# Using Solid-State Drives (SSDs) for Virtual Block Devices

Sang-Hoon Kim
KAIST
sanghoon@calab.kaist.ac.kr

Jin-Soo Kim
Sungkyunkwan University
jinsookim@skku.edu

Seungryoul Maeng
KAIST
maeng@kaist.ac.kr

## Abstract

In a virtualized environment, the block devices on the I/O domain can be provided to guest domains by the virtual block device (VBD). As the VBD incurs low latency and no network access is involved in accessing data, VBD has been used for storing intermediate data of data-intensive applications such as MapReduce. To accelerate the performance further, SSD can be considered as a backing device of the VBD because SSD outperforms HDD by several orders of magnitude. However, the TRIM command that plays an important role in space management of SSD has not been discussed nor handled properly in the virtualized environment.

In this paper, we clarify the challenges in supporting the TRIM command in the VBD configuration and propose a novel file operation, called FTRIM (file trim). The FTRIM bridges the semantic gaps among key components of VBD and makes them to be aware of the file deletion in the guest domain. As a result, the each component of VBD can utilize the space of the deleted files, and the SSD in the I/O domain can be notified by the TRIM command. Our evaluation with a prototype shows that the proposed approach achieves up to 3.47x speed-up and sustains the improved performance in the consolidated VM environment.

***Categories and Subject Descriptors*** D.4.2 [*Operating Systems*]: Storage Management; D.4.3 [*Operating Systems*]: Filesystems Management; D.4.7 [*Operating Systems*]: Organization and Design

***General Terms*** Design, Performance, Measurement

***Keywords*** SSD, trim, virtual disk image, storage, filesystem

## 1. Introduction

Virtualization and cloud computing have been receiving substantial attention over the past decades. The concept of virtualization facilitates elastic resource management such as VM (virtual machine) consolidation and on-demand VM instance allocation. The success of Amazon Web Service (AWS) and Elastic Compute Cloud (EC2) demonstrates the flexibility and versatility of cloud computing in the IT industry.

In a virtualized environment, a guest domain has a number of choices for its storage system. Among those options, using a virtual block device (VBD) has been the primary option for the root filesystem and the scratch disk of data-intensive applications because of its simplicity, mobility and ease of management. A hypervisor provides a VBD to a guest domain, and block requests to the VBD are delivered to the I/O domain[1]. The requests are then translated to a set of file operations to a virtual disk image (VDI) file which is stored in the I/O domain filesystem. A number of formats for the VDI file have been proposed such as VMware VMDK(Virtual Machine DisK) [36], VirtualBox VDI [33], Microsoft Virtual Server VHD (Virtual Hard Disk) [24], and QEMU QCOW2 [22].

In the meantime, NAND flash memory (or flash for short) becomes popular storage media. Solid-state drives (SSDs) use the flash and provide the traditional block I/O interface such as SATA or SAS. SSDs outperform hard disk drives (HDDs) by several orders of magnitude because SSDs access data electronically and do not have mechanical parts which are the primary source of latency in HDDs. Also SSDs inherit many attractive characteristics from flash, such as light weight, low energy consumption, shock resistance, and small form factor. Note that modern SSDs support the "TRIM" command which notifies SSDs that the specified sectors are no longer needed. SSDs can use the hint to manage the underlying flash media more efficiently. It is known that the TRIM affects the performance and the lifetime of SSD greatly because it effectively increases the amount of the internal buffer, which is an important parameter for SSD performance [30].

There have been many discussions on how to deploy SSDs to accelerate I/O-intensive applications [2, 4, 14, 16, 17]. However, using SSDs in a virtualized environment is not trivial. As Figure 1 shows, the filesystem in the guest domain interacts with the VBD over the block interface and the VDI emulates block requests on top of the I/O domain filesystem. Even if a file is deleted from the guest domain filesystem, the event of the file deletion cannot be delivered to the VBD, and the VBD cannot instruct the I/O domain filesystem to issue the TRIM commands to the underlying SSD. Thus, the space associated with the deleted file cannot be reclaimed, and the SSD cannot be informed by the TRIM commands neither. To the best of authors' knowledge, the use of the TRIM command is not discussed in a virtualized environment, and no hypervisor nor VBD implementation handles the TRIM command properly. Consequently, the performance of SSDs cannot be fully utilized in a virtualized environment

In this paper, we identify the challenges and advantages of handling the TRIM command in a VBD configuration. We study the key components of VBD and observe that the VBD driver does not advertise the TRIM capability so that the guest domain generates no TRIM command to the VBD. We also find out that the current filesystems are incapable of emulating the TRIM command because none of filesystem operations matches to the semantics of TRIM. We introduce a new file operation called "*file trim*" or



*RESoLVE '12* 3 March 2012, London, UK



[1] We refer to the I/O domain as the privileged domain or VM which is designated to perform the actual I/O, such as the Xen Isolated Driver Domain (IDD) or the host OS of the Linux KVM.

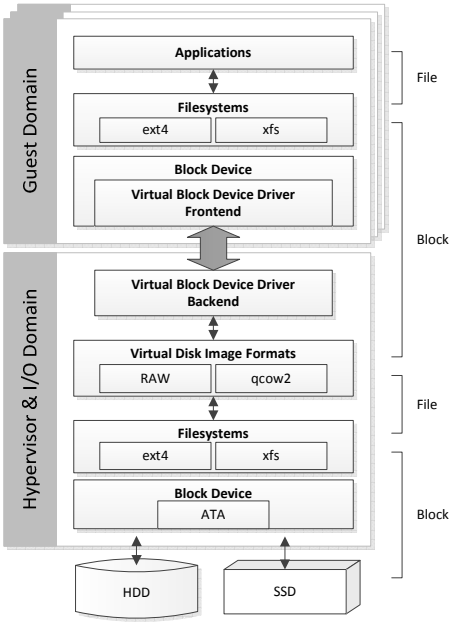



**Figure 1.** System components in a guest domain and the I/O domain for a virtual block device (VBD)

FTRIM. It is a file-level counterpart of the TRIM command and notifies a filesystem of an unnecessary portion of a file. By means of the FTRIM command, VBDs can interact with the I/O domain filesystem and the filesystem can issue TRIM to the corresponding sectors on SSD in turn. In this way, the performance of SSDs in a virtualized environment can be maximized. FTRIM also gives a hint to the I/O domain filesystem that the portion of the file can be deallocated and reclaimed, which can significantly improve the efficiency and flexibility of space management in the I/O domain.

