Multimed Tools Appl (2015) 74:63–83
DOI 10.1007/s11042-013-1497-6

Improving write amplification in a virtualized and multimedia SSD system

Dingding Li · Hai Jin · Xiaofei Liao · Jia Yu

Published online: 15 June 2013
© Springer Science+Business Media New York 2013

Abstract Due to offering fast random-access disk I/O, it appears that solid-state drives (SSD), which is based on NAND flash memory, can suit well with the environment of cloud computing, especially for the cloud providing video streaming services. However, by investigating a practical virtual desktop system, where runs video streaming workloads, we find that importing this kind of SSDs into a virtualized system is not as simple as merely a mechanical replacement. Because a large proportion of disk I/O being included in the video streaming workload is write I/O, the inherent weaknesses of NAND flash memory, write amplification (WA), will be magnified in a guest operating system (OS). Worse, some useful remedies in a native OS become disabled or inefficient due to the interposition of hypervisor layer. This paper describes and analyzes these problems based on a practical virtual desktop system, and then proposes a tailor-made method to relieve them. By evaluating realistic user workloads and several typical benchmarks, the result shows that our method can effectively improve these problems in our virtualized SSD system.

Keywords Solid state drive · Virtualization · Write amplification

1 Introduction

In a typical virtualized environment, host machine uses *virtual machine monitor* (VMM or hypervisor) to multiplex the scarce hardware resource and then provides guest operating systems (guest OS) with an exclusive illusion. In such a way, various guest OSes can be consolidated into a single physical server, with strong isolation for each other, thus improving the hardware utilization and power consumption.

D. Li (✉) · H. Jin · X. Liao · J. Yu
Services Computing Technology and System Lab,
Cluster and Grid Computing Lab, School of Computer Science and Technology,
Huazhong University of Science and Technology, Wuhan, 430074, China
e-mail: dingly.hust@gmail.com

However, there is a trade-off between these benefits and I/O performance [17]. If multiple guests are running I/O intensive workloads simultaneously, the related I/O devices, with slower processing speed than CPU or memory subsystem, will be quickly saturated. As a result, a performance bottleneck in the system is formed, where severely disturbs the subsequent I/O operations.

Disk I/O is such a typical example [10]. Unfortunately, the common rotating disk device (also called hard disk drive, HDD) will further deteriorate the above issue. In a general case, different guests have different image files. When dealing with the concurrent I/O requests from different guests, the magnetic head of a rotating disk may do irregular movement among the image files, even these I/O flows are logically sequential from the perspective of their own guests. Therefore, this fragmental pattern produces random disk I/O and contradicts the physical characteristics of a rotating disk device. Eventually, the virtualized system presents a poor disk I/O performance to upper guest applications.

Instead of using electromechanical devices containing spinning disks and movable heads, SSD is often built with microchips which retain data in non-volatile flash memory chips and contain no moving parts [1]. In this way, access latencies are submillisecond, thus exhibits high random access performance. By considering many other advantages, such as lower power consumption, less susceptible to physical shock, and less noise, SSD has the potential to revolutionize the storage system landscape [19].

Nevertheless, SSD has its own Achilles' heel—*Write Amplification* (WA). Due to its inherent nature, flash memory must be erased before it can be rewritten. Therefore, a rewrite operation may invoke an extra and undesirable process—*read-modify-erase-write*, in which consumes the internal bandwidth inside the flash memory and thus reduces the incoming write performance on SSD [9].

Although many researchers and manufactures have proposed many methods to improve WA at device level [5], through a case study based on a virtualized SSD system, we find three features could directly trap a guest OS into one severe and interminable WA storm.

Application likes to write Compared with traditional applications, the proportion of write I/O in modern ones is increasing. For an example, *iPhoto* [3], a typical desktop multimedia application, can call `fsync` thousands of times in even the simplest of tasks [11]. This feature can spawn many write operations in a moment.

Rewrite is everywhere This is caused by the fact that when a guest deletes a file from its file system, the host file system only correspondingly delivers several block level writes into the meta data portion of guest image file to update the related blocks [12]. This action does not physically erase any data at host-level. In this way, after a guest image file is exhausted its specified capacity, any writes from this guest will become rewrites at host level, no matter how the guest cleans its virtual disk space. Worse, because of hypervisor cutting the semantic bridge of guest's deletion off, it also disables TRIM semantic [14] in guest OS.[1]

[1]To avoid the WA as much as possible, a native SSD system often uses TRIM command to physically delete the invalid files in its NAND flash memory for increasing the clean space.

Limited on-board cache in a shared SSD On-board cache inside a SSD can effectively relieve the WA issue [14], because a write request is complete immediately after arriving at this cache, before the data are actually written to flash memory. However, this kind of caches inside a SSD often has limited capacity [23]. Considering virtualization technology consolidates more workloads atop a single disk device, there exists a wide gap between the performance requirement of increasing workloads and the limited size of a on-board disk cache. Therefore, combined with the first two features, a virtualized system could use up the limited disk cache much quicker, allowing more rewrite operations to rush into flash memory of the shared SSD.

