# Rapid Prototyping and Evaluation of Intelligence Functions of Active Storage Devices

Yongsoo Joo, *Member*, *IEEE*, Junhee Ryu, Sangsoo Park, *Member*, *IEEE*, Heonshik Shin, *Member*, *IEEE*, and Kang G. Shin, *Fellow*, *IEEE*

**Abstract—**Active storage devices further improve their performance by executing "intelligence functions," such as prefetching and data deduplication, in addition to handling the usual I/O requests they receive. Significant research has been carried out to develop effective intelligence functions for the active storage devices. However, laborious and time-consuming efforts are usually required to set up a suitable experimental platform to evaluate each new intelligence function. Moreover, it is difficult to make such prototypes available to other researchers and users to gain valuable experience and feedback. To overcome these difficulties, we propose `IOLab`, a virtual machine (VM)-based platform for evaluating intelligence functions of active storage devices. The VM-based structure of `IOLab` enables the evaluation of new (and existing) intelligence functions for different types of OSes and active storage devices with little additional effort. `IOLab` also supports real-time execution of intelligence functions, providing users opportunities to experience latest intelligence functions without waiting for their deployment in commercial products. Using a set of interesting case studies, we demonstrate the utility of `IOLab` with negligible performance overhead except for the VM's virtualization overhead.

**Index Terms—**Active storage device, intelligence function, device emulation

## 1 Introduction

Storage devices have constantly been upgraded with cutting-edge technologies to perform "something" more than just handling the usual I/O requests they receive, in order to improve their performance. We call this type of devices *active storage devices*, and "something" *intelligence functions*.

Researchers initially focused on large-scale workloads running on HDD-based massive storage systems to develop application-specific intelligence functions, such as pattern matching in the database systems [1], [2], data mining for multimedia applications [3], and text search and biological gene sequence matching [4].

However, active storage devices have been expanding their coverage since their inception. Today's active storage devices include not only HDDs but also a new type of storage devices, such as solid-state drives (SSDs), flash caches, and hybrid drives. Also, various types of general-purpose intelligence functions are now under consideration for individual user workloads running on a single storage device. Examples include prefetching [5]-[8], defragmentation [9], hot data clustering [10], [11], replication [12], [13], data pinning [14], write caching [15], run-time data deduplication and compression [16], and so on.


- Y. Joo, S. Park, and K. G. Shin are with the Department of CSE, Ewha Womans University, Seodaemun-gu 120-750, Seoul, Korea. E-mail: sangsoo.park@ewha.ac.kr.
- J. Ryu and H. Shin are with the School of CSE, Seoul National University, Seoul 151-742, Korea.
- K.G. Shin is with the Department of EECS, University of Michigan, Ann Arbor, MI 48019.


*Manuscript received 12 Jan. 2012; revised 29 May 2012; accepted 24 Apr. 2013. Date of publication 28 April, 2013; date of current version 07 Aug. 2014. Recommended for acceptance by S. Ranka.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TC.2013.101*

Although the evolution of active storage devices offers more opportunities for researchers and developers, it has been accompanied with increasing difficulties in setting up a suitable experimental platform to explore new intelligence functions. Researchers often used to rely on storage device simulators [17], [18] that provide a great deal of flexibility and low setup overhead. However, they mostly support only trace-driven simulation, lacking the ability to interact with real applications. Also, they are unable to account for the data transfer delay between main memory and a storage device, yielding inaccurate evaluation of intelligence functions.

A real system implementation would be an ultimate solution to this problem. However, it requires enormous amounts of time and effort in hacking an OS kernel or developing a new device driver that could otherwise be used to focus on the intelligence functions themselves. Moreover, researchers who are interested in emerging active storage devices often need to build a new hardware prototype by themselves [19], [20] because either they are not available as a commodity product or researchers have no access to their firmware code.

The above problems motivate us to develop `IOLab`, a new virtual machine (VM) based evaluation platform for the latest intelligence functions of active storage devices. `IOLab` is a user-space module interposed between a VM and the virtual file system (VFS) of a host OS. `IOLab` intercepts block-level I/O requests from the VM and then forwards them to the intelligence function under test that is running inside `IOLab`. The target intelligence function works by analyzing and extracting useful information from the captured I/O sequence.

As the I/O requests from the VM contain only block-level information, `IOLab` has an inherent limitation that it only supports intelligence functions based on block-level semantics. Despite the limitation, the VM based structure of `IOLab` offers a number of desirable features: (1) `IOLab` effectively

0018-9340 © 2013 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.

decouples the implementation of an intelligence function from the target applications and OSes; (2) IOLab enables real-time execution of intelligence functions as well as interacting with real applications running on the VM; and (3) IOLab is able to use any commodity block devices or combinations of them without the need of customized device drivers or special hardware.

To demonstrate the usefulness and effectiveness of IOLab, we have conducted a set of case studies, including optimizing application startup and OS boot, and prototyping a SSD + HDD hybrid drive. The performance overhead of IOLab is found to be negligible except the VM's inherent virtualization overhead.

The contribution of IOLab is that it is the first evaluation platform that exploits a VM to bridge the gap between simulation and real implementation. We expect that IOLab can significantly reduce the effort to set up an experimental platform, improving the productivity of storage and I/O systems researchers and developers. Also, IOLab enables easy distribution and real-time execution of intelligence functions, allowing users to experience the latest intelligence functions with daily workloads before their deployment in commodity systems.

The remainder of the paper is organized as follows. Section 2 provides background on active storage devices with their intelligence functions. Section 3 provides an overview of IOLab, and Section 4 discusses its features and coverage. Section 5 describes its implementation, and Section 6 presents a set of case studies to demonstrate the usefulness and effectiveness of IOLab. Section 7 compares IOLab with existing approaches, and finally, Section 8 concludes the paper.

## 2 Background

This section provides background on active storage devices and intelligence functions as well as their prevalent implementation methods.

### 2.1 Active Storage Devices

Discussed below are active storage devices that are currently available as commercial products or being actively discussed in the research community.

#### 2.1.1 Hard Disk Drive (HDD)

Currently, this is the most widely deployed active storage device. The state-of-art HDD is equipped with a sophisticated on-disk controller built with a microprocessor and a DRAM buffer of up to 64 MB. The on-disk controller, however, performs only the basic intelligence functions that are essential for the HDD to operate (e.g., LBA-to-PBA mapping and bad sector management) or have a critical influence on disk performance (e.g., write buffering and read lookahead). Numerous intelligence functions have been proposed for HDDs, but most of them are implemented on the host OS, rather than on the HDD itself.

#### 2.1.2 Solid-State Drive (SSD)

Rapid advances of the semiconductor technology have made NAND flash-based SSDs affordable even for personal storage systems. A SSD, just like a HDD, is also equipped with a controller, but the controller performs different intelligence functions. It implements a flash translation layer (FTL) which performs not only LBA-to-PBA mapping but also wear-leveling and garbage-collection. New SSD controllers are getting equipped with even more complicated intelligence functions, such as deduplication and real-time data compression [16] as well as a built-in sanitization function [21].

#### 2.1.3 Hybrid Drive

Two or more different types of storage media can be combined to form a "hybrid" drive to overcome the performance drawback of HDDs without increasing costs too much. A small size of flash memory is integrated into a HDD [22], or implemented as a PCI-express card [23] to be used as a nonvolatile cache. The authors of [19], [24] proposed to combine a small SSD and a large HDD, where data can be dynamically migrated between two devices to optimize metrics such as performance and device lifetime.

### 2.2 Intelligence Functions

Active storage devices may perform an application-specific intelligence function, such as database applications [2], data mining for multimedia applications [3], and text search and biological gene sequence matching [4]. Another type of active storage device provides multiple views of a file [25] or supports context-aware adaptation [26] to meet the various needs of users.

In addition to these, there also exist various types of intelligence functions for optimizing general workloads that are currently under study, including:

1. *Prefetching*: Various types of prefetching techniques have been proposed and studied to hide disk access latencies, based on sequential pattern detection [5], [7], history-based prediction [8], or user access pattern analysis [27].
2. *Defragmentation*: Fragmented files are rearranged periodically or upon a user's request so as to make each file occupy a contiguous disk space [9].
3. *Hot data clustering*: Hot data blocks are identified and migrated to a small, dedicated region of a disk so as to reduce disk access time for successive accesses of the hot data blocks [10], [11].
4. *Replication*: A data block is replicated to two or more physical locations on a disk, and when an I/O request for that data block is issued, a disk controller chooses the replica closest to the current location of the disk head to service the I/O request [12], [13].
5. *Data pinning*: Frequently-used data blocks can be pinned to dedicated non-volatile cache memory to accelerate the access speed for them [14].
6. *Write caching*: Non-volatile cache is used to maximize the spin-down time of a HDD, aiming at reducing energy consumption [15].
7. *Deduplication and data compression*: Runtime data deduplication and compression are performed to reduce the amount of data to be written, which is shown to be beneficial, especially for SSDs [16].

### 2.3 Prevalent Implementation Methods

Depending on the target of optimization and the type of information being exploited, intelligence functions can be

implemented at one or more places between the application level and the device level in a computer system:

1. *Application level*: Intelligence functions can be tightly integrated into an application to fully exploit the application-level information [28]-[30].
2. *OS level*: Many intelligence functions are implemented as an OS daemon process, such as application prefetching and disk defragmentation [31].
3. *File-system level*: File systems can be extended to include intelligence functions that exploit file-level semantics [12], [32], [33].
4. *Block I/O level*: Intelligence functions can be inserted in the OS block I/O layer if they use only the block-level information [11].
5. *Device-driver level*: A pseudo device driver can be used to represent an active storage device where the intelligence function is able to manipulate block-level information like the block I/O level implementation [24], [34].
6. *Device level*: A recent hybrid HDD integrates flash cache management functions with its controller, which is different from previous hybrid HDDs in that it is fully OS-independent [35]. Researchers utilize reconfigurable logic on the disk to implement their intelligence functions [4].

## 3 The Proposed Evaluation Platform

Our goal is to build a flexible evaluation platform to allow for rapid prototyping and easy distribution of intelligence functions for active storage devices. We propose `IOLab`, a new VM based evaluation platform, which is a userspace module interposed between a VM and the VFS of a host OS.

### 3.1 A Structural Overview

Fig. 1 depicts a structural overview of a system with `IOLab`, which consists of an application layer, a host OS layer, and a block device layer.

#### 3.1.1 Application Layer

There are two applications running on a host OS: a virtual machine monitor (VMM) and `IOLab`.

1. VMM: `IOLab` uses I/O requests generated by real applications as its input I/O trace. Instead of executing a target application directly on the host OS, `IOLab` runs it on the guest OS managed by the VMM. This design allows `IOLab` to effectively separate the implementation of an intelligence function from a specific OS. `IOLab` is designed to support a VMM that: (1) uses full virtualization mode, (2) uses a file-backed virtual disk image (VDI), and (3) accesses the VDI using file I/O system calls.
2. `IOLab`: intercepts file-level I/O requests from the VM to the VDI file that would otherwise be sent directly to the host OS. The intercepted file I/Os actually contain the information of block-level accesses to the VDI. An intelligence function is running inside `IOLab`, analyzing all the I/O requests it receives to extract useful information. Based on the thus-obtained information, the intelligence function can modify the original I/O requests or create new I/O requests, all of which are gathered and reordered according to their priorities before they are sent to the host OS.

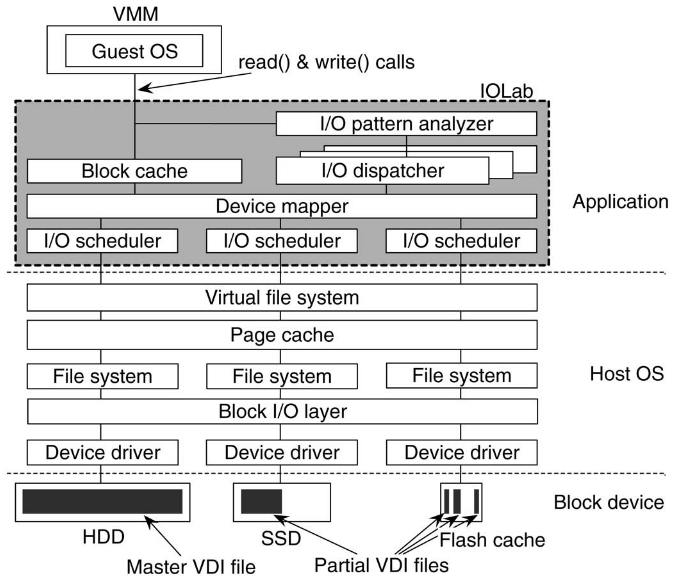


Fig. 1. The overview of a system with `IOLab`.

#### 3.1.2 Host OS Layer

The host OS receives I/O requests from `IOLab` and passes them to the associated component block device while keeping the received I/O stream as unmodified as possible. Linux Fedora 14 x64 (2.6.37 kernel) with the EXT4 file system is used as the host OS.

#### 3.1.3 Block Device Layer

To express various types of active storage devices, `IOLab` uses a set of commodity block devices, which we call "component block devices," that are connected to a host machine via the device drivers of the host OS. For example, using two component block devices—a HDD and a SSD—`IOLab` can express four types of active storage devices: a HDD, a SSD, a HDD with a flash cache, and a SSD + HDD hybrid drive.

### 3.2 VDI Manipulation

`IOLab` keeps a master VDI file, which can be split into multiple partial VDI files distributed over other component block devices, as shown in Fig. 1. The distribution depends on the type of the intelligence function that is to be evaluated. Regardless of how the master VDI file is distributed over component block devices, `IOLab` provides an illusion of the master VDI file to the VMM.

Modern VMMs support two types of VDIs: file-backed and partition-backed VDIs. For the latter, the partition can be logical or physical. `IOLab` supports file-backed VDIs because: (1) it is easy to intercept I/O requests via wrapping a few file I/O system calls; (2) run-time creation of partial VDIs is straightforward by creating new files; and (3) multiple VDIs can share a single component block device, allowing convenient setup of `IOLab` for users not having a dedicated block device for experimental purpose.

## 4 Features and Coverage

In this section, we discuss the limitations and possible extensions of `IOLab`.

## 4.1 Features

IOLab offers several important advantages as follows:

1. *Easy deployment:* IOLab is implemented as an application of the host OS, and hence, does not rely on customized device drivers or any special hardware. IOLab only requires minor modification of a few file-I/O-related system calls of the host OS at installation time. Once installed, IOLab does not require OS kernel recompilation upon change of an intelligence function or an active storage device.
2. *Modular design:* IOLab employs a VMM to run a target application, which effectively decouples the implementation of an intelligence function from the guest OS where the target application is running. Thus, IOLab facilitates the evaluation of intelligence functions for different OSes without any modification of target applications or guest OSes.
3. *Real-time execution:* IOLab expresses an active storage device by combining a set of commodity component block devices that are real hardware. Consequently, IOLab supports real-time execution, and the performance improvement achieved by using IOLab is an immediate benefit to users who run their applications on a VM.
4. *Extensibility:* IOLab utilizes the device drivers provided for the host OS, thereby inheriting the extensibility of the host machine. Any block device can be used as a component block device as long as it can be connected to the host machine and recognized by the host OS. IOLab also supports rapid prototyping of a hybrid storage device by combining heterogeneous component block devices at block level.

## 4.2 Support of Intelligence Functions

Depending on how to obtain the semantic information from I/O requests, intelligence functions are categorized into "blackbox," "graybox," and "whitebox" approaches, as discussed in [36], [37].

### 4.2.1 Blackbox

IOLab basically supports intelligence functions by taking a blackbox approach, which operates with only block-level information. Although IOLab intercepts file-level I/O requests from the VM to the VDI file, they are in fact block-level I/Os to the virtual block device abstracted by the VDI file (i.e., the file offset and size are always multiples of the block size). As IOLab only sees block-level information, it cannot directly access the semantic information available inside the guest OS (e.g., the file type or process ID of each I/O request).

### 4.2.2 Graybox

The authors of [38] proposed a graybox approach to infer the semantic information inside the OS by running a probe process on the OS. IOLab can support this approach by running the probe process in the VM.

### 4.2.3 Whitebox

There have been various whitebox approaches that explicitly pass the semantic information to an intelligence function by extending the traditional storage I/O interface [28], [34], [37], [39]. Although the current implementation of IOLab does not support these whitebox approaches, it can be easily extended to support them. A possible extension could be to create a communication channel between the guest OS and IOLab to transfer the semantic information.

## 4.3 Support of Active Storage Devices

Similar to the taxonomy presented in Section 4.2, intelligence functions can also be categorized as blackbox or whitebox, depending on how they treat an active storage device.

### 4.3.1 Blackbox

Most modern active storage devices provide only a logical block addressing interface, hiding their internal structure. Hence, intelligence functions running outside the active storage devices are usually implemented without knowing the internal structure of a target storage device. Their common assumption is that a target storage device will provide better performance for large sequential I/O requests. On the other hand, researchers have suggested the inference of its performance characteristics by running a prove function [40]-[44], which is similar to the graybox approach discussed in Section 4.2.2. IOLab supports these types of intelligence functions as it runs on commodity block devices, i.e., real hardware.

### 4.3.2 Whitebox

Some intelligence functions, such as a FTL for SSDs, essentially exploit the information on the internal structure of the target device, and are thus integrated into the device controller [45], [46]. The most common approach taken in studying this type of intelligence functions is to use a device simulator [17], [18], [47]. Although IOLab does not support this type of intelligence functions, it can be extended to recognize the device simulator as a component block device by taking a similar approach of device emulation, as discussed in Section 7. Through this extension, IOLab can also emulate emerging storage devices that are not yet commercially available (e.g., PRAM caches). Once such devices become available, IOLab can immediately support them by simply connecting them to the host machine, very much like HDDs and SSDs.

## 4.4 Input Workload

Although IOLab is intended to support as many types of intelligence functions and active storage devices as possible, its current implementation focuses mainly on individual user workloads from personal storage systems. For example, IOLab receives input I/O requests from a single VM, focusing on individual user workloads. IOLab is also designed to evaluate a single active storage device for personal storage systems.

# 5 Implementation

The implementation of IOLab consists of a set of subcomponents, most of which are reconfigurable. We prototype an intelligence function by customizing necessary subcomponents while using the default configuration for the rest.

### 5.1 Intercepting Input I/O Requests

IOLab should intercept the file I/O requests from the VMM to the VDI file without touching the I/O requests made to other files. To implement this, we first define a flag bit O_VDI to indicate that a file opened with O_VDI is one of the VDIs composing a target active storage device. All read() and write() calls to the file opened with O_VDI are to be intercepted by IOLab. To set O_VDI, we exploit one of the unused bits in the unsigned int variable f_flags of the file structure, which is defined in "include/linux/fs.h."

To be consistent with the conventional procedure of the open() call, the VMM must pass the O_VDI flag as an argument to the open() call when it opens the VDI file. Alternatively, we modify the open() call such that, when it is invoked, it first checks the sticky bit of the file to be opened. If it is set, the modified open() call sets O_VDI. We assume that the sticky bit is set by a user for all of the VDI files of interest, which can be done with the command:

```
$ chmod + t [VDI_filename].
```

We then modify the read() call such that, when it is invoked, it first checks the f_flags of the file structure, which can be accessed using its file descriptor. If O_VDI is set, all of its function arguments are forwarded to IOLab. Otherwise, the original code of read() call is executed. Likewise, we modify the write(), aio_read(), and aio_write().

### 5.2 I/O Pattern Analyzer

The I/O pattern analyzer monitors all accesses to the VDI to infer useful information to be exploited by the intelligence function. Example types of information include deterministic I/O request sequences, access frequency statistics of data blocks, and the event of accessing a certain data block.

The I/O pattern analyzer also has the role of initiating a new I/O dispatcher when a certain condition predefined by the intelligence function is met (e.g., a counter variable reaches its threshold, or an access to a certain data block occurs).

### 5.3 Block Cache

Data blocks requested by the VM are serviced by the block cache of IOLab, instead of being sent directly from the active storage device. The block cache also manages the data blocks fetched by the I/O dispatchers.

The block cache is implemented in such a way that it stores only metadata of the I/O requests (e.g., a start address and a block count) to keep track of the lists of data blocks it manipulates. It relies on the page cache of the host OS to actually store the data blocks to be cached. This way, the block cache can keep its implementation simple and efficient, while saving the main memory space of the host PC.

Under the proposed block cache structure, the total size of data blocks managed by the block cache can grow up to the page cache size of the host OS. To control the effective capacity of the block cache, we deploy a separate page replacement policy in addition to that included in the page cache of the host OS. The page replacement policy of the block cache can be configured according to a target intelligence function. For example, a data block can be immediately evicted from the page cache as soon as it is read by the VM. Another example is to set a constant capacity limit so that victim blocks are evicted when the block cache size exceeds the preset value.

Note that some VMMs use O_DIRECT flag for read() and write() system calls to bypass the page cache of the host OS. In such a case, we force the VMM not to use O_DIRECT flag.

### 5.4 I/O Dispatcher

I/O dispatchers perform actual I/O operations intended by the intelligence functions, such as prefetching, data migration, and data replication. It is done by dispatching I/O requests one-by-one from the predefined block request stream configured by the I/O pattern analyzer.

The dispatching rate of I/O dispatchers can be controlled directly by setting the maximum data transfer rate or indirectly by the capacity limit of the block cache. For example, an I/O dispatcher will be blocked if there is no free space in the block cache. A separate space limit may also be specified for each I/O dispatcher.

### 5.5 Device Mapper

The device mapper provides mapping between a master VDI and partial VDIs distributed over component block devices. When two or more component block devices form a single active storage device, the intelligence function manages the mapping table of the device mapper.

The device mapper receives input I/O requests from both the block cache and the I/O dispatchers, and then converts their offset to the file descriptor and the offset of the partial VDI file containing the requested data. If there are two or more locations storing the requested data, the intelligence function is responsible for deciding on how to handle it. For a read request, the intelligence function should choose one of the locations it found (e.g., one stored in the fastest component block device). For a write request, it may either choose one and invalidate the rest, or update all the locations by issuing I/O requests to every partial VDI file that has the data. The thus-processed I/O requests are then sent to their corresponding I/O scheduler.

The device mapper works at page-level granularity to reduce the overhead of maintaining the mapping table. Its block-mapping information is managed by using the radix tree—provided by the linux kernel library—to limit the mapping overhead of large VDI files. A radix-tree node consists of (dev, inode, page offset) of the partial VDIs. For example, the height/level of the radix tree is only 4 when the master VDI file size is 64 GB and the page size is 4 KB (i.e., 8 blocks).

### 5.6 I/O Scheduler

IOLab deploys separate I/O schedulers for each component block device, which operates independently from the I/O scheduler of the host OS. The purpose of the dedicated I/O scheduler is to prioritize between the I/O requests from the VM and those from the I/O dispatchers, which is often essential for realizing various types of intelligence functions.

Each I/O scheduler receives I/O requests from the device mapper to reorder them according to their preset priorities. The prioritized I/O requests are then sent to the VFS of the host OS, which finally commits them to the corresponding partial VDI file.

## 6 Evaluation

We first describe the experimental setup we use to evaluate IOLab, and present a set of case studies focusing on application launch and OS boot optimization to demonstrate the usefulness of IOLab. Finally, we evaluate the prototyping effort and performance overhead of IOLab.

### 6.1 Experimental Setup

We chose VMWare Workstation 7.1.1 as the VMM of IOLab, and selected NOOP as the I/O scheduler of the Linux (the host OS of IOLab), and disabled its readahead function to minimize the host OS intervention. The host machine we used for experiments is equipped with an Intel i7-860 2.8 GHz quad-core CPU and 4 GB of main memory. We configured the VM to use 2 cores of the CPU with hyper-threading enabled and 1 GB of memory. For component block devices, we used a Western Digital 3.5" 7200 RPM 640 GB HDD (WD6400AAKS), a Fujitsu 2.5" 5400 RPM 120 GB HDD (MHZ2120BH), and an Intel 40 GB MLC SSD (X25-V). The disk block accesses shown in the following case studies are measured with IOLab.

### 6.2 Case Study 1: Application Prefetcher

The application prefetcher optimizes application launch performance by prefetching all the data blocks necessary for starting the application in an optimized fashion just before its launch process begins. The application prefetcher has been included in the Windows XP and its subsequent versions, which we will call *Windows prefetcher*.

The execution of the application prefetcher involves the following phases.

1. *Learning phase*: monitors and logs all the I/O requests generated during the launch of each target application to determine the set of data blocks necessary for its launch.
2. *Post-processing phase*: creates an *application launch sequence* using the information obtained from the learning phase. It reorders the data blocks using a predefined sort key (e.g., inode number and file offset), and stores the resulting sequence in the reserved system folder (e.g., C : \ WINDOWS\ PREFETCH for the Windows prefetcher).
3. *Prefetching phase*: When it detects a new launch of the target application that has an application launch sequence file, this phase (1) immediately pauses the launch process; (2) fetches the data blocks in the order specified in the sequence file; and (3) resumes the launch process.

#### 6.2.1 Observation

To see how the Windows prefetcher works in practice, we captured the output I/O requests from the block cache of IOLab while launching MS Office Word 2007 on the HDD. To ensure a cold start scenario, we flushed the page cache of both the guest OS and the host OS before launching the application. Fig. 2a shows the disk block accesses with the Windows prefetcher disabled. In this case, the launch time was measured to be 8.3s. Once we enabled the Windows prefetcher, the launch time was reduced to 5.7s (Fig. 2b), clearly showing the benefit of the Windows prefetcher.

#### 6.2.2 Motivation

Despite the performance improvement, Fig. 2b shows that there still exists a considerable number of random block accesses even with the Windows prefetcher, which is far from what we expected. As the Windows OS is not an open source, it is difficult to analyze its behavior in detail. Instead, we decided to implement our own application prefetcher on IOLab.

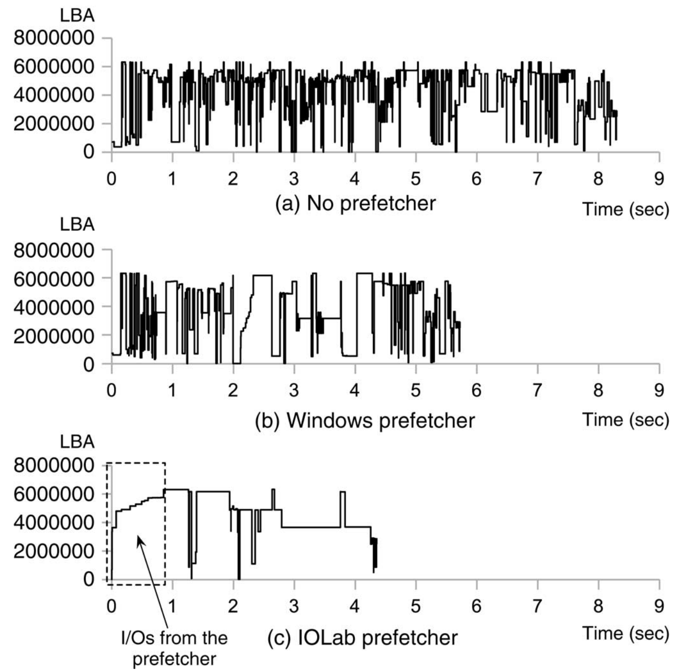


Fig. 2. Disk block accesses of the Windows prefetcher and the IOLab prefetcher (OS: Windows XP, application: Word 2007, used device: WD6400AAKS. (a) No prefetcher; (b) Windows prefetcher; (c) IOLab prefetcher.

#### 6.2.3 Configuration

We configured IOLab so that it performs the application prefetch as described above, which we call the IOLab prefetcher. Given below is a detailed account of how we configure the IOLab prefetcher.

1. *Launch sequence creation*: For this we can use the method in [36] that automatically and accurately mines the block correlation information. For simplicity, in this experiment, we manually captured the I/O request sequence generated during the application launch.
2. *Application launch detection*: The VM does not inform IOLab when the target application is launched. Hence, we configured IOLab to initiate an I/O dispatcher immediately when 3 consecutive block requests from the VM match with the captured sequence.
3. *Application prefetcher generation*: The above-created I/O dispatcher is configured to fetch the blocks of the captured sequence in their sorted order of LBAs.
4. *Launch control*: We set the priority of the I/O dispatcher higher than the VM to pause the launch process of the target application running on the Windows OS. Upon completion of the prefetching, IOLab resumes the target application by responding to the I/O requests from the VM.

#### 6.2.4 Experiment

We conducted experiments on the WD6400AAKS HDD, and plotted the resulting disk accesses in Fig. 2c. The dashed box

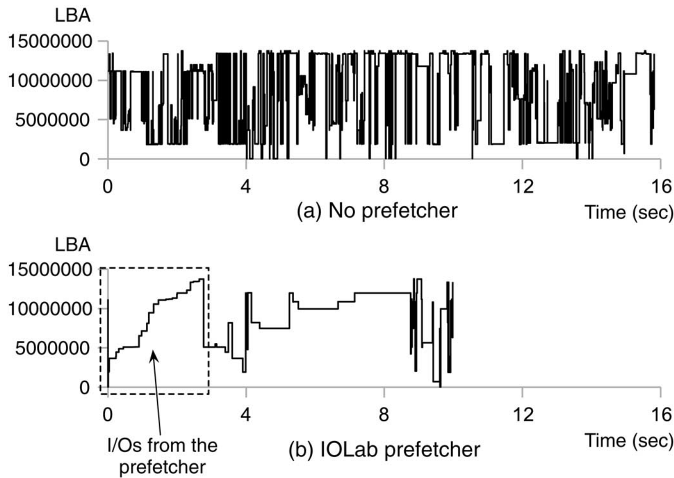


Fig. 3. Disk block accesses of the IOLab prefetcher on the Linux OS (application: Eclipse, device: WD6400AAKS). (a) No prefetcher; (b) IOLab prefetcher.

in Fig. 2c indicates that the IOLab prefetcher made the LBAs of I/O requests monotonically increased as intended. As a result, the application launch time is reduced to 4.4s (a 23% improvement over the Windows prefetcher). However, some I/O requests still occur after the completion of the prefetcher because they were not included in the captured I/O request sequence. The prefetch hit ratio was observed to be 95.4% (by dividing the number of I/O requests IOLab prefetched by the total number of I/O requests from the VM).

#### 6.2.5 Test on the Linux OS

Although there are suggestions to use the application prefetcher for Linux OS [48], it is not yet available in official Linux distributions. We tested the IOLab prefetcher for the Eclipse application installed on the Linux OS, which required no change in the IOLab configuration. Fig. 3 shows a 37% reduction of the application launch time (from 15.8s to 10.0s), where the prefetch hit ratio was 97.5%.

#### 6.2.6 Summary

This case study demonstrates the usefulness of IOLab in emulating proprietary intelligence functions and assessing the possibility of improving them further. In particular, IOLab supports adjusting the I/O priority between its I/O dispatcher and the target application, through which IOLab is able to control the pause and resumption of the target application without modifying the guest OS of the VM. It also demonstrates the convenience of deploying the same intelligence function on different OSes without any porting effort.

### 6.3 Case Study 2: OS Boot Optimization

Modern OSes often employ various optimizations to reduce their boot time. For example, the Windows OS applies its application prefetch technique again for its boot process, and Mac OS deploys a similar technique called *bootcache* [49]. Thanks to the VM based structure, IOLab can easily observe the I/O requests generated during the OS boot, and apply an intelligence function to assess its potential for improving the OS boot time reduction.

#### 6.3.1 Observation

Windows XP supports use of its application prefetcher for the OS boot process [31]. As Windows XP allows its boot prefetch to be turned on via registry configuration, we could capture the I/O requests during the boot process without and with the boot prefetch, which are shown in Figs. 4a and 4b, respectively.

Enabling the boot prefetch is shown to increase the boot time of Windows XP by 13%, but Windows XP reads 192.7 MB of data with the boot prefetch, which is more than twice the data (93.4 MB) reads without the boot prefetch. This is because Windows XP fetches not only boot files but also frequently-used application files when the boot prefetch is enabled [50], allowing their quick launch after the boot completion. The dash-lined boxes in Fig. 4b show that the boot prefetch fetches data blocks in an optimized way.

For Mac OS X, we also observed a similar pattern as marked by the dash-lined box in Fig. 4c, indicating that its bootcache works well. In Fig. 4d, however, Linux does not show any optimized disk access pattern, indicating that not much has been done on Linux boot optimization.

It is interesting to see considerable disk idle periods for all the cases of Fig. 4. For example, the idle period was about 20s for Windows XP. One possible reason for this is the delay for I/O device detection, but it needs a further study for accurate understanding of this behavior.

#### 6.3.2 Motivation

The results shown in Fig. 4 raise two questions: (1) what will be the performance improvement of Linux with boot optimization applied; and (2) will it be possible to eliminate the inefficiency resulting from disk idle periods, especially for the boot of Windows XP. To answer the first question, we decided to apply the IOLab prefetcher, described in Section 6.2. We chose correlation-directed prefetching (CDP) [36] to answer the second question.

#### 6.3.3 Correlation-Directed Prefetching (CDP)

Block access streams, which are non-sequential, often repeatedly occur in storage systems [11], [36], [51]. The determinism in block access patterns can be used to reorganize the disk layout of related blocks [51], or to perform prefetching while optimizing the disk head movements as in the Windows prefetcher. CDP takes a different approach in that it performs prefetch using the detected I/O request sequence as hints for what to prefetch next. A similar approach was also suggested for improving application startup performance on SSDs [6].

#### 6.3.4 Configuration

For the experiment of Linux boot optimization, we used the configuration of IOLab prefetcher in Section 6.2. For the implementation of CDP, we modified the IOLab prefetcher such that (1) the created I/O dispatcher maintains the original block request sequence instead of reordering them; and (2) the priority of the I/O dispatcher is set lower than that of the VM in order to simultaneously execute the OS boot process and the CDP. If the guest OS generates an I/O request that is not covered by the CDP, the I/O request will not be blocked by the prefetcher. We will call this configuration the IOLab CDP.

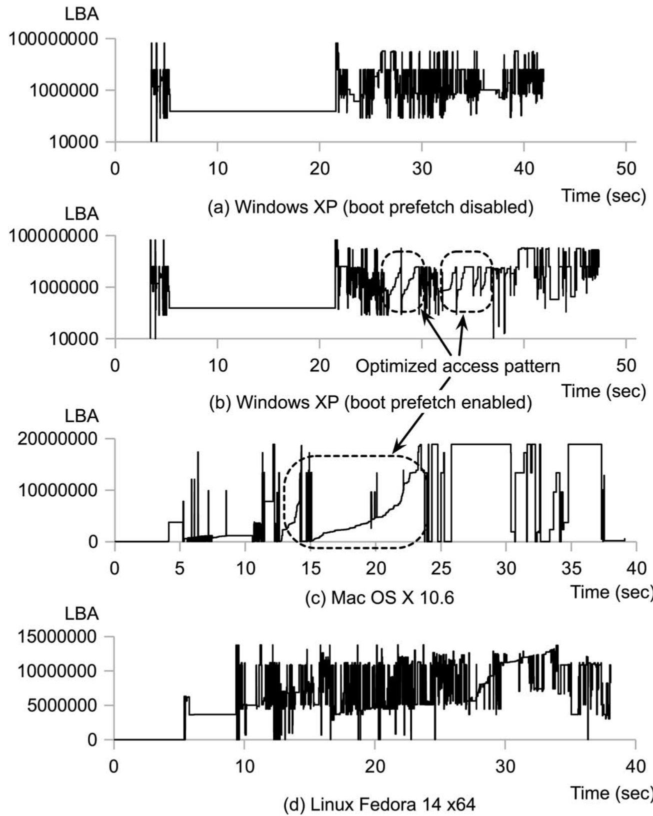


Fig. 4. Disk block accesses of the boot processes of Windows XP, Mac OS X, and Linux Fedora (device: WD6400AAKS, $y$-axis: in log scale for Windows XP). (a) Windows XP (boot prefetch disabled); (b) Windows XP (boot prefetch enabled); (c) Mac OS X 10.6; (d) Linux Fedora 14 x64.

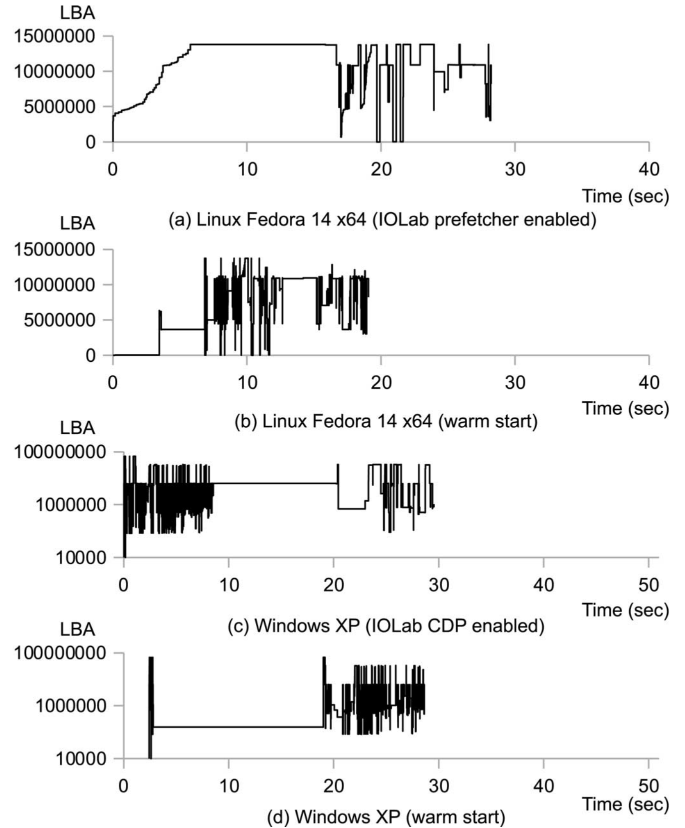


Fig. 5. Disk block accesses of Linux and Windows XP with `IOLab` optimization applied (used device: WD6400AAKS, $y$-axis: in log scale for Windows XP). (a) Linux Fedora 14 x64 (`IOLab` prefetcher enabled); (b) Linux Fedora 14 x64 (warm start); (c) Windows XP (`IOLab` CDP enabled); (d) Windows XP (warm start).

#### *6.3.5 Experiment*

Fig. 5 shows the experimental results on the WD6400AAKS HDD, where Figs. 5b and 5d visualize the artificial disk block accesses assuming use of an ideal block device having zero access latency. These results are obtained by booting the VM immediately after shutting it down, so all the boot files remained in the page cache of the host OS, causing no disk access for the second boot. This "warm start OS boot time" provides an upper bound of I/O performance improvement.

Fig. 5a depicts the resulting disk block accesses by applying the `IOLab` prefetcher to the booting of Linux. The boot time is reduced from 38.1s (Fig. 4d) to 28.2s (a 26% improvement), and the prefetch hit ratio was 95.2%. Fig. 5c shows that the `IOLab` CDP works as intended by aggressively fetching the boot files during the otherwise idle period of the HDD. The achieved boot time of Windows XP (29.5s) is close to its warm start boot time (28.6s) of Fig. 5d. The prefetch hit ratio was 98.0%.

#### *6.3.6 Summary*

This case study shows that `IOLab` can be used to observe and optimize the disk access pattern during the OS boot process. The experimental results allow us to estimate the potential advantage of optimizing the boot process of different OSes. The achieved performance improvement is an immediate benefit to the users who use a VM that frequently reboots the guest OS.

### 6.4 Case Study 3: SSD + HDD Hybrid Drive

Newly emerging memory devices such as NAND flash and PRAM make it natural for researchers to conceive a hybrid drive by combining heterogeneous block devices, such as SSD + HDD and PRAM + NAND. However, only recently researchers have begun to actively explore hybrid drives [19], [24], [52].

#### *6.4.1 Motivation*

A major impediment in exploring the design of hybrid drives has been their unavailability as commodity products. Although Intel Turbo memory [23] and Seagate Momentus XT [22] have recently been announced, their cache management policies are proprietary, not open to researchers. As a result, most of the previous work on hybrid drives relies on simulation, which lacks evaluation accuracy. Payer *et al.* [19] used a SATA bridge chip to prototype a real SSD + HDD hybrid drive, but their prototype was not available to the researchers (including ourselves). Thus, we decided to prototype a SSD + HDD hybrid drive like the one in [19] using `IOLab`.

#### *6.4.2 Configuration*

We "composed" a hybrid drive using the Intel X25-V SSD and the Fujitsu MHZ2120BH HDD. The capacity of the MHZ2120BH HDD after formatting it is 107 GB, and the master VDI size is set accordingly. We configured a 107 GB SSD + HDD hybrid drive such that its first 4 GB is mapped to the X25-V SSD and the rest to the MHZ2120BH HDD.

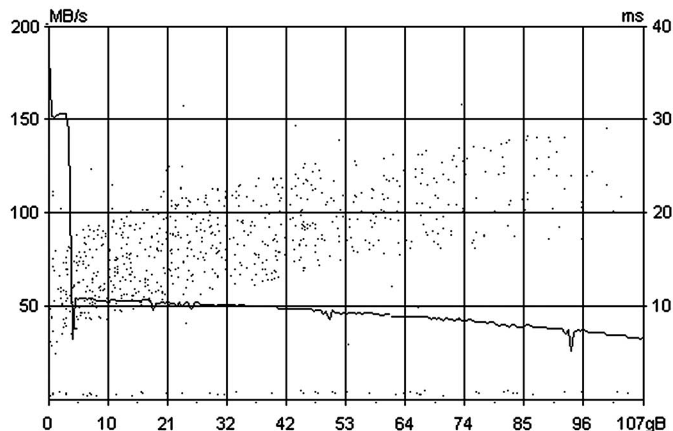


Fig. 6. Read throughput (line) and access latency (dots) of the hybrid drive, captured from HD Tune Pro 4.6 (device used: X25-V and MHZ2120BH, left $y$-axis: average throughput, right $y$-axis: access latency).

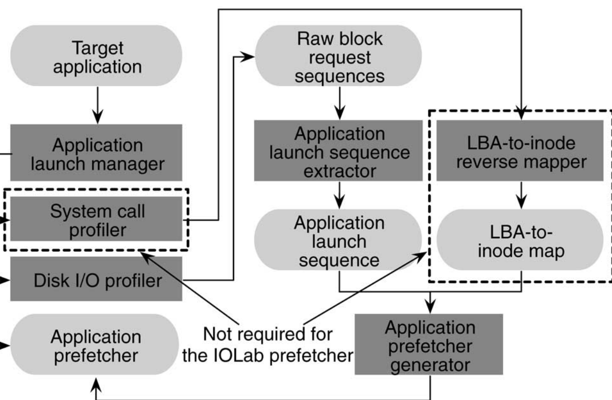


Fig. 7. The structure of FAST, a Linux implementation of the application prefetcher [6].

#### 6.4.3 Experiment

We ran HD Tune Pro, a HDD benchmarking tool, to observe the read performance of the emulated hybrid drive, and plotted the results in Fig. 6. The $x$-axis denotes the location of the hybrid drive for read throughput, and the seek distance for access latency. The read throughput, depicted as a solid line, is shown to be 150 MB/s for the first 4 GB SSD region. Then, it drops to about 50 MB/s as it enters the HDD region, and gradually decreases as the disk head moves to the inner track of the MHZ2120BH HDD. The dotted-line access latency curve represents the seek latency of the MHZ2120BH HDD, where the variation for the same seek distance corresponds to the rotational delay variations. Dots showing a near zero access latency spread over all the seek distances, corresponding to the access latency from anywhere in the HDD region to the SSD region. The boot time of Windows XP on this hybrid drive was measured to be 34.6s. We also measured the boot time using the MHZ2120BH HDD only, which was 49.1s. The warm start boot took 28.6s.

#### 6.4.4 Summary

This case study demonstrates the ability of `IOLab` in rapid prototyping of a hybrid drive by combining commodity block devices.

### 6.5 Prototyping Effort

We evaluated the effectiveness of `IOLab` in reducing the effort to prototype an intelligence function. We used the `IOLab` prefetcher of Section 6.2 as the target intelligence function. For comparison purpose, we chose FAST (Fast Application STarter), which is a Linux implementation of the application prefetcher we have developed in our recent work [6]. Both the `IOLab` prefetcher and FAST were developed on the same host machine described in Section 6.1.

#### 6.5.1 Structural Difference

FAST has OS-dependent constraints in its implementation, making its structure more complicated than the `IOLab` prefetcher. In particular, FAST should fetch the blocks of an application launch sequence to the Linux page cache, whereas the `IOLab` prefetcher uses the block cache of `IOLab`.

The Linux page cache is organized as a radix tree per node, and its cached blocks are always indexed and searched by using their associated file name and file offset. To comply with this, FAST needs to convert the block-level representation of each block request in the application launch sequence into file-level one before generating an application prefetcher (i.e., $(\texttt{LBA}, \texttt{size}) \rightarrow (\texttt{filename}, \texttt{offset}, \texttt{size})$). In contrast, the `IOLab` prefetcher does not require such conversion because the block cache of `IOLab` operates outside the guest OS with only using block-level information.

Fig. 7 depicts the structure of FAST, where the components in the dash-lined boxes are related to the block-level to file-level conversion process of the application launch sequence. Note that the LBA-to-inode reverse mapping is essential to perform the conversion process, but most file systems including EXT3 of the Linux OS do not support it. Hence, we had to develop one by ourselves, which was the most time-consuming task, though intuitive.

#### 6.5.2 Quantification of the Prototyping Effort

Table 1 compares the prototyping effort of FAST and the `IOLab` prefetcher in terms of lines of code (LOC) and developing time. As we do not have exact data for developing time of each component, we presented only the total time for development. The total LOC of the `IOLab` prefetcher and its total development time are 11% and 4% of FAST, respectively.

### 6.6 Performance Overhead

While the performance overhead of `IOLab` mostly comes from the VM's virtualization overhead, `IOLab` itself can also affect the performance. We performed a set of experiments to estimate the performance overhead of `IOLab`.

#### 6.6.1 Virtualization Overhead

To evaluate the effect of the virtualization overhead of `IOLab`, we chose an application prefetcher as a target intelligence function, as in Section 6.5. In particular, we prepared two system configurations: (1) we run the Linux application `Eclipse` on the VM with the `IOLab` prefetcher; and (2) we run `Eclipse` directly on the host OS with FAST. We used the same host machine of Section 6.1 for the experiment.

TABLE 1
Comparison of the Prototyping Effort between FAST [6] and IOLab Prefetcher

| Component | FAST | | IOLab prefetcher (Section 6.2) | |
|---|---|---|---|---|
| | LOC | Note | LOC | Note |
| Application launch manager | 538 | | 410 | |
| System call profiler | - | use strace | - | not required |
| Disk I/O profiler | - | use blktrace | - | included in IOLab |
| Application launch sequence extractor | 353 | | 286 | |
| LBA-to-inode reverse mapper | 5608 | | - | not required |
| Application prefetcher generator | 421 | | 69 | |
| Total | 6920 | took 6 months to develop | 765 | took 1 week to develop |

We first observed the effect of CPU virtualization overhead by comparing warm start time for both the configurations. We created a warm start scenario by launching Eclipse immediately after completing its associated prefetcher. In this way, all the data blocks in the application launch sequence hit in the main memory, effectively minimizing disk accesses during the launch time. The first row of Table 2 shows that the VM achieved 20% longer launch time than that of the host OS.

We then measured execution time of only an application prefetcher to estimate I/O virtualization overhead. As an application prefetcher only issues I/O requests for the predetermined set of data blocks, the CPU remains mostly idle except for processing the I/O requests. According to the second row of Table 2, the IOLab prefetcher achieves 5% longer execution time than that of FAST, showing that the I/O virtualization overhead of IOLab is relatively smaller than its CPU virtualization overhead.

The above results indicate that IOLab effectively renders an application launch procedure more CPU-bound, which in turn reducing the efficiency of an application prefetcher that optimizes only I/O latency. This is also supported by the measurements of the third and fourth rows of Table 2; FAST reduced application launch time by 42% whereas the IOLab prefetcher achieved only 37% reduction.

### 6.6.2 I/O Interception

Since IOLab modifies a set of system calls to intercept I/O requests from the VM, it may increase the latency of these system calls. We chose the OS boot time to capture the I/O performance degradation caused by IOLab, because the accumulated delays of thousands of read() calls from OS boot are likely to make the measurement easier.

IOLab was set to its default configuration so that it just passes the intercepted I/O requests from the VM to the VDI file. For the experiment with IOLab disabled, we cannot use the I/O trace logging function of IOLab to measure the OS boot time. So, we modified the host OS kernel to monitor the I/O requests from the VM to the first and last block of the Windows XP boot sequence. For the experiment with IOLab enabled, however, we monitored the OS boot time using IOLab.

TABLE 2
Performance Differences between FAST [6] and IOLab Prefetcher (Unit: Seconds)

| Scenario | FAST | IOLab |
|---|---|---|
| Warm start | 6.34 | 7.60 |
| Prefetcher execution | 2.65 | 2.77 |
| Cold start with the prefetcher disabled | 15.50 | 15.82 |
| Cold start with the prefetcher enabled | 8.99 | 9.98 |

We measured the boot times of Windows XP on the WD6400AAKS HDD with and without running IOLab. We repeated each experiment five times. The total amount of data transferred in the Windows XP boot process was 93.4 MB, which accounts for 4103 read() and 57 write() calls. The average OS boot times with and without IOLab were 41.6s and 41.7s, respectively. The measured min-max boot time difference was 1.0s without IOLab and 0.6s with IOLab. In summary, the I/O interception overhead of IOLab appears to be unmeasurable due to the boot time variations.

### 6.6.3 Device Mapper

The device mapper of IOLab lies on the critical path of the I/O processing routine, directly affecting I/O performance. We chose the SSD + HDD hybrid drive in Section 6.4 to test the performance overhead of the device mapper, since it makes extensive use of the device mapper.

As the hybrid-drive case study of Section 6.4 used a simple mapping, we were able to configure the same hybrid drive using the logical volume manager (LVM) of the Linux OS—the host OS of IOLab. We made 4 GB and 103 GB volumes on the X25-V SSD and the MHZ2120BH HDD, respectively, and combined them to create a 107 GB logical partition using the LVM. We then copied the VM used in Section 6.4 to the logical partition to measure the boot time of Windows XP. Finally, we used the same measurement method described above. The measured OS boot time on the logical partition was 34.61s, while that in Section 6.4 was 34.60s, showing that the overhead of the IOLab device mapper is similar to that of LVM.

## 7 Related Work

Over recent years, significant efforts have been made to develop an efficient evaluation method for storage system research. Table 3 summarizes representative evaluation methods with their advantages as well as limitations in comparison with IOLab. Each method is discussed in detail below.

**Real implementation.** Prototyping intelligence functions on a real system [19], [20] enables a thorough investigation of various implementation issues and their accurate evaluation. However, real system implementations require significant time and effort, thus impeding prompt evaluation of new intelligence functions. Also, the thus-developed prototypes are often not suitable for wide distribution because they are

TABLE 3
Comparison of IOLab with Representative Evaluation Methods

| Evaluation method | Support of target intelligence functions | Performance accuracy | Real-time execution | Developing time |
|---|---|---|---|---|
| Real implementation [19], [20] | not limited | baseline | support | very high |
| Full system simulation [53] | not limited | high | not support | high |
| Device emulation [54] | block-level | high | partially support | moderate |
| Device simulation [17], [18] | block-level | low | not support | moderate |
| File system extension [55], [56] | file-level | moderate | support | very low |
| IOLab | block-level | moderate | support | very low |

tightly coupled with a customized OS and a file system, or even require custom hardware.

**Device simulation.** Storage device simulators [17], [18] have a great deal of flexibility in modeling the internal structure of an active storage device. However, they mostly support only trace-driven simulation, lacking the ability of interacting with real applications. Also, they are unable to account for the data transfer delay between main memory and a storage device, yielding inaccurate evaluation of intelligence functions.

**Full system simulation.** Full-system simulators [53] can execute real applications with a real OS because they model most major components of a computer system in enough detail. They can also simulate I/O connect delays, enabling accurate evaluation of intelligence functions. However, setting up a new target intelligence function on a full-system simulator requires substantial time and effort, which is often comparable to that of real system implementation. Also, full-system simulators are not suitable for getting user experience from daily workloads because they do not support real-time execution.

**Device emulation.** The device emulation approach can overcome the limitations of the trace-driven device simulators discussed above. For example, MEMULATOR [54] extends Disksim [17] to perform timing-accurate emulation of its disk model. MEMULATOR uses a part of main memory as its RAM cache to perform actual data load/store operations. For timing emulation, it inserts an artificial delay before the completion of each I/O request according to the I/O latency calculated by Disksim. This approach allows interaction with real applications as well as real-time execution of intelligence functions. However, this approach is not perfect for the purpose of distributing intelligence functions to users because (1) the data stored in the RAM cache can be lost by sudden power loss; and (2) it cannot maintain real-time execution if the working set size is larger than the RAM cache size, while IOLab does not have such a limitation.

**File system extension.** There have been continuous efforts to facilitate exploring new experimental file systems, such as FUSE (Filesystem in USErspace) [55] and FiST (File System Translator) [56]. FUSE is a userspace file system framework that provides an interface between an OS kernel and an experimental file system running in user space. FiST is a file system generation tool to create a new file system from a standard file system template and new functionalities described using its own template language. Both FUSE and FiST enable rapid development and evaluation of a new experimental file system with much less prototyping effort than developing one from scratch. These tools can also be used to evaluate a specific type of intelligence functions exploiting file-level semantics [12], [32], [33]. However, IOLab is differentiated from the file system extension tools in that it focuses on supporting intelligence functions using block-level information as discussed in Section 4.

## 8 Conclusion

In this paper, we revisited the definition of active storage devices in accordance with their evolution, and introduced IOLab, a VM based platform for the evaluation of intelligence functions of active storage devices. We demonstrated the usefulness and capability of IOLab via a set of case studies that are difficult to prototype in real systems in spite of their obvious benefits. In particular, we have shown that IOLab (1) can evaluate the same intelligence function on different OSes without any modification; (2) can emulate even proprietary intelligence functions; and (3) is useful in rapid prototyping of hybrid drives by combining a set of commodity block devices.

As IOLab supports real-time execution of intelligence functions on VMs, the performance improvement achieved using IOLab is not only useful for researchers but also can be an immediate benefit to users who use VMs for their daily workloads. Using the thus-obtained user experience, IOLab can facilitate the design and testing of intelligence functions for active storage devices, and speeding up their deployment in commodity storage devices and computing systems.

## Acknowledgments

This research was supported by RP-Grant 2011 of Ewha Womans University and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (2011-0013422).

## References

[1] A. Acharya, M. Uysal, and J. Saltz, “Active Disks: Programming Model, Algorithms and Evaluation,” *Proc. 8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 81-91, 1998.
[2] K. Keeton, D.A. Patterson, and J.M. Hellerstein, “A Case for Intelligent Disks (IDISKs),” *ACM SIGMOD Record*, vol. 27, pp. 42-52, 1998.
[3] E. Riedel, G.A. Gibson, and C. Faloutsos, “Active Storage for Large-Scale Data Mining and Multimedia,” *Proc. 24th Int'l Conf. Very Large Data Bases (VLDB)*, pp. 62-73, 1998.
[4] R.D. Chamberlain, R.K. Cytron, M.A. Franklin, and R.S. Indeck, “The Mercury System: Exploiting Truly Fast Hardware for Data Search,” *Proc. Int'l Workshop Storage Network Architecture Parallel I/Os (SNAPI'03)*, pp. 65-72, 2003.
[5] W. Fengguang, X. Hongsheng, and X. Chenfeng, “On the Design of a New Linux Read Ahead Framework,” *ACM SIGOPS Operating Systems Rev.*, vol. 42, no. 5, pp. 75-84, 2008.

[6] Y. Joo, J. Ryu, S. Park, and K.G. Shin, "FAST: Quick Application Launch on Solid-State Drives," *Proc.* USENIX *Conf. File and Storage Technologies (FAST)*, pp. 259-272, 2011.
[7] R. Pai, B. Pulavarty, and M. Cao, "Linux 2.6 Performance Improvement Through Readahead Optimization," *Proc. Ottawa Linux Symp. (OLS)*, pp. 105-116, 2004.
[8] V. Vellanki and A.L. Chervenak, "A Cost-Benefit Scheme for High Performance Predictive Prefetching," *Proc. ACM/IEEE Conf. Supercomputing (SC'99)*, Article no. 50, 1999.
[9] M.E. Russinovich and D. Solomon, *Microsoft Windows Internals*, 4th ed. Microsoft Press, pp. 727-728, 2004.
[10] Apple. *Technical Note TN1150 HFS Plus Volume Format* [Online]. Available: http://developer.apple.com/library/mac/#technotes/tn/tn1150.html2004.
[11] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-ReORGanization for Self-Optimizing Storage Systems," *Proc. Conf. File Storage Technologies (FAST)*, pp. 183-196, 2009.
[12] H. Huang, W. Hung, and K.G. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption," *Proc. 20th ACM Symp. Operating Systems Principles (SOSP)*, pp. 263-276, 2005.
[13] S.W. Ng, "Improving Disk Performance via Latency Reduction," *IEEE Trans. Computers*, vol. 40, no. 1, pp. 22-30, 1991.
[14] Intel. *Intel Turbo Memory with User Pinning* [Online]. Available: http://www.intel.com/design/flash/nand/turbomemory/index.htm, 2008. Accessed on: 17 Nov. 2010.
[15] T. Bisson, S.A. Brandt, and D.D. Long, "NVCache: Increasing the Effectiveness of Disk Spin-Down Algorithms with Caching," *Proc. 10th IEEE Int'l Symp. Modeling, Analysis and Simulation of Computer and Telecomm. Systems (MASCOTS)*, pp. 422-432, 2006.
[16] A.L. Shimpi, *This Just In: Corsair Force 100GB SSD (SF-1200). Anandtech* [Online]. Available: http://www.anandtech.com/show/3654/this-just-in-corsair-force-100gb-ssd-sf1200, 2010.
[17] J.S. Bucy and G.R. Ganger, "The DiskSim Simulation Environment Version 3.0 Reference Manual," Dept. of Computer Science, Carnegie-Mellon Univ., Tech. Rep. CMU-CS-03-102, 2003.
[18] Microsoft. *SSD Extension for DiskSim Simulation Environment* [Online]. Available: http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/default.aspx, Mar. 2009.
[19] H. Payer, M. Sanvido, Z. Bandic, and C. Kirsch, "Combo Drive: Optimizing Cost and Performance in a Heterogeneous Storage Device," *Proc. Workshop Integrating Solid-State Memory into the Storage Hierarchy (WISH)*, pp. 1-8, 2009.
[20] A.M. Caulfield, A. De, J. Coburn, T.I. Mollow, R.K. Gupta, and S. Swanson, "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories," *Proc. 2010 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO'43)*, pp. 385-395, 2010.
[21] M. Wei, L.M. Grupp, F.E. Spada, and S. Swanson, "Reliably Erasing Data from Flash-Based Solid State Drives," *Proc.* USENIX *Conf. File and Storage Technologies (FAST)*, pp. 105-118, 2011.
[22] Seagate. *Data Sheet Momentus XT* [Online]. Available: http://www.seagate.com/docs/pdf/datasheet/disc/ds_momentus_xt.pdf, 2010.
[23] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud, "Intel Turbo Memory: Nonvolatile Disk Caches in the Storage Hierarchy of Mainstream Computer Systems," *ACM Trans. Storage*, vol. 4, no. 2, pp. 1-24, 2008.
[24] X. Wu and A. Reddy, "Managing Storage Space in a Flash and Disk Hybrid Storage System," *Proc. 10th IEEE Int'l Symp. Modeling, Analysis and Simulation of Computer and Telecomm. Systems (MASCOTS)*, pp. 1-4, 2009.
[25] X. Ma and A. Reddy, "MVSS: Multi-View Storage System," *Proc. 21st Int'l Conf. Distributed Computing Systems (ICDCS)*, pp. 31-38, Apr. 2001.
[26] K. Veeraraghavan, J. Flinn, E.B. Nightingale, and B. Noble, "quFiles: The Right File at the Right Time," *Proc.* USENIX *Conf. File and Storage Technologies (FAST)*, pp. 1-14, 2010.
[27] Microsoft. *Windows PC Accelerators* [Online]. Available: http://msdn.microsoft.com/en-us/windows/hardware/gg463388.aspx. Accessed on: 17 Nov. 2010.
[28] F. Chang and G.A. Gibson, "Automatic I/O Hint Generation Through Speculative Execution," *Proc.* USENIX *Symp. Operating Systems Design and Implementation (OSDI)*, pp. 1-14, 1999.
[29] S. VanDeBogart, C. Frost, and E. Kohler, "Reducing Seek Overhead with Application-Directed Prefetching," *Proc. USENIX Ann. Technical Conf. (ATC)*, pp. 299-312, 2009.
[30] C.-K. Yang, T. Mitra, and T.-C. Chiueh, "A Decoupled Architecture for Application-Specific File Prefetching," *Proc. USENIX Ann. Technical Conf. (ATC)*, pp. 157-170, 2002.
[31] M.E. Russinovich and D. Solomon, *Microsoft Windows Internals*, 4th ed. Microsoft Press, pp. 458-462, 2004.
[32] J.A. Garrison and A.L.N. Reddy, "Umbrella File System: Storage Management Across Heterogeneous Devices," *ACM Trans. Storage*, vol. 5, pp. 1-24, 2009.
[33] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim, "FlexFS: A Flexible Flash File System for MLC NAND Flash Memory," *Proc. USENIX Ann. Technical Conf. (ATC)*, pp. 115-128, 2009.
[34] G. Sivathanu, S. Sundararaman, and E. Zadok, "Type-Safe Disks," *Proc.* USENIX *Symp. Operating Systems Design and Implementation (OSDI)*, pp. 15-28, 2006.
[35] A.L. Shimpi, *Seagate's Momentus XT Reviewed, Finally a Good Hybrid HDD. Anandtech* [Online]. Available: http://www.anandtech.com/show/3734/seagates-momentus-xt-review-finally-a-good-hybrid-hdd, 2010.
[36] Z. Li, Z. Chen, S.M. Srinivasan, and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems," *Proc.* USENIX *Conf. File and Storage Technologies (FAST)*, pp. 173-186, 2004.
[37] G. Soundararajan, M. Mihailescu, and C. Amza, "Context-Aware Prefetching at the Storage Server," *Proc. USENIX Ann. Technical Conf. (ATC)*, pp. 377-390, 2008.
[38] M. Sivathanu, V. Prabhakaran, F.I. Popovici, T.E. Denehy, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Semantically-Smart Disk Systems," *Proc.* USENIX *Conf. File and Storage Technologies (FAST)*, pp. 73-88, 2003.
[39] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *Proc.* USENIX *15th ACM symp. Operating Systems Principles (SOSP)*, pp. 79-95, 1995.
[40] F. Chen, D.A. Koufaty, and X. Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives," *Proc. 11th ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 181-192, 2009.
[41] J. Gim and Y. Won, "Extract and Infer Quickly: Obtaining Sector Geometry of Modern Hard Disk Drives," *ACM Trans. Storage*, vol. 6, pp. 6:1-6:26, 2010.
[42] S. Li and H.H. Huang, "Black-Box Performance Modeling for Solid-State Drives," *Proc. 10th IEEE Int'l Symp. Modeling, Analysis and Simulation of Computer and Telecomm. Systems (MASCOTS)*, pp. 391-393, 2009.
[43] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. Ganger, "Storage Device Performance Prediction with CART Models," *Proc. 10th IEEE Int'l Symp. Modeling, Analysis and Simulation of Computer and Telecomm. Systems (MASCOTS)*, pp. 588-595, 2004.
[44] L. Yin, S. Uttamchandani, and R. Katz, "An Empirical Exploration of Black-Box Performance Models for Storage Systems," *Proc. 10th IEEE Int'l Symp. Modeling, Analysis and Simulation of Computer and Telecomm. Systems (MASCOTS)*, pp. 433-440, 2006.
[45] F. Chen, T. Luo, and X. Zhang, "CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory Based Solid State Drives," *Proc.* USENIX *Conf. File and Storage Technologies (FAST)*, pp. 77-90, 2011.
[46] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-Based SSDs," *Proc.* USENIX *Conf. File and Storage Technologies (FAST)*, pp. 91-104, 2011.
[47] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," *Proc. USENIX Ann. Technical Conf. (ATC)*, pp. 57-70, 2008.
[48] B. Hubert, "On Faster Application Startup Times: Cache Stuffing, Seek Profiling, Adaptive Preloading," *Proc. Ottawa Linux Symp. (OLS)*, pp. 245-248, 2005.
[49] A. Singh, *Booting Mac OS X. OSXBOOK.COM* [Online]. Available: http://osxbook.com/book/bonus/ancient/whatismacosx/arch_boot.html, 2003.
[50] S. Wiseman, *Why Windows Takes So Long to Start Up. IntelliAdmin* [Online]. Available: http://www.intelliadmin.com/index.php/2006/09/why-windows-takes-so-long-to-start-up/, 2006.
[51] W.W. Hsu, A.J. Smith, and H.C. Young, "The Automatic Improvement of Locality in Storage Systems," *ACM Trans. Computer Systems*, vol. 23, pp. 424-473, 2005.

[52] J.K. Kim, H.G. Lee, S. Choi, and K.I. Bahng, "A PRAM and NAND Flash Hybrid Architecture for High-Performance Embedded Storage Subsystems," *Proc. 8th ACM Int'l conf. Embedded software (EMSOFT)*, pp. 31-40, 2008.
[53] P. Magnusson et al., "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50-58, Feb. 2002.
[54] J.L. Griffin, J. Schindler, S.W. Schlosser, J.S. Bucy, and G.R. Ganger, "Timing-Accurate Storage Emulation," *Proc.* USENIX *Conf. File and Storage Technologies (FAST)*, pp. 75-88, 2002.
[55] M. Szeredi, *Filesystem in Userspace* [Online]. Available: http://fuse.sourceforge.net, 2005.
[56] E. Zadok and J. Nieh, "FiST: A Language for Stackable File Systems," *Proc. USENIX Ann. Technical Conf. (ATC)*, pp. 55-70, 1999.

**Yongsoo Joo** received the BS and MS degrees in computer engineering and the PhD degree in electrical and computer engineering from Seoul National University, Seoul, Korea, in 2000, 2002, and 2007, respectively. Currently, he is a research professor in the Department of Computer Science and Engineering at Ewha Womans University, Seoul, Korea. His research interests include embedded systems, memory systems, and I/O performance optimization of storage systems.

**Junhee Ryu** received the BS degree in computer engineering from Korea Aerospace University in 2003, and the MS degree in computer science and engineering from Seoul National University, Korea, in 2005. He is currently working toward the PhD degree in computer science and engineering at the same institution. His research interests include file systems, storage systems, and network systems.

**Sangsoo Park** received the BS degree from Korea Advanced Institute of Science and Technology, Daejeon, Korea, in 1998, and the MS and PhD degrees from Seoul National University, Seoul, Korea, in 2000 and 2006, respectively. Currently, he is an assistant professor in the Department of Computer Science and Engineering at Ewha Womans University, Seoul, Korea. His research interests include real-time embedded systems and system software.

**Heonshik Shin** received the BS degree in applied physics from Seoul National University, Seoul, Korea, and the PhD degree in computer engineering from the University of Texas at Austin. He is a professor in the Department of Computer Science and Engineering at Seoul National University. His research interests include mobile systems and software, real-time computing, and storage systems.

**Kang G. Shin** is the Kevin & Nancy O'Connor Professor of Computer Science in the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor. His current research focuses on computing systems and networks as well as on embedded real-time and cyber-physical systems, all with emphasis on timeliness, security, and dependability.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.