The rest of this paper is organized as follows. We review the background and related work on storage systems for a virtualized environment and SSDs in Section 2. We explain our motivation and approach to handling the TRIM command in Section 3. Section 4 shows the evaluation results of the proposed approach using a working prototype and real-world benchmarks. Finally, Section 5 concludes the paper.

## 2. Background and Related Work

### 2.1 Storage Systems for a Virtualized Environment

In a virtualized environment, a number of storage systems and configurations can be considered for guest domains. Using the networked or distributed filesystems [6, 28, 35] is one of the options. They allow transparent and seamless view of storage over network even if a guest domain might be migrated to other locations, and store data reliably by handling faults which are the norm rather than the exception [6] in a large scale system. The I/O domain can offload the burden of maintaining reliable storage environment to the filesystems and focus on virtualizing resources. However, storage access may incur high latency because network access might be involved in accessing data, and the operational cost can be expensive as commercial cloud service providers charge not only for the storage space but also for the network traffic incurred while accessing the storage [1]. Thus, this option is suitable for storing application's source data or processed results which should be kept reliably.

Using *virtual block devices (VBDs)* is the other storage option for guest domains. The hypervisor provides a block device to a guest domain, and the guest domain uses the device as if the device were a usual block device which is attached directly to the guest domain. Requests to the block device are translated by the hypervisor and backed by a file in the I/O domain filesystem. As the configuration is simple and no network access is involved in accessing the storage, VBD has been the primary storage option for the root filesystems of guest domains and intermediate data of data-intensive applications.

The file storing the contents of VBD is called the *virtual disk image (VDI)*. A request to VBD is translated to one or more VDI file operations by the VDI-specific translation scheme. The request expressed in virtual block address (VBA) is translated to virtual file address (VFA). Note that VBA is expressed by a tuple of the sector and length in the VBD, whereas VFA is represented by a tuple of the offset and length in the VDI file.

RAW is the basic VDI format which is usually provided by hypervisors. Address translation between VBA and VFA is straightforward — the offset from the beginning of the block device is the offset in the VDI file. The translation is same to the loopback device. It is simple yet effective in many cases. But storage space must be allocated at creation time[2], and the common data between VDI files cannot be shared even if they are stored in the same I/O domain. Thus, the efficiency and flexibility of space management is limited.

For the sophisticated space management, a number of VDI formats have been proposed by industries and open source communities. Several examples include VMDK(Virtual Machine DisK)[36] for VMware, VDI [33] for VirtualBox, VHD (Virtual Hard Disk) for Microsoft Virtual Server, and QCOW2 [22] for QEMU. All of them are based on the *copy-on-write (COW)* capability to manage the space efficiently. A VDI *data file* is created instantly from a *template image* which can be shared by several data files. A VBD is virtualized on top of the data file, and writing to the VBD is served from the newly created data file rather than the template image. Read requests to the written data are served from the data file. Otherwise, it is served from the template image. The data file only stores the changes made after the creation time. This approach improves the space efficiency, and the small size of the data file lowers the overhead of VM instance migration [19].

QCOW2 [22] is a copy-on-write VDI file format which is supported by QEMU. It is an enhanced version of the QCOW format and has been used as an I/O framework in KVM and Xen-HVM. QCOW2 data file can be "chained" to backing image files. Block address space is managed by two-level address translation which is similar to the radix tree in page table. If an entry which corresponds to the requested block address is invalid, QCOW2 allocates space for the block at the end of the data file. Space for the translation index is also allocated at the end of the data file on demand.

Address translations in VBD makes the layout of the address space in a guest domain not to coincide with the actual layout in a storage device, and nullifies the "unwritten contract" [29] previously made between applications and filesystems. The locality in the VBA space might not be preserved in the VFA space, which harms the performance of VBD. Tang [32] identifies the locality issues of VBD and suggests Fast Virtual Disk (FVD) which tries to preserve the locality by separating dirty-block tracking and storage space allocation. FVD also deploys a copy-on-read policy which copies the read blocks from a template file (copy-on-read), and an adaptive prefetching. But FVD cannot understand the filesystem semantics of the guest domain which lowers the space utilization.

[2] If the filesystem which stores the VDI file supports the sparse file, the space allocation can be deferred until an actual write happens.

There have been discussions in optimizing the storage systems in the I/O domain. VMware VMFS [37] is a proprietary block-level storage virtualization system for VMware ESX. Although the internal details of VMFS are not available publicly, it is known to be a cluster filesystem tuned to host image files in VMDK format efficiently. Parallax [23] is a distributed storage system which proposes the similar approach to VMFS and attempts to provide advanced storage services for virtual machines. However, both VMFS and Parallax implement at block device layers, which have no knowledge of the filesystem semantics, hence, the optimization is limited. Ventena [25] is a virtualization-aware filesystem which effectively virtualizes filesystem namespace and shares the contents of files if possible. However, it forces guest domains to use the Ventana filesystem instead of the general filesystems such as ext4.

### 2.2 Flash Memory and Solid-State Drives (SSDs)

NAND flash memory, or flash, is non-volatile solid-state storage media. Flash is comprised of a number of *pages*, where the page is the unit of read and program (write) operations. The size of the page depends on the type and configuration of flash, and ranges from 2 KB to 8 KB usually. Flash has very unique characteristics. Most of all, the flash does not allow overwriting. If a page is programmed once, the page must be erased prior to be written again. The unit of the erase operation is called *erase block* and it is composed of multiple pages. The erase operation produces a *free block* which contains pages that can be written again. Flash wears as the erase blocks are programmed and erased repeatedly. If the number of programing / erase cycle of an erase block exceeds a threshold, the probability of read or program failure increases, and the block finally becomes unable to be programmed nor read [5]. Flash exhibits asymmetric operation time for read and programming operations. Programming a page takes several orders of magnitude longer than reading a page, and erasing a block takes even longer than the page programming. Table 1 summarizes the operation times and parameters of flash memory chip that is available in the market.

A solid-state drive (SSD) is a storage device that uses solid-state memory to store persistent data [38]. As most of SSDs in the market are built with flash because of its high density and low cost, we will focus on flash-based SSDs in this paper. SSDs outperform hard disk drives (HDDs) because they do not have any mechanical parts such as rotating disk or moving arms, which are the major sources of I/O latency in HDDs. Modern SSDs exhibit excellent random read performance which is comparable to sequential read performance. Both the sequential and random write throughput is slower than the sequential and random read throughput, yet much better than the throughput of HDDs. The excellent performance of SSDs makes them suitable for data-intensive workloads [2, 4, 14, 16**?** , 17].

Figure 2 depicts the general architecture of SSD. As flash does not allow in-place update, the flash interface in SSD cannot be directly exported to the host interface. To hide the limitations of flash and to improve space management, SSD employs an intermediate layer, called flash translation layer (FTL). FTL lies between the host interface controller and flash controllers, and manages the space of the flash. A block operation from the host interface is translated to a set of flash operations, and sent to one or more flash controllers. FTL maintains mappings between block address space (sector) and flash address space (erase block and page), and translates overwrite to the block address space to write to the free blocks. The pages that were written once but will not be used anymore are called *invalid pages*. FTL reclaims the invalid pages by erasing the containing block (*victim block*). The series of flash operations to reclaim invalid pages is called *garbage collection*. Many FTLs have been proposed and discussed in the literature [3, 7, 13, 18], which aim at maximizing the performance and lifetime of SSDs by exploiting the inherent characteristics of flash and/or managing address translation efficiently.

SSDs make use of overprovisioning [30]. SSDs are equipped with extra amount of space that is not visible to outside of the SSDs. The space is used internally as the working space for garbage collection, the buffer to absorb bursts of write, and the reserved space for bad blocks. The unused flash blocks in the SSDs are also utilized as the overprovisioning space internally. It is known that the overprovisioned space reduces the write amplification and extends the lifetime of SSDs [9].

TRIM is an ATA command standardized as part of the AT Attachment (ATA) Interface standard [10, 11], led by Technical Committee T13 of the International Committee for Information Technology Standards (INCITS). TRIM notifies SSD that the specified sectors are no longer needed by filesystems or operating systems. SSD can invalidate the corresponding pages and avoid unnecessary copy of the invalid pages while performing garbage collection. Consequently, TRIM enhances the garbage collection efficiency and increases the effective amount of overprovisioning in SSD. Modern operating systems and filesystems [21, 31] enable the use of TRIM commands if the underlying SSD supports the TRIM capability.

| Parameter | Value |
|---|---|
| Page size | 4096 bytes for data |
| Block size | 512 KB (128 pages per block) |
| Page read time | 60 $\mu$s (maximum) |
| Page program time | 0.8 ms (typical) |
| Block erase time | 1.5 ms (typical) |
| Endurance | 5K program/erase cycles (with 4bit per 512 bytes ECC) |

**Table 1.** Operational parameters of Samsung NAND flash memory (K9GAG08U0M) [27]

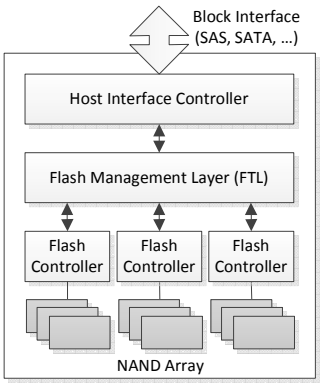



**Figure 2.** General architecture of flash-based SSD

| `int` **ftrim**(`int` **fd**, `uint` **offset**, `uint` **length**) | |
|---|---|
| `int` **fd** | File descriptor of the target file |
| `uint` **offset** | Offset from the beginning of the file to FTRIM |
| `uint` **length** | Length to FTRIM |

**Table 2.** The interface and arguments of the FTRIM (file trim)

## 3. TRIM in a Virtualized Environment

### 3.1 Motivation

The performance of data-intensive applications is known to be heavily affected by the performance of the storage device for their intermediate data [15]. As the virtual block device (VBD) incurs low overhead in accessing data, using the VBD is the simple yet effective solution to store the intermediate data. To accelerate data-intensive applications even further, the VBD can be backed by SSDs that outperforms HDDs by several orders of magnitude. The intermediate data lives shortly, and file deletion to the filesystem is frequent. Ideally, if a file is deleted from a guest domain, its corresponding space in the VBD has to be reclaimed and the TRIM commands have to be delivered all the way to the underlying SSDs. However, the VBD is agnostic to the filesystem semantics. The guest domain cannot deliver the filesystem semantics the VBD, and the VBD cannot reclaim the space of the deleted file. Also, the VBD is not implemented to handle the TRIM command, which leads to the guest domain filesystem not to issue TRIM to the VBD. Consequently, TRIM is not issued to SSD, and the performance of SSD cannot be fully utilized. In fact, none of the well-known hypervisors handles the TRIM command properly.

We revise storage system components of VBD in order to propagate the event of a file deletion from a guest domain filesystem all the way to the SSD attached to the I/O domain. We also suggest optimizations that can be adopted to improve the performance and efficiency of storage in a virtualized environment. Figure 3 depicts the propagation of a file deletion event from a guest domain filesystem to the SSD and address translations performed at each layer. The figure shows that a single file deletion may issue a number of requests to the underlying layers, and some of them can be merged along the way. Note that the label on the leftmost side denotes the unit of the request at each layer.

### 3.2 Key Components

The key components and their roles in virtualizing the block device are as follows:

- **Virtual block device (VBD) driver.** Provides an interface to VBD and conveys requests to the VBD to the backing VDI layer. The interface shows a virtual block address (VBA) space to a guest domain. If the driver is implemented as a split driver, it consists of a frontend and a backend driver, and they are located at the guest domain and the I/O domain, respectively.
- **Virtual disk image (VDI) layer.** Determines the layout of VBD contents on a VDI file in the I/O domain filesystem. The VDI layer dispatches block requests from the VBD (backend) driver and translates the requests to file operations to the VDI file. The file operations are expressed in virtual file address (VFA) which is a tuple of the offset and length in the VDI file.
- **Filesystem in the I/O domain.** Manages a block address space and maps the VDI file operations to block requests to the SSD. The block address space is expressed in sector. If the SSD supports TRIM, the filesystem composes and issues TRIM requests to the SSD.

### 3.3 Virtual Block Device (VBD) Driver

Traditional VBDs which are provided by the well-known hypervisors emulate HDDs and do not advertise the TRIM capability. Thus, the filesystem in a guest domain does not issue TRIM to VBDs, and consequently no VBD implementation handles the TRIM command. The TRIM capability of VBD can be easily turned on by advertising the TRIM capability with the following three parameters:

- **Granularity.** The granularity of a TRIM request. The offset in the TRIM request needs to be aligned to this granularity, and the length in the TRIM request should be expressed in a multiple of this granularity. It forces the allowed length of the TRIM request to be equal to or greater than the granularity.
- **Maximum length.** The maximum length of a single TRIM request the block device supports. The value is expressed in the unit of the granularity.
- **Deterministic read.** This indicates whether or not the deterministic read is supported. The deterministic read means that a read on a trimmed block is guaranteed to be same until a subsequent write to the block happens. Otherwise, the read contents of the block are undefined.

A guest domain operating system will query the TRIM capability of the VBD while initializing the VBD driver and will make use of the capability. The parameters are used to compose a TRIM request to the VBD. For VBD, it is sufficient to pick some reasonable values of the parameters since the TRIM request will be processed again by subsequent translations so that the final requests to the SSD will conform to the actual parameters of the SSD.

The VBD driver needs to convey the TRIM request from the guest domain to the I/O domain. Due to the simplicity of the TRIM request which is only a tuple of the block number and length, the TRIM request can be delivered just the same as the ordinary read or write block requests.

### 3.4 Virtual Disk Image (VDI) Layer

Handling TRIM in the VDI layer is a bit more tricky than in the VBD driver. As we discussed in Section 3.2, the VDI layer dispatches a block request from a VBD driver, translates the destined blocks in VBA to VFA, and emulates the request on a VDI file. A read and write request from the VBD backend can be emulated with a number of read and write file operations to the VDI file easily. However, a TRIM request cannot be emulated in an obvious way because none of filesystem operations matches to the semantics of TRIM. The absence of the interface between the VDI layer and filesystems in the I/O domain incurs semantic gap and blockades the event propagation of the file deletion.

To bridge the semantic gap between VDI and the filesystem, we propose a new file operation, called "*file trim*" or FTRIM. FTRIM is a file-level counterpart of TRIM and notifies a filesystem that *a portion of the file is no longer needed.* Note that the TRIM command is defined at block-level and implies that the specified blocks are no longer needed. Table 2 summarizes the interface and arguments of the proposed FTRIM operation. Figure 3 captures the meaning of FTRIM in the VBD architecture. Each arrow between VDI and the I/O domain filesystem indicates the associated FTRIM operation which enables the interaction between VDI and the I/O domain.

The VDI layer translates the TRIM requests in VBA to VFA of a VDI file. The translation can be carried out just same as the ordinary read and write block requests. As a VDI layer might employ a VDI-specific space allocation policy such as copy-on-write, one TRIM request can be split and mapped to a number of FTRIM invocations scattered over the VDI file. The number of invocations can be minimized by merging adjacent FTRIM requests.

Without TRIM, the VDI layer has no knowledge of a file deletion in the guest domain, and the space associated with the file cannot be reclaimed or deallocated even if the file is deleted. It restricts the deployment of efficient space management and optimizations. As the VDI layer is informed by TRIM from the VBD driver, two optimizations can be considered to improve the efficiency and performance of the VDI layer. First, the VDI layer can unmap the

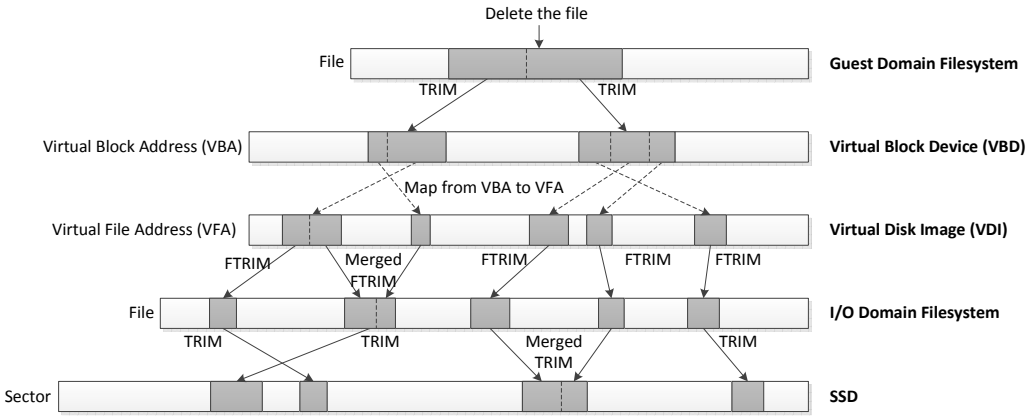



**Figure 3.** The propagation of the file deletion event from a guest domain filesystem to SSD, and the required address translations. The "Merged FTRIM" and "Merged TRIM" denote that the two requests are merged and issued by one request.

deleted blocks and free the space which previously stored the mapping information. It enables the VDI layer to keep a compact view of the VBA space and reduces the overhead to keep the mapping information. Second, VDI can utilize TRIM to improve the locality of VBA space in VFA space. The mapping from VBA space to VFA space was determined at allocation time, and the mapping was permanent so that the space for the VBA could not be relocated. If the locality of VBA space in VFA space was broken once, it could not be recovered without a time-consuming offline optimization process. As the VDI layer becomes aware of the unnecessary portion of VBA, VDI can reorganize the VBA space by migrating a portion of the VDI file to the trimmed location and incrementally improve the locality of VBA in the VFA space.

### 3.5 Filesystem in the I/O Domain

A filesystem receives a FTRIM request from the VDI layer, identifies the sectors which correspond to the FTRIM request, composes TRIM requests to the sectors, and issues the requests to the SSD. One FTRIM can be transformed to a number of TRIM requests because of the space allocation policy of the filesystem as Figure 3 depicts. The generated TRIM requests can be merged if they are destined to adjacent blocks.

By means of FTRIM, the filesystem becomes capable of identifying an unnecessary portion of a file. The filesystem can utilize the information in block space management just same as SSD utilizes the TRIM command in flash space management. The filesystem can deallocate the space where the portion of the file has occupied, and mark the space as free. Effectively, the space usage of the file is decreased and the free space of the filesystem is increased. It enables the VDI file to be shrunk while being used. The feature is extremely useful in the consolidated environment because the size of the VDI file is adjusted on-the-fly and the space overhead is minimized. Note that the current architectures of VBD only allow the size of the VDI file to monotonically increase as the file is being used.

The file deallocation can be implemented with the "file hole". The file hole is an area in a sparse file where the space for the area is not actually allocated. The file hole can be usually created by writing data beyond the end of the file or truncating the file to be larger than the current size. The hole does not occupy filesystem space and the actual space allocation for the hole is deferred until a write to the hole happens. The file deallocation can be thought as of the reversed procedure of the space allocation of the file hole. Filesystem can unmap the specified blocks, and reclaim the space for the blocks. The subsequent access to the portion of the file can be treated just same as the ordinary file hole. The deallocation might accompany a filesystem metadata manipulation. A filesystem keeps tracks of free blocks on a block device and the file deallocation inevitably manipulates the metadata. Thus, recovery from a sudden crash must be considered. We believe that the journaling or write-ahead-logging is sufficient for the preparation, just like a file deletion. If handling the file deallocation is too much burden for a filesystem, the filesystem can simply do nothing but issue TRIM to the SSD. If the SSD supports the deterministic read, this approach does not harm the consistency of the filesystem at all. Moreover, the FTRIM interface can be extended by adding an argument which controls the behavior of FTRIM so that the TRIM and file deallocation can be performed selectively.

## 4. Evaluation

### 4.1 Prototype Implementation

We have implemented a prototype on a real-world environment to evaluate the effectiveness of the proposed approach. The prototype is built on the Linux KVM (Kernel Based Virtual Machine) which uses the QEMU 0.14.1 and the Linux Kernel 2.6.39. Our implementation includes the QEMU virtio_blk virtual block device driver, the QEMU block driver layer, RAW and QCOW2 virtual disk images, and the ext4 filesystem.

The QEMU virtio_blk driver is a paravirtualized block device driver provided by the QEMU virtio framework [26]. The virtio_blk is a split driver comprised of a frontend and a backend. The frontend provides a VBD to a guest domain and delivers block requests to the VBD to the backend in the I/O domain. We modified the frontend so that the VBD advertises the TRIM capability. It is configured with the realistic parameters obtained from Intel X-25M SSD — 512 bytes of the trim granularity and 4,294,966,784 bytes (0xFFFFFFFF - 512) of the maximum length with no support for the deterministic read. TRIM requests to the frontend are conveyed to the backend by the virtio framework, and dispatched by the QEMU block driver layer. The QEMU block driver layer interacts with a number of VDI implementations which are abstracted to the unified `bdrv` block driver handler interface. We define a new `bdrv` handler (`bdrv_trim`) which is designated to handle the TRIM requests to the virtio_blk device. The handler has the responsibility to translate the TRIM requests to the FTRIM requests according to the VDI-specific space management policy. We implemented the `bdrv_trim` handler in two VDI formats, RAW and QCOW2.

In RAW, the target offset and the length of a FTRIM request is obtained by multiplying the block number of TRIM by the

trim granularity of the VBD. As RAW does not employ any space management policy, one TRIM is converted to a single FTRIM request.

In QCOW2, obtaining the target offset and the length of FTRIM becomes complicated. QCOW2 manages the address space in an allocation unit basis (64 KB by default). The blocks in the TRIM request are divided by the unit into segments, and the segments are mapped individually according to the QCOW2 address translation scheme. To minimize the number of the FTRIM invocations, the mapped addresses are buffered and merged if two segments are mapped to adjacent addresses. The buffered addresses are flushed to the I/O domain filesystem when the entire blocks in the TRIM request are mapped.

Filesystems translate the FTRIM requests from the handlers to TRIM commands to the underlying SSDs. We define the FTRIM operation as one of the Linux VFS (Virtual Filesystem) file operations and filesystems can implement its own FTRIM operation selectively. We implement the FTRIM operation (`ext4_ftrim()`) on the ext4 filesystem. The offset and the length of the FTRIM request is converted to a block address on the SSD, and delivered to `blkdev_issue_discard()` kernel function which composes and issues the actual TRIM requests to the underlying devices. Note that one FTRIM request can be split into several TRIM requests by the ext4 space management policy. If possible, they will be merged in the I/O scheduler of the Linux kernel.

For fast prototyping, we implemented the proposed scheme only on the two representative VDI formats (RAW and QCOW2) and the ext4 filesystem. However, we believe that the other VDI formats and filesystems can easily adopt the proposed scheme because it is concise and straightforward. We do not implement the space management and optimizations in the filesystem and VDI layer, and we leave them as future work.

### 4.2 Environment and Methodology

Using the working prototype, we have conducted several experiments to justify the effectiveness of the proposed approach. The evaluation is performed on a server machine equipped with one Intel Core i7-870 2.93GHz CPU and 16 GB RAM. We use the Intel X25-M G2 MainStream SATA 120 GB SSD throughout the evaluation.

As SSDs are easily affected by recent usage pattern, we initialize the SSD before every run of evaluations as follows. We issued TRIM requests to the entire SSD block address. Then, we build the ext4 filesystem on it, and mount the filesystem to the I/O domain. To trigger the garbage collection in the SSD quickly, the filesystem is filled by a file containing zeroes until no subsequent write is possible, then the file is deleted. Finally, the SSD was left idle for 30 seconds to settle down internal activities.

Our evaluations are performed on the following three configurations:

- **HOST.** The evaluation is carried on the I/O domain. The SSD is not virtualized and accessed directly.
- **RAW.** The SSD is virtualized by the RAW VDI format. As RAW employs a simple address translation in the VDI file, it represents the baseline of the performance in the virtualized environment.
- **QCOW2.** This configuration uses the QCOW2 VDI format. It represents the VDI which employs an address translation and provides the copy-on-write capability.

RAW and QCOW2 use the ext4 filesystem as the guest domain filesystem. The TRIM capability is controlled by the 'discard' mount option of the ext4 filesystem. In RAW and QCOW2, the option is applied while mounting the filesystem in the guest domain. In HOST, the option is applied while mounting the filesystem in the I/O domain. We will refer to *"baseline"* and *"trimmed"* as the configurations where the TRIM capability is turned off and turned on, respectively. In the virtualized configurations (RAW and QCOW2), each guest domain is configured to use 2 CPUs and 1 GB of RAM. The cache of the VDI file is turned off to minimize the effect of the page cache in the I/O domain.

### 4.3 Trim Latency

First, we measure the latency of issuing the TRIM command. We use a synthetic micro-benchmark which issues TRIM commands to random locations of the SSD and records the latency to complete the request. The latency is measured by varying the length of the request.

We find that a single TRIM takes about 1.5 ms regardless of the length of the request. We have also measured the overhead of the QEMU block driver, the VDI handler, and the ext4 filesystem. However, it turns out to be about 1 $\mu$s which is negligible compared to the TRIM latency.

We conclude that the overhead of TRIM is mostly dominated by the number of the TRIM requests rather than the size of the requests. Thus, merging adjacent TRIM requests can reduce the overhead significantly.

### 4.4 Benchmark Results

To evaluate the primary effect of TRIM and FTRIM with simple workloads, we use two benchmarks. Postmark [12] gives the fundamental performance metrics of storage systems under the mailbox workload. It fills the filesystem with a specified number of files, then iterates a number of transactions. Each transaction is chosen among read, delete, or write operations. Compilebench [20] measures the performance of storage system while aging the system. The storage is filled with a predefined number of the Linux kernel source tree, and file operations which create, patch, compile, and clean the source tree are replayed. It generates a huge number of file operations to the filesystem and makes the filesystem get aged.

These benchmarks are configured to use the entire SSD space because sectors belonging to untouched areas can be used as the overprovisioning space inside the SSD which may affect the overall performance.

#### 4.4.1 Postmark

Postmark is configured to fill 96 GB of the filesystem with initial files, and to iterate transactions twice as much as the number of the initial files. The buffered I/O is turned off and the other parameters including the create bias are unchanged. We evaluated the performance by varying the file size from 128 KB to 4 MB. Figure 4 summarizes the evaluation results. Note that the value represents the write bandwidth of the trimmed configuration normalized to its baseline configuration (the higher, the better).

The trimmed configurations outperform the baseline configurations regardless of the file size. HOST, RAW and QCOW2 configurations improve the performance by 78%, 52%, and 32% respectively. We can verify that TRIM affects the SSD performance in the virtualized environment as well as in the native environment. QCOW2 shows diminished benefits compared to RAW. The performance gap mostly comes from the QCOW2 allocation policy which generates more TRIM requests than the RAW configuration by dividing a single TRIM request into several TRIM requests.

#### 4.4.2 Compilebench

We configure Compilebench to populate 300 instances of the Linux kernel source tree which occupy about 86 GB of the SSD. Then, 600 transactions are replayed on the source trees while measuring the elapsed time of each transaction. The transaction is followed

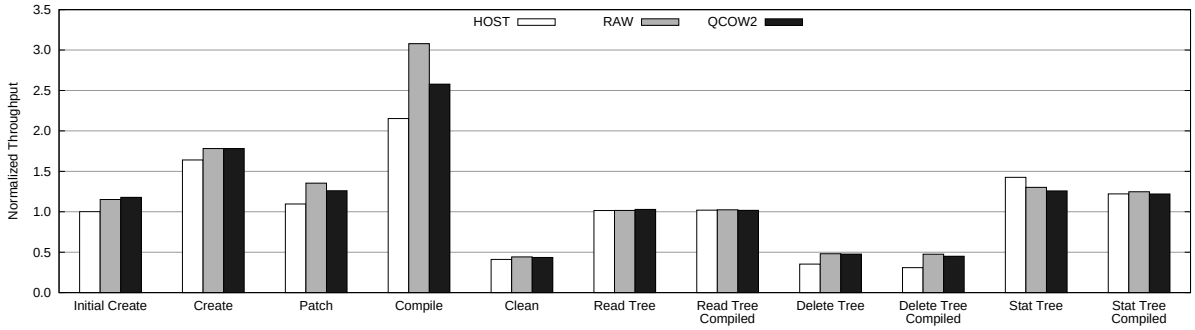



**Figure 5.** Compilebench results. Note that the value is the throughput of the trimmed configuration normalized to its baseline configuration.

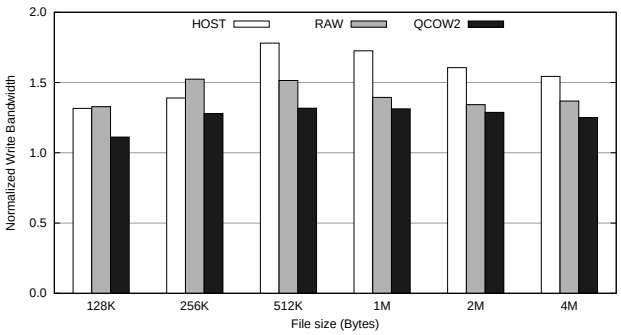



**Figure 4.** Postmark results. Note that the value is the write bandwidth of the trimmed configuration normalized to its baseline configuration.

by invalidating the page cache to minimize the effect of the page cache. Figure 5 shows the results of Compilebench. Note that the value is the performance of the trimmed configuration normalized to its baseline configuration.

For the write-dominant transactions, the trimmed configurations outperform the baseline configurations. For the compile transactions, the trimmed configurations of HOST, RAW, and QCOW2 achieve 115%, 207%, and 157% better performance than the baseline configurations, respectively. For the read-dominant transactions, the trimmed configurations show comparable or better performance than the baseline configurations. However, the trimmed configurations show worse performance than the baseline configurations for the delete-dominant transactions because many TRIM requests are involved in the transactions.

The benefit of TRIM in the QCOW2 configuration is lower than or comparable to the RAW configuration. The reason is same to that of the Postmark results — the space management policy of QCOW2 can split a single TRIM requests into many TRIM requests, resulting in more TRIM requests than the RAW configuration.

### 4.5 Hadoop MapReduce

To see the effect of the TRIM command in a consolidated VM environment, we have conducted an evaluation with the Hadoop MapReduce platform [34, 35]. We set up a Hadoop MapReduce environment on the evaluation machine with the Apache Hadoop v0.21. HDFS is configured to run a namenode on the I/O domain and four datanodes on four VM instances on the same machine.

| Application | Operation | Map (MB) | Reduce (MB) | Total (MB) |
|---|---|---|---|---|
| WORDCOUNT | Local Read | 9,850 | 24,462 | 32,312 |
| | Local Write | 19,858 | 24,538 | 44,396 |
| | HDFS Read | 10,523 | 0 | 10,523 |
| | HDFS Write | 0 | 1,518 | 1,518 |
| SORT | Local Read | 0 | 41,819 | 41.819 |
| | Local Write | 20,718 | 41,819 | 62,537 |
| | HDFS Read | 21,024 | 0 | 21,024 |
| | HDFS Write | 0 | 21,023 | 21,023 |

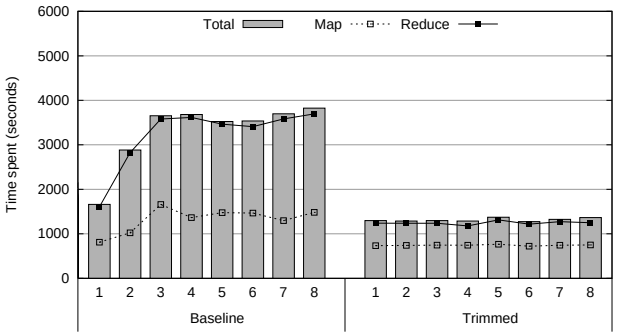
**Table 3.** The amount of storage access while running the MapReduce application.

Each VM is configured to use 2 CPUs and 1 GB of RAM. MapReduce is configured to use 2 map slots and 2 reduce slots per VM instance. A VBD whose VDI file is stored in the SSD is provided to each VM, and the MapReduce uses the VBD as the temporary local storage. We run eight runs in a row and measure the time to complete the map tasks, the reduce tasks, and the job separately.

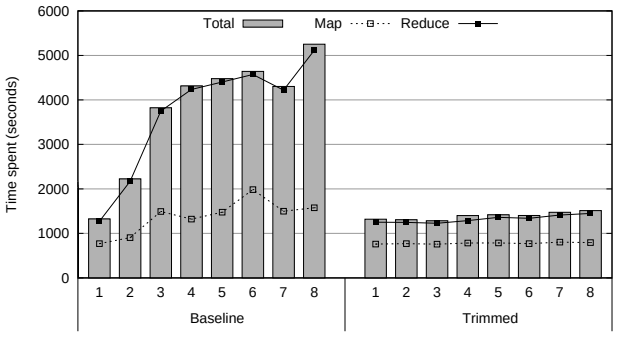
We evaluated two MapReduce applications, WORDCOUNT and SORT. The WORDCOUNT application represents a typical MapReduce application while the SORT application represents a more write-intensive application. Table 3 summarizes the amount of storage access for each application. Note that MapReduce applications can incur heavy storage access to the local storage which is provided by the VBD.

For WORDCOUNT, we collect 10 GB of text from a department mailbox, and put them into HDFS. Then, we run the "wordcount" application which is one of the default examples of the Hadoop MapReduce. The results for the first eight runs are depicted in Figure 6. The total running time of WORDCOUNT depends on the running time of the reduce phase. The reduce phase is comprised of copying, shuffling, and writing phases, and massive local storage access is observed during the copying and shuffling phases. Hence, the access to the local storage influences the running time of the reduce phase.

The performance of the baseline configuration degrades as we run the WORDCOUNT application repeatedly. It is mainly due to the degraded garbage collection efficiency of the SSD. As the space occupied by deleted files cannot be reclaimed, the effective amount of the overprovisioned space in the SSD is decreased and it impairs the performance of the garbage collection. Meanwhile, the trimmed configurations of both RAW and QCOW2 outperform the baseline configurations, and show sustained performance. At the eighth run, the trimmed configuration shows 2.80x and 3.47x speed-up

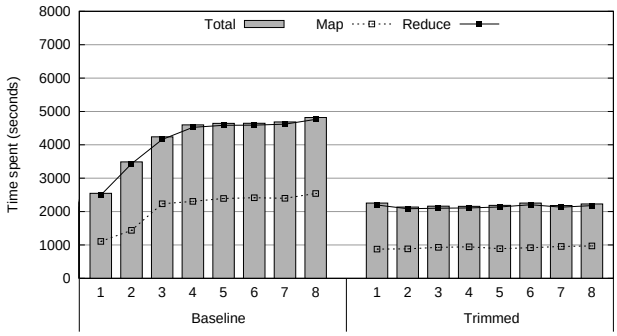



(a) WORDCOUNT on RAW

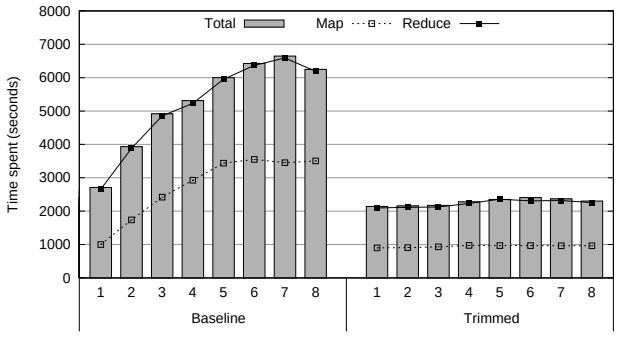



(b) WORDCOUNT on QCOW2

**Figure 6.** Performance of the WORDCOUNT MapReduce application on various VDI formats.




(a) SORT on RAW




(b) SORT on QCOW2

**Figure 7.** Performance of the SORT MapReduce application on various VDI formats.

compared to the baseline configuration of RAW and QCOW2, respectively.