To improve above problem, we propose VFlashCache—a block level write cache tailor-made for a virtualized SSD system, especially for those who only have inexpensive commodity SSD devices. This dedicated write cache reinforces the limited on-board cache in a virtualized SSD device, allows each guest has a special cache structure to buffer and absorb rewrite operations, presenting a symmetric caching scale to fit the multiplied workloads in the environment of server consolidation. Furthermore, by devising a tailor-made management policy on this cache, VFlashCache effectively reduces the number of rewrite operation that necessarily touching the flash memory. This even can benefit SSD's endurance.[2]

VFlashCache locates in hypervisor layer, where is isolated from the upper guest instances and the bottom SSD hardware device. Therefore, it is a drop-in method, regardless of the specific virtualization technology (e.g. para-virtualization or hardware virtual machine), the specific type of guest file system (e.g. NTFS on Windows or Ext3 on Linux) and the specific kind of SSD device (e.g. produced by Intel or Kingston, target low or high ends of market). This makes VFlashCache highly compatible for existing systems.

It should be noted that a write cache can be somewhat inapplicable to some applications [22], because a power outage would make the data lost from the cache. To solve this issue, VFlashCache provides an option disk I/O path to these applications which require strict write semantic. This renders VFlashCache quite desirable in practice.

We implement our method in Xen hypervisor and evaluate its performance. The result shows that VFlashCache can effectively improve the WA issue in a virtualized system.

In summary, we have made the following contributions in this paper:

- First, we have studied the issue of write amplification in a practical virtualized SSD system based on realistic user traces.
- Second, we have carefully designed a drop-in and general method, called VFlashCache, to relieve WA in our virtualized SSD system.
- Third, we have implemented VFlashCache in Xen hypervisor and evaluated its performance and shown the effectiveness of improving the WA through realistic user workloads and typical benchmarks.
- Finally, we have assessed the inherent overhead in hypervisor layer, which is brought by VFlashCache.

[2]Flash memory can only be programmed and erased a limited number of times. This is often referred to as the maximum number of program/erase cycles (P/E cycles) it can sustain over the life of the flash memory. This metric largely depends on the number of rewrite that SSD has served.

The rest of this paper is organized as follows. In Section 2, we discuss our motivation, which is derived from a practical virtual desktop system. Sections 3 and 4 introduce the design and implementation of our solution respectively. We present our evaluation in Section 5. Section 7 describes the related work. The last section discusses and concludes this paper.

2 Motivation

2.1 ClouDesk

ClouDesk [18] is a typical virtual desktop system, which is deployed on a set of Intel blade servers. These servers often have enterprise-level CPU and memory subsystems, but without a special storage pool. It uses the local and inexpensive commodity storage device to act as its back-end storage. Therefore, there is a challenge to meet the demands of performance and reliability on the storage layer of ClouDesk.

End-users in ClouDesk incline to edit documents, browse web pages, and watch P2P streaming video [21]. These applications often contain heavy write I/O [11]. For an example, P2P streaming media software usually download playing files ahead into local storage to get a better watching experience. In particular, when users are watching hot live TV simultaneously on a single machine, the write I/O upon the shared storage is quite intensive. Therefore, workloads in ClouDesk are often present *multimedia* and *text-editing* style.

We give a detail introduction of ClouDesk as follows: (1) An end-user can use terminal device, such as the common PC or tablet PC, to connect his own virtual desktop environment. This desktop environment is provided by a special guest OS, which is hosted by one of Intel blade servers. (2) Each Intel blade server in ClouDesk has dual Quad-Core Intel Xeon 1.6 GHZ processors, 8 GB DDR2 RAM, a typical and popular SSD device with 80 GB capacity and 32 MB on-board disk cache,[3] and dual Full-duplex Intel Pro/1000 Gbit/s NIC. (3) A VNC-like remote desktop protocol will maintain the interaction between guest OS and end-user, through a 1000Mb network. (4) Xen hypervisor is installed inside each Intel blade server. The driver domain (also called Dom 0) is running 64-bit Fedora 14 distribution with `ext4` file system and the hypervisor is Xen 4.0.1 with Linux 2.6.34.6 kernel. Guest OSes (also called Dom Us) are running CentOS 5.6 with Linux 2.6.18.8 kernel and use `ext4` as their file systems based on `qcow` format. A 8 GB upper limit is enforced on the size of each virtual disk. Each Dom U is allocated 512 MB memory. The block device driver is `Blktap` [24] with `qcow` protocol.

2.2 Methodology

We choose one of Intel blade servers as our experimental machine and boot eight Dom Us to provide equivalent end-users with desktop service. The typical user operation includes document editing, web-page browsing, and code development. To have a comprehensive comparison, we also install a HDD device in this experimental

[3]This SSD device has similar specