements of devices and applications. The noticeable features of VIO-prefetching include not only feedback-controlled aggressiveness, but also the ability to identify I/O access patterns and providing inherent prefetching support for virtual I/O in data centers, which presents a good extension to [41].

To demonstrate its feasibility and benefits, we have implemented a prototype in Linux that dynamically controls its prefetching aggressiveness at runtime to maximize performance benefits, by making good trade-offs between data prefetching and resource consumption. We evaluate VIO-prefetching with a wide range of data-intensive cloud applications on a Xen virtualized system. Evaluation results show that VIO-prefetching can improve virtual I/O performance up to 43%.

The main contributions of this paper are:

- VIO-prefetching tunes itself to prefetch data in a manner that matches application needs without being so aggressive that useful pages are evicted from the cache. By measuring performance metrics in real-time and adjusting the aggressiveness accordingly, the effectiveness of VIO-prefetching is significantly improved.

- VIO-prefetching is able to identify I/O access patterns from guest VMs and successfully improve virtual I/O performance. Our comprehensive study provides insights of VIO-prefetching’s behavior at various virtualization system configurations.

- We conduct a comprehensive study of the effects of VIO-prefetching in the context of heterogeneous applications. The results show that VIO-prefetching is essential for identifying application patterns and prefetching needed data in virtualized environments.

The rest of the paper is organized as follows: Section 2 presents the challenges of building the VIO-prefetching and Section 3 presents the architecture of VIO-prefetching and describes each component. The evaluation is presented in Section 4 and related works are discussed in Section 5. We conclude in Section 6.

2 CHALLENGES

2.1 Challenge #1: No One-size-fits-all

Migrating VMs among servers is a very common practice. However, there is no one-size-fits-all solution for data prefetching because many things could be different from one server to another. Here we focus on three aspects: storage devices, applications, and prefetching techniques.

First, storage devices are different. SSDs are clearly different from HDDs in many ways. To name a few: no seek latency, excellent random read and write performance, inherent support for parallel I/O, expensive small writes, and limited erase cycles. At a high level, modern SSDs consist of several components such as NAND flash packages, controllers, and buffers. In our previous study [41], we tested four different SSDs from two manufacturers (roughly covering two recent generations of SSDs): OCZ Vertex and Vertex2, and Intel X-25M and 510. We compared their performance with a Samsung Spinpoint M7 (HDD) hard drive. If we look at the device specifications, the specification numbers for SSDs are close. However, the differences between different SSDs tend to be subtle, mostly in architectural designs. When measured under Linux, the four SSDs clearly have higher bandwidths than the hard drive (measured read bandwidth at about 90 MB/s), that is, the four SSDs outperform the hard drive by 189%, 189%, 294%, and 239%, respectively. The four SSDs differ noticeably, especially in write performance. Their measured write bandwidths range from 80 MB/s to 200 MB/s.

Second, applications are different. Although data-intensive applications are in dire need of high-performance data access, they tend to have different I/O requirements. When we look at the average application throughput in I/O operations per second (IOPS) for applications ranging from large file operations, web server traces, and video streaming workloads as shown in our preliminary study [41], the two replayed WebSearch traces reach the highest throughput at about 6,000 IOPS, and 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.

Third, prefetching for HDDs and SSDs is different. Traditional disk drives can read sequential blocks quickly because the head can be stationary while the platter rotates underneath. A random read operation from flash can be completed quickly in a few microseconds, compared to several milliseconds seek latencies on hard drives. In addition, multiple simultaneous requests for data on one SSD that address different flash chips can be satisfied simultaneously—a challenging task for a hard disk. The internal controllers of SSDs have already taken advantage of this inherent parallelism for high performance I/O [1], and it has been shown that this parallelism can also be exploited at a higher system level.

To clearly explain the parallelism in HDD and SSD,
suppose that two applications simultaneously issue sequential read requests to a hard disk; such patterns are likely to interfere with each other. To satisfy the simultaneous requests, the access patterns must occur on different platters, otherwise the disk heads might move back and forth to different tracks. An I/O scheduler will try to minimize head movements, but this movement overhead still limits the number of simultaneous prefetch operations that can occur on a traditional hard drive. In contrast, parallel I/Os in SSDs can benefit greatly from better hardware structure and organization.

2.2 Challenge #2: Virtual I/O Semantic Gaps

Virtualization is widely used among cloud service providers because virtualization technology provides a number of advantages, such as flexible task management, efficient resource utilization, etc. But, it comes at the cost of I/O performance. Vanilla operating systems such as Linux assume an exclusive control of storage drives and optimize disk I/O under that assumption. Virtualization puts operating systems into guest VMs and hosts many VMs on one physical machine. Storage devices are now shared among numerous guest VMs. So, that assumption of exclusivity is no longer valid. However, the I/O stacks inside each guest VM still try to optimize the I/O patterns for sequential access on the virtual disk. When these I/O requests are forwarded to the hypervisor, they will be likely blended in an unpredictable fashion. By the time the I/O requests arrive the physical storage devices, they are no longer sequential [36]. This is called the virtual I/O blending. The more VMs involved in the blending, the more obvious the effect.

Fig. 1(a) shows the virtual I/O blending effect on overall performance. It compares the combined throughput of sequential read operations on one HDD running multiple guest VMs, each identically configured with a benchmark tool called IOZONE [31]. The performance of a single VM is almost identical to that of a non-virtualized server. As we add more VMs, the combined throughput of the HDD decreases dramatically. At eight VMs, the throughput is about half that of a single VM. An SSD does mitigate the situation. Fig. 1(b) presents the same tests on an SSD. The combined throughput reaches the maximum bandwidth at four VMs. Then, the performance goes down from there because the test is running on a four-core machine. However, the guest systems are not aware of this competition and the host system does not know the I/O access patterns inside each guest VMs. Knowing guest I/O process information can help prefetching methods more effectively capture targeting patterns. Therefore, we implement a virtualized system which passes through guest I/O process identifications to the prefetcher in the host system for more accurately identifying potential patterns. Section 4 demonstrates that our VIO-prefetching can successfully prefetch needed data by identifying I/O access patterns and maintain high throughput with feedback control.

3 The Architecture of VIO-prefetching

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

We design an architecture of VIO-prefetching which implements the four stages in a virtualization environment. Fig. 2(a) depicts the I/O path in a virtualized server integrated with VIO-prefetching. Guest VMs use the front-end drivers to talk to the backend drivers in the driver domain, and the backend drivers utilize real device drivers to access the physical devices. To complete a virtual I/O operation through the communication between these drivers, the I/O channel also needs to map memory pages and translate the addresses. These operations are part of the virtualization overheads. Because of these complex operations and the semantic gaps among domains, the I/O access patterns are hard to identify when arriving at physical storage devices. VIO-prefetching provides a remedy to this problem by bridging information gaps among domains and identifying access patterns at the block device level. The in-guest process identifications are passed to the host domain for the pattern recognition module of VIO-prefetching. Then, VIO-prefetch groups and prefetches needed data by exploiting the underutilized bandwidth on physical devices.

In a virtualized environment, each VM has its own view of a disk, called the virtual machine disk (VMDK). VMDK could be an image file on the host machine's file system, a disk partition, or a physical block device on the host machine. We choose to integrate VIO-prefetching with virtualized host systems, not guest VMs for the following reasons:

- There are many I/O layers between a guest VM and the underlying physical devices. When a sequential prefetching from a guest VM arrives at underlying physical devices, it may be no longer sequential. As a result, prefetching from guest VMs may have limited benefit.
- VIO-prefetching utilizes underutilized bandwidth...
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2469295, IEEE Transactions on Services Computing

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

In the current implementation, an application execution is identified by using a combination of process id, in-guest process id, drive id and block region. Note that original blktrace API did not have a field for in-guest process id, which is supported in this work.

In order to collect I/O event traces from all VMs to corresponding VMDKs, all VMDKs are stored in one logic drive. This is a common practice to manage storage systems and can be achieved by utilizing Logical Volume Manager (LVM), loopback devices, or RAIDs. After placing VMDKs in one single logical drive, VIO-prefetching monitors the logic drive for virtual I/O event traces. Fig. 2(b) shows how VIO-prefetching is integrated with a virtualized host. First, the trace collection module records every I/O request. Note that not every request by VMs will actually reach its VMDK because some of them may be satisfied by the system cache, but VIO-prefetching traces both issued VM requests and those that actually reach the disk. Then, the pattern recognizer wakes up to look at the accumulated I/O events when a timer expires. The pattern recognizer then informs the controller whether, where, and how many to prefetch. The controller optionally adjusts the aggressiveness based on recent prefetching performance. The details of these four stages are described individually in the following sections.

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

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

### 3.2 Pattern Recognition

The design of VIO-prefetching for virtualized environments considers the access patterns from VMs and on storage devices. Internally, pattern recognition of VIO-prefetching is designed around the idea of a polling interval. When a timer expires, VIO-prefetching wakes up, looks at the accumulated I/O events, decides whether, where, and how much to prefetch, performs the prefetch request, optionally adjusts its aggressiveness based on recent prefetching performance, and sleeps for the remainder of the interval. The polling interval determines how long events accumulate in the I/O request buffer before VIO-prefetching analyzes them. We have tested 0.25, 0.5, 0.75, ..., and 2 seconds as the polling interval and found no obvious differences from 0.25 to 0.75. When the interval is larger than or equal to 1 second, the prefetching accuracy is decreased unless the application has a very large sequential pattern. Therefore, we choose 0.5 seconds as the polling interval.

A single I/O event contains several pieces of information, but VIO-prefetching is primarily interested in read requests, the starting block number, number of blocks in the request, and the process identification making the request. If a particular process makes a recognizable pattern of read accesses within a specific period of time, VIO-prefetching begins to prefetch the same pattern. Currently, VIO-prefetching recognizes four major types of accesses: sequential forward reads, sequential backward reads, strided forward reads, and strided backward reads. In this discussion a strided pattern is simply a recurring pattern with a number of blocks read and a gap where no blocks are read.

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

When the fraction of blocks that occurred in consecutive, reverse, or strided requests divided by the overall count of blocks read exceeds a certain threshold over the previous time interval, the state machine for that hash entry is ready to perform a prefetch during the remainder of the current time interval. Sequential threshold \( T_{\text{SEQ}} \) determines the percentage of the virtual disk blocks must fit a usable pattern (sequential, reverse, or strided) before VIO-prefetching attempts to start prefetching. For example, the default value of 0.60 indicates that if 60 percent of the requests during a polling interval are sequential, VIO-prefetching guesses that a sequential access is occurring and will fetch a sequential series of blocks for the next interval. We choose the default threshold based on the data from our testing applications and machines. The qualified intervals and their consecutive followers provide an average of 84% usable blocks, while the threshold of 0.5 gives 73% and 0.7 gives 85%. The selection of the threshold mainly depends on the access pattern distribution of the system. It would be an interesting future work to dynamically adapts the threshold based on observed pattern distribution. When VIO-prefetching begins prefetching on behalf of a VM, it simply begins with the next block contiguous to the most recent request. Next section will demonstrate how the stop block is set by extrapolating into the future from the end of the last read operation.

### 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 an SSD into the system cache, but only those blocks that the application will actually request in the near future. For simplicity, we describe the logic for consecutive prefetching. The logic for strided and reverse prefetching is similar. In VIO-prefetching, we utilize two key parameters that control the amount of data that will be prefetched:

**Aggressiveness scale factor** \( S \) is defined as

\[
S = \frac{\text{prefetched data amount}}{\text{application read data amount}}
\]

which means how aggressive prefetching is compared to an application’s measured request rate. While we can measure an application’s rate to tailor prefetching based on the application’s needs, we have found that using a fixed, static scale factor does not work well.
The optimal value for this scale factor is application-specific and can be adjusted by feedback (which will be described in the next section). Our experiments showed that the values near 1.0 typically work well as the starting point for the feedback mechanism. A value of 1.0 means that the amount of prefetched data matched the application’s request rate. If the value is higher than one, VIO-prefetching might prefetch some unneeded data. On the other hand, if the value is less than one, some requests may not be satisfied by the prefetched data.

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

Since the maximum disk throughput depends on the characteristics of each drive, we measure the raw throughput from each disk by reading a large, uncached file, and using this as the maximum. In a virtualized environment, guest VMs do not know the utilization status of physical drives. Therefore, VIO-prefetching helps each VM to prefetch data with device characteristics in mind.

Putting these two parameters together, the prefetcher uses the last known (read) stop block as its start block (\(B_{\text{start}}\)) and finds the stop block as follows. It first tries to determine the linear throughput of the application (\(TP_L\)) by multiplying the total throughput (\(TP\)) with the percentage of consecutive reads (\(PCT_{\text{SEQ}}\)). We consider the remainder of the total application throughput to be from random accesses (\(TP_R\)). Next, the prefetcher uses the scale factor \(S\) and total available bandwidth \(BW_A\) (by subtracting \(TP_R\) from the maximum disk throughput \(BW_T\)) to determine the stop block (\(B_{\text{stop}}\)) for the next polling interval. For quick reference, a summary of all the terms used is listed in Table 1.

Suppose the polling interval is \(T\) seconds, the calculation to find \(B_{\text{stop}}\) shows as follows:

\[
TP_L = TP \times PCT_{\text{SEQ}} \\
TP_R = TP \times (1 - PCT_{\text{SEQ}}) \\
BW_A = BW_T - TP_R \\
PF_{\text{limit}} = \min(S \times TP_L, BW_A) \\
B_{\text{stop}} = B_{\text{start}} + T \times PF_{\text{limit}}
\]

Once the quota of number of blocks to prefetch for one application during an interval is found, VIO-prefetching simply issues a system call (e.g., readahead in Linux) with the starting block number and the number of blocks to read. (For strided access, there may be multiple readahead calls.) We leave the details of the cache management itself to the underlying operating system.

### TABLE 1: Variables in the formulas and the pseudocode

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_{\text{SEQ}})</td>
<td>The threshold on linear operations to start prefetching</td>
</tr>
<tr>
<td>(B_{\text{start}})</td>
<td>The block to start prefetching</td>
</tr>
<tr>
<td>(B_{\text{stop}})</td>
<td>The block to stop prefetching</td>
</tr>
<tr>
<td>(TP)</td>
<td>Total throughput</td>
</tr>
<tr>
<td>(TP_L)</td>
<td>Throughput by linear access</td>
</tr>
<tr>
<td>(TP_R)</td>
<td>Throughput by random access</td>
</tr>
<tr>
<td>(PCT_{\text{SEQ}})</td>
<td>Percentage of sequential reads</td>
</tr>
<tr>
<td>(BW_A)</td>
<td>Available bandwidth</td>
</tr>
<tr>
<td>(BW_T)</td>
<td>The maximum disk throughput</td>
</tr>
</tbody>
</table>

### Algorithm 1: The Pseudocode for VIO-prefetching

```
Data: Disk read event \(\text{ReadDisk}\); Requested read event \(\text{ReqRead}\);
Number of consecutive blocks \(B_{\text{SEQ}}\);
Number of total blocks \(B_{\text{Total}}\);

begin
  for every \(T\) seconds do
    // Collect read operations that reached the physical disk (i.e., not satisfied by cache)
    for each \(\text{ReadDisk}\) do
      Update per-disk counters
      Update \(S, TP_L\), and \(TP_R\);
    end
    // Collect read operations that are partially satisfied by cache
    for each \(\text{ReqRead}\) do
      // Track concurrent multiple read
      \(h = \text{Hash(process id, in-guest id, } B_{\text{start}})\);
      Update the counters of the mapped state machine;
    end
    for each state machine \(h\) do
      \(PCT_{\text{SEQ}} = h.B_{\text{SEQ}} / h.B_{\text{Total}}\);
      if \(PCT_{\text{SEQ}} > T_{\text{SEQ}}\) then
        // Set a prefetching throughput limit
        \(PF_{\text{limit}} = \min(S \times TP_L, BW_A)\);
        // Begin prefetching
        \(B_{\text{stop}} = B_{\text{start}} + T \times PF_{\text{limit}}\);
      end
    end
  end
end
```

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
3.4 Feedback Monitoring

Feedback monitoring classifies the stream of read operations reaching disk as linear (meaning sequential, reverse, and strided) similar to the way read requests to the operating system were classified during pattern recognition. The intuition is that if there are any linear, easily predictable reads that were not prefetched, and still reached disk, then the prefetching aggressiveness should be increased. On the other hand, if there are no linear reads reaching the disk and the statistics show that the prefetching amount is more than what the applications are requesting, we decrease the aggressiveness accordingly.

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

During each polling interval, the feedback monitoring analyzes the actual performance of the prefetch operations from the last time interval and adjusts its aggressiveness accordingly. This monitoring is done by comparing the access pattern of reads that the application makes to the operating system (entering the cache) vs. the pattern of reads reaching disk (missed in the cache). Algorithm 1 presents the high-level pseudocode of VIO-prefetching.

4 Evaluation

We have implemented a prototype VIO-prefetching in a Linux system with Xen virtualization. To assist prefetching in the host domain, VIO-prefetching passes guest I/O requests’ owner process identifications to the driver domain by embedding requests’ owner identifications when generating a Xen blkfront I/O request and extracting them when transforming a Xen block request into a normal one. In the following sections, we will first introduce the applications and environments used for the experiments. Then, we explain the experiments and results.

4.1 Experiment Environment and Applications

High-performance storage systems are needed in many different types of data-intensive applications. To evaluate the performance of VIO-prefetching technique, we choose a wide variety of benchmarks, including numerous cloud applications and file system benchmarks. Table 2 shows a number of popular cloud applications and their configurations used in our experiments. In brief, Cloudstone is a performance measurement framework for Web 2.0 [39]; Wiki with Database dumps is from Wikimedia foundation [46] and the real request traces are from the Wikibench web site [42]; Darwin is an open source version of Apple’s QuickTime video streaming server; FS, WS, VS, and WP are file, web, video, and web proxy servers respectively, which are all from Filebench [28]. All applications are running in eight concurrent threads/workers unless elsewhere specified.

The test system has two six-core Intel Xeon CPUs at 2 GHz and 32 GB memory. This machine is running Linux 3.2, Xen 4.1, and eight 120 GB SSDs configured as RAID0 on a MegaRAID card. Each storage device is formatted with an ext2 file system, 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 file system and also mounted with the noatime option for running the benchmarks. The noatime option prevents read operations to the file system from generating metadata updates which would require writes to the device and is intended to improve the I/O throughput.

4.2 Interval and Overhead Analyses

When a timer expires, VIO-prefetching parses the accumulated I/O events, decides whether, where, and how much to prefetch, performs the prefetch request, optionally adjusts its aggressiveness based on recent prefetching performance, and sleeps for the remainder of the interval. The polling interval determines how long events accumulate in the I/O request buffer before VIO-prefetching analyzes them. A long interval may reduce the overhead of invoking excessive times of pattern recognition module, but an access pattern may already be over. In addition, a long interval means more records to be processed than a short one. However, if the interval is too short, there may not be enough accumulated events to discern a pattern. Therefore, we want the interval to be as small as possible while keeping reasonable computing overheads.

The 0.5 seconds interval is used as the default in this prototype because it performs better than other tested values. Fig. 3 demonstrates part of these tests. In Fig. 3, each VM is sequentially reading a 10 GB file with an 1 MB IO size. The lines in Fig. 3 demonstrate the latency changes at varying polling intervals. The four lines are the test results when 2, 4, 8, and 16 VMs concurrently running, respectively. The Y-axis in Fig. 3 represents the average latency of reading 1 MB and the X-axis is the polling interval. In Fig. 3, latency is decreasing as the interval increases from 0.25 to 0.5 because of the increased accuracy and less interrupts. The differences are more obvious when there are more VMs because there are more IO operations in the queue and more events for VIO-prefetching to process. The latency doesn’t keep decreasing when the interval is larger than 0.5 seconds partly because of the processing overhead of VIO-prefetching as it is shown in Fig. 4.

Fig. 4 shows the average processing time of one VIO-prefetching event. A longer polling interval means less
virtualized environments by bridging the missed process
identification among domains. To see if this change
makes VIO-prefetching better than flashy prefetching in
a virtualized environment, we start the evaluation sec-
tion with the comparisons between these two prefetching
schemes.

We firstly want to see if the VIO-prefetching performs
as flashy prefetching when there is only one major I/O
process in guest VMs. Therefore, in the first experiment,
there is only one major process which is sequentially
reading 64 KB from a 1 GB file in a guest VM. The
VM has one VCPU and 512 MB memory. The baseline
is running this sequential read process in a guest VM
with the default readahead setting in Linux. Then, we
turn off the default Linux readahead function and run
the same sequential read process with the help of flashy
and VIO-prefetching respectively. The above experiment
is repeated three times at 1, 3, 6, 9, and 12-VM cases
respectively. The average speedups and standard de-
viations are reported in Fig. 5(a). The speedup is ob-
tained by normalizing the measured throughput to the
baseline. Flashy and VIO-prefetching perform closely in
this experiment because there is only one major I/O
threads in guest VMs. Both flashy and VIO-prefetching
have better aggregate throughputs than the baseline. The
average speedups by flashy and VIO-prefetching are 1.19
and 1.2 respectively. The peak speedup of flashy and
VIO-prefetching schemes appear at 6-VM case with the
value of 1.35 and 1.3 respectively. The reason of the
reduced speedup at large numbers of VM is because of
the saturated bandwidth. As the number of sequential
reading process increases, the available bandwidth for
prefetching decreases and thus limits the benefit.

The second test is to verify if passing in-guest pro-
cess identification can help prefetching in virtualized
environments. To validate this, there are multiple bench-
marking processes in a guest VM. More specifically, we
have 1, 3, 6, 9, and 12-process cases. Each process is se-
quentially reading 64 KB from a 1 GB file. Note that each
process has its own file. There is only one VM in this test,
which has 12 VCPUs and 6 GB memory. We measure the
aggregate throughputs of these processes when using the
baseline, flashy, and VIO-prefetching. Then, the speedup
is obtained by normalizing the measured throughput to
the baseline. The tests are repeated three times and the
average speedups and standard deviations are drawn
as the columns and whiskers in Fig. 5(b). When there is

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Workload Types</th>
<th>Data size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloudstone</td>
<td>Social event application (Olio)</td>
<td>Interactive update and read</td>
<td>20 GB</td>
</tr>
<tr>
<td>Wiki</td>
<td>Wikimedia website</td>
<td>Similar to but less updates than WS</td>
<td>60 GB</td>
</tr>
<tr>
<td>Darwin</td>
<td>Video streaming</td>
<td>Heavy sequential read operations</td>
<td>36 GB</td>
</tr>
<tr>
<td>FS</td>
<td>File server</td>
<td>create, delete, read and write files</td>
<td>20 GB</td>
</tr>
<tr>
<td>WS</td>
<td>Web server</td>
<td>open, read, and close multiple files</td>
<td>20 GB</td>
</tr>
<tr>
<td>VS</td>
<td>Video server</td>
<td>sequential read and write</td>
<td>20 GB</td>
</tr>
<tr>
<td>WP</td>
<td>Web proxy server</td>
<td>open, create, read, write, and close files</td>
<td>20 GB</td>
</tr>
</tbody>
</table>

![Fig. 3: The average latency on different polling intervals](image)

![Fig. 4: The average processing time of one VIO-prefetching event](image)

4.3 VIO-Prefetching vs. Flashy Prefetching

Flashy prefetching, the preliminary work of VIO-
prefetching, has shown its ability to read ahead just be-
fore needed blocks are accessed by the application [41].
Preliminary results also indicate that flashy prefetching
is capable of adapting the speed of data prefetching
on the fly to match the needs of the application. In
this work, VIO-prefetching adapts flashy prefetching for
virtualized environments by bridging the missed process

Note that the optimal polling interval depends on
many system characteristics such as the access behavior
and the efficiency of the pattern recognition. As future
work, the next VIO-prefetching will explore the self-
tuning intervals and thresholds based on the perfor-
mance feedback.

![Fig. 5(a): The average speedup of VIO-prefetching in a virtualized environment.](image)

![Fig. 5(b): The average speedup of VIO-prefetching in a virtualized environment.](image)
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2469295, IEEE Transactions on Services Computing

CloudStone, FS, and WP demonstrate good performance because there is only one major I/O thread in each guest VM. In all cases, both flashy and VIO-prefetching perform closely in this experiment because there is only one major I/O thread in each guest VM. There are five cases in this test. The first case is that one VM is doing sequential read and the other VMs are doing random read and write. The number of sequential-read VMs is changed to 3, 6, 9, and 12 in the second to fifth cases respectively. In all cases, the other VMs other than the sequential-read one are doing random read/write 64 KB from/to 1 GB files with the 50:50 read:write ratio. We measure the aggregate throughputs of the sequential-read VMs when using the baseline, flashy, and VIO-prefetching. Then, the speedup is calculated by normalizing the measured throughput to the baseline. The tests are repeated three times and the average speedups and standard deviations are shown as the columns and whiskers in Fig. 5(c). When there is only one process, the speedups by flashy (1.08) and VIO-prefetching (1.10) are close. In all cases, VIO-prefetching has higher speedups than flashy. On average, the speedup of VIO-prefetching (1.13) is higher than the flashy’s (1.06) by 0.07. Because of the extra random I/O processes, the speedups in Fig. 5(d) is less than those in Fig. 5(b).

![Fig. 5: The speedups and standard deviations of three prefetching systems.](image)

Fig. 5(d) is less than those in Fig. 5(b).

### 4.4 Evaluation with Cloud Applications

After comparing VIO-prefetching with other prefetching methods in a virtualized environment, we now evaluate VIO-prefetching with numerous cloud applications and file system benchmarks in this section.

**Experiments on Different VM Numbers.** The experimental environment and applications are described in Section 4.1. The goal of this test is to see how VIO-prefetching works at different applications and numbers of VMs. We have tested 1, 2, 4, ⋯, and 12 VMs, while each VM has 1 VCPU and 1 GB memory.

![Fig. 6: The speedups by VIO-prefetching for different applications and numbers of VMs](image)

![Fig. 6](image)

Fig. 6 shows the speedups of the applications for different numbers of co-located VMs. At each run, all co-located VMs are executing the same application. The results are the average numbers of three runs. The overall mean of the speedups in Fig. 6 is 1.14. As shown in Fig. 6, WS and Wiki have relatively little speedups. We believe this is for two reasons. First, the nature of web servers are random I/O accesses, thus only a few number of sequential patterns can be found. Second, most requests to web servers are small in size, e.g., 4K, which makes prefetching less effective. On the other hand, CloudStone, FS, and WP demonstrate good performance...
improvements, up to a 21% speedup on one VM and 2-7% for 12 VMs, because of more predictable, sequential access patterns. For example, the FS in the experiments has the mean file size at 64 MB and each request size is larger than 1 MB. Note it is fair and reasonable to use large file sizes because several new distributed file systems have large file sizes in practice, e.g. the Google file system uses 64 MB chunk size. For Darwin and VS, VIO-prefetching can provide over 14% speedup for 1 to 12 concurrent VMs because a number of sequential read requests are made by these video streaming services.

Note that VIO-prefetching is aware of the maximum I/O bandwidth of the system. When more VMs are sharing the same bus and storage, the available I/O bandwidth is decreasing, which leaves less room for data prefetching. Therefore, VIO-prefetching has a reduced benefit when there are more VMs in the system. However, the I/O performance can be significantly improved when the VMs are supported by a high-end storage system, such as Fibre Channel based Storage Area Network (SAN), that comes with much larger I/O bandwidth and lower latency.

Fig. 7 presents the accuracy of VIO-prefetching for different numbers of co-located VMs and applications. The accuracy is measured as the amount of prefetched and used data divided by total data used by VMs. The higher value the more accurate. The accuracy reasonably corresponds to the speedups. It is not surprising that benchmarks with more random access patterns have lower accuracies. But, VIO-prefetching effectively detects that these benchmarks have no sequential patterns and limits the number of attempts to prefetch. Fig. 8 explains this phenomenon by showing the cost of VIO-prefetching.

The idea of cost is to show the ratio of waste data amount to the total data usage. If the cost is smaller, the system spends less bandwidth in prefetching unneeded data. Therefore, the cost is defined as the ratio of the amount of unused prefetched data to the total data usage of the application and VIO-prefetching. As it is shown in Fig. 8, although VIO-prefetching does not speed up WS greatly, the prefetcher does not waste I/O bandwidth on prefetching data. When the number of VMs is large, the cost is lower because the available bandwidth is reduced and thus the prefetched and unused data amount is also reduced. The cost is higher at small numbers of VMs because VIO-prefetching aggressively consumes bandwidth. Note that the higher cost at small numbers of VMs does not mean lower applications’ performance because VIO-prefetching is using the spare bandwidth. This result clearly shows the success of pattern recognition and feedback control modules.

Prefetching on Different Read/Write Ratios We have demonstrated the experiment results on multiple VMs with a single in-guest process in Fig. 5(c) and one VM with multiple in-guest processes in Fig. 5(d). We now study how the VIO-prefetching reacts at workloads with different read/write ratios. To see how VIO-prefetching works in a more complicated environment, we demonstrate the experiments of different read/write ratios in a multiple VMs and in-guest processes environment. The following experiments are conducted on a server with two six-core Intel Xeon CPUs at 2 GHz and 32 GB memory. This machine is running Linux 3.2, Xen 4.1, and eight 120 GB SSDs configured as RAID 0 on a MegaRAID card. There are eight VMs and each has one VCPU and one GB memory. We use four in-guest processes in each VM to synthesize read/write ratios. For example, a system has a read/write ratio of 75/25 means each VM has three sequential read and one sequential write processes where the I/O size is 64 KB and file size is one GB for each process. We draw box plots in Fig. 9 to show the results. As the read ratio increases, the speedup on the VM’s throughput is enhanced because of prefetching, VIO-prefetching successfully identifies read-intensity processes and prefetches data for future use. The overall average speedup is 1.1.

Prefetching on Different Schedulers System admin-
Prefetching on Different Workload Mixtures

One user may be running video streaming services and a user’s VM is hosting web pages. But, there are many incentives, e.g., high utilization, for service providers to consolidate different VMs on the same physical machine. Thus, VMs on the same server may not run the same application. In this test, we want to see how VIO-prefetching works at different workload mixes. Because video streaming’s sequential I/O patterns are good for prefetching, we control the number of VMs that run streaming services (Darwin) from 0, 2, 4, 6, to 8 VMs. The rest VMs are randomly assigned to run either a file server, web server, web proxy, or data server. These are all multi-threaded applications and there are totally 8 VMs running concurrently. Each VM has one VCPU and one GB memory. Average speedups and standard deviations of ten runs are shown in Fig. 11. The trend shows that VIO-prefetching brings more speedup when the number of video streaming VMs increases in the workload mix. The overall average speedup is 1.07.

Prefetching on Different Request Queue Sizes

A bigger I/O request queue could provide more chances for merging multiple small random requests into a large sequential one. We benchmark Darwin and YCSB3 to test how VIO-prefetching works at various I/O request queue sizes, where Darwin is an open source version of Apple’s QuickTime video streaming server and YCSB3, which emulates Hadoop workloads, is from a performance measurement framework for cloud serving systems, YCSB (Yahoo! Cloud Serving Benchmark) [10]. The default queue size is 128 and both the guest and host have I/O request queues. In our experiments, the guest/host queue is fixed at the default size when varying host/guest queue sizes from 128, 512, 2048, to 8192. Fig. 12 shows the average speedups and standard deviations of ten runs. VIO-prefetching has improved Darwin’s performance a lot at the default queue size. When increasing either guest or host queue sizes, the speedup improvement on Darwin is limited. When increasing guest queue sizes, the speedup of YCSB3 could be improved from 1.04 to 1.1 because more random requests become a sequential one from guest machines. However, the large variance shows the unsteadiness at this setting. On the other hand, increasing host queue sizes does not improve the speedup of YCSB3. This setting may not increase the ratio of merging random requests to sequential requests because the host queue is handling requests from all guests and the random guest requests are distributed onto different separate files (virtual disks) on the storage devices.

4.5 Observations

The evaluation demonstrates how VIO-prefetching works under different system settings, e.g., I/O schedulers, queue sizes, and polling intervals. The experiments at different system settings provide useful information for the future of VIO-prefetching. We plan to explore the features of self-tuning intervals and thresholds based on the system overhead and performance feedback. Also, by utilizing the information of access...
patterns, VIO-prefetching can also dynamically adapt queue size and scheduler settings to improve the performance. By bridging the semantic gaps and providing a comprehensive virtual I/O prefetching scheme among domains, VIO-prefetching effectively utilizes the available bandwidth and detects access patterns to control prefetching and enhance virtual I/O performance. The feedback and pattern recognition modules successfully improve the accuracy and reduce the cost of prefetching in the evaluation.

5 RELATED WORK

There has been a rich set of prior research on data prefetching on hard disks, which we cannot possibly enumerate. Some representative techniques include probability graph [15], data compression [11], data mining [25], semantics-aware [5], [38], address tracking [13], [14], compiler support [30], [4], off-line information [20], [21], and hints [7]. Data prefetching can also be done at both block level (e.g., [25], [12]) and file level (e.g., [23], [50], [45]), and has been closely studied with caching [47], [50], [14], [3], [51] and parallel I/O [6]. In addition, prefetching techniques are common for fetching data from main memory on high-performance processors into processor caches where similar challenges about I/O bandwidth and pollution apply, notably [40], [16].

Our proposed method is orthogonal to techniques previously applied on bare metal systems using hard drives in the sense that we focus on prevailing virtualized servers with emerging flash-based solid-state drives and SSD based RAIDs whose high throughput provides new opportunities and challenges for data prefetching. In particular, sharing the self-monitoring and self-adapting approach as in [35], we work on the adaptation of prefetching aggressiveness in runtime to meet the needs from virtualized applications and stress SSDs within a reasonable range. In essence, our technique is also similar to freeblock scheduling [27] that utilizes free background I/O bandwidth in a hard drive. We believe that our technique can be potentially combined with a few existing prefetching techniques, e.g., [7], [50], [45].

Note that SSD devices are performing data prefetching on a small scale by utilizing parallel I/Os and an internal memory buffer. Work has been started to measure and understand this effect [8], [22], [17]. In comparison, our proposed prefetching is designed and implemented in the software layer, which can be used to complement the hardware-based approach.

Current operating systems do not have a good support for data prefetching on solid-state drives. For example, Windows 7 recommends computer systems with SSDs not use features such as Superfetch, ReadyBoost, boot prefetching, and application launch prefetching, and by default turns them off for most SSDs [37]. The key reason is that such features were designed with traditional hard drives in mind. It has been shown that enabling them provides little performance benefit [43]. Linux developers also realize the need to have a tunable I/O size as well as the need for more aggressive prefetching [48]. Development efforts on improving prefetching performance on SSDs are ongoing, and we believe that our findings will be beneficial in this area.

FAST is a recent work that focuses on shortening the application launch time and utilizes prefetching on SSDs for quick start of various applications [19]. 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 uses the blktrace API with an LBA-to-inode mapper instead of using a loopback device like us. A similar work to FAST is C-Miner [25], which discovers block correlations to predict which blocks will be accessed. This approach can cope with a wider variety of access patterns while ours is limited to simpler strided forward and backward patterns. Our approach differs from these two in that it can handle request streams from multiple simultaneous applications and includes an aggressiveness-adjusting feedback mechanism. We believe that incorporating block correlations would improve VIO-prefetching’s accuracy in some cases and plan to investigate this approach in the future.

We would also like to point out that some researchers have expressed reservations against data prefetching on solid-state drives. IotaFS chooses not to implement prefetching among the file system optimizations it used for SSDs [9]. In addition, FlashVM [34] found out that disabling prefetching can be beneficial to some benchmarks. As we have discussed before, prefetching is not always helpful for some applications, prefetching has limited benefits and may even lead to some modest regressions.

For virtual I/O prefetching, Li et al. choose to implement their prefetching method in guest OS [24]. Although prefetching in guest OS has a clear view of I/O processes in guest domain, this approach does not have the knowledge of physical disk block mappings, where the sequence of disk blocks is critical to prefetching performance. Xu and Jiang [49] proposed a scheduling framework, called stream scheduling (SS), which judiciously tracks I/O requests in both time and spatial domains for identifying streaming access patterns. Then, the I/O scheduler keeps serving I/O requests in the current stream until the stream is broken or a request with a higher priority arrives. SS does not require process information and only uses characteristics of I/O requests, e.g., request arrival times and addresses. This feature makes SS can work in a virtualized system without knowing the guest process information. On the other hand, VIO-prefetching solves this problem by bridging information gaps among domains and identifying access patterns at the block device level. We make VMs pass the in-guest process identifications to the host domain for the pattern recognition module of VIO-prefetching. Note that customizing the virtual I/O channels to enhance performance is a common practice. For example, the paravirtualization framework uses customized drivers...
to improve I/O performance. VIO-prefetching, which shares a similar philosophy as the paravirtualization technique, modifies the frontend and backend drivers to improve pattern recognition for prefetching. Several previous works utilize caching in the hypervisor to improve virtual I/O performance [18], [26], [44]. These caching techniques are orthogonal to prefetching and may be combined to provide better virtual I/O performance. It would be an interesting future work to investigate the effectiveness of VIO-prefetching with a hypervisor managed cache at various cache sizes.

6 Conclusions

We have designed and implemented a virtualization-friendly data prefetching method for prevailing virtualized servers with emerging high-performance storage devices, including flash-based solid-state drives that detects application access patterns, retrieves data to match both drive characteristics and application needs, and dynamically controls its aggressiveness with feedback. The prominent features of VIO-prefetching include not only taking advantage of the high bandwidth and low latency of SSDs, but also providing inherent support for virtual I/O and feedback-controlled aggressiveness, demanded by virtualized, shared servers in data centers. We have implemented a prototype in Linux and conducted a comprehensive evaluation on a Xen virtualization server. We integrated VIO-prefetching with the Xen virtualization server with a wide range of data-intensive cloud applications. VIO-prefetching has shown the ability to speedup a number of I/O-intensive cloud applications at various virtualized configurations. The proposed method improves virtual I/O performance up to 43% with the average of 14% for 1 to 12 VMs while running various applications on a Xen virtualization system.

In brief, the main contributions of this paper are:

- A self-tuned prefetching architecture that matches application needs without being so aggressive. Monitoring performance metrics in real-time and adjusting the aggressiveness accordingly significantly improves the effectiveness of prefetching.
- We integrate VIO-prefetching with the Xen virtualization system. The evaluation results show that VIO-prefetching successfully improve virtual I/O performance. Our comprehensive study also provides insights of VIO-prefetching’s behavior at various virtualization system configurations.
- The effects of VIO-prefetching in the context of heterogeneous applications are comprehensively studied, which indicate that adaptive prefetching and bridging the semantic gap of virtualization are essential for identifying application patterns and prefetching needed data.

7 Acknowledgments

The authors are grateful to the anonymous reviewers for their feedback and suggestions. This work is in part supported by the National Science Foundation under grants OCI-0937875. Ron Chiang did part of this research at the GWU.

References