For SORT, we prepare 20 GB of text with the "randomtextwriter" and run the "sort" application, where the applications are also provided as examples. The results are similar to that of the WORDCOUNT application. The performance of the baseline configurations is degraded as the application iterates each run, yet the trimmed configurations exhibit the sustained performance and achieve 2.16x and 3.47x speed-up compared to the baseline configurations of RAW and QCOW2, respectively.

## 5. Conclusion

We confirm that the performance of SSDs can be maximized and sustained by handling the TRIM commands in a virtualized environment properly. We introduce FTRIM which bridges the semantic gap between VDI and filesystems and gives optimization opportunities to them. We believe that FTRIM can be used in other domains than the virtualized environment as it becomes common for applications to manage their own storage space on a file [8]. FTRIM can provide the applications with the opportunity to manage their storage space more efficiently.

We are working on implementing the file deallocation and optimizations that we mentioned. We plan to deploy our scheme on a large-scale environment, and conduct quantitative analysis on the benefits of the optimization.

## Acknowledgments

This work was supported by Next-Generation Information Computing Development Program (No. 2011-0020520) and by Mid-career Researcher Program (No. 2011-0027613) through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology. This work was also partly supported by the IT R&D program of MKE/KEIT (KI10041244, SmartTV 2.0 Software Platform).

## References

[1] Amazon, Inc. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`, 2011.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *Proceedings of the 22th ACM Symposium on Operating Systems Principles (SOSP'09)*, 2009.

[3] H. J. Choi, S. Lim, and K. H. Park. JFTL: a flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage*, 4:14:1–14:22, Feb. 2009.

[4] B. Debnath, S. Sengupta, and J. Li. ChunkStash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, page 1616, 2010.

[5] P. Desnoyers. Empirical evaluation of NAND flash memory performance. In *Proceedings of the 2010 USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'10)*, pages 50–54, Mar. 2010.

[6] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 29–43, 2003.

[7] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th international conference*

*on Architectural support for programming languages and operating systems (ASPLOS'99)*, volume 44, page 229240, Mar. 2009.

[8] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, SOSP'11, pages 71–83, 2011.

[9] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009.

[10] INCITS. T13 ATA8 draft specification 1697-d. 2010.

[11] INCITS T13. Data set management commands proposal for ATA8-ACS2 (revision 6),(draft specification t13/e07154r6). 2007.

[12] J. Katcher. Postmark: A new file system benchmark. `http://www.netapp.com/tech\_library/3022.html`, 1997.

[13] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[14] S. Kim, D. Jung, J. Kim, and S. Maeng. HeteroDrive: reshaping the storage access pattern of OLTP workload using SSD. In *Proceedings of 2009 4th International Workshop on Software Support for Portable Storage (IWSSPS'09)*, pages 13–17, Nov. 2009.

[15] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim. A case for flash memory SSD in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, page 10751086, 2008.

[16] S. Lee, B. Moon, and C. Park. Advances in flash memory SSD technology for enterprise database applications. In *Proceedings of the 2009 international conference on Management of data (SIGMOD'09)*, June 2009.

[17] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the 23th ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011.

[18] D. Ma, J. Feng, and G. Li. LazyFTL: a page-level flash translation layer optimized for NAND flash memory. In *Proceedings of the 2011 international conference on Management of data (SIGMOD'11)*, pages 1–12, 2011.

[19] A. Mashtizadeh, E. Celebi, T. Garfinkel, and M. Cai. The design and evolution of live storage migration in VMware ESX. In *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011.

[20] C. Mason. Compilebench. `http://oss.oracle.com/~mason/compilebench`.

[21] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, June 2007.

[22] M. McLoughlin. The QCOW2 image format. `http://people.gnome.org/~markmc/qcow-image-format.html`, 2008.

[23] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: Virtual disks for virtual machines. In *Proceedings of the 2008 3rd European Conference on Computer Systems (EuroSys'08)*, 2008.

[24] Microsoft, Inc. Microsoft virtual hard disk overview. `http://technet.microsoft.com/en-us/bb738373`.

[25] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation (NSDI'06)*, pages 26–26, 2006.

[26] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[27] Samsung Electronics Co. K9XXG08UXM flash memory data sheet, 2007.

[28] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. pages 119–130, 1985.

[29] S. Schlosser and G. Ganger. MEMS-based storage devices and standard disk interfaces: A square peg in a round hole. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST'04)*, pages 87–100, 2004.

[30] E. Seppanen, M. T. O'Keefe, and D. J. Lilja. High performance solid state storage under Linux. pages 1–12, 2010.

[31] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference*, 1996.

[32] C. Tang. FVD: a High-Performance virtual machine image format for cloud. In *Proceedings of the 2011 USENIX Annual Technical Conference*, Portland, OR, June 2011.

[33] TerryE. Tutorial: All about VDIs. `https://forums.virtualbox.org/viewtopic.php?p=29266`, 2008.

[34] The Apache Software Foundation. Hadoop MapReduce. `http://hadoop.apache.org/mapreduce`, 2011.

[35] The Apache Software Foundation. Hadoop distributed file system. `http://hadoop.apache.org/hdfs`, 2011.

[36] VMware, Inc. Virtual machine disk format (VMDK). `http://www.vmware.com/technical-resources/interfaces/vmdk.html`, .

[37] VMware, Inc. VMware VMFS produce datasheet. `http://www.vmware.com/pdf/vmfs_datasheet.pdf`, .

[38] Wikipedia, The Free Encyclopedia. Solid-state drive. `http://en.wikipedia.org/wiki/Ssd`, 2011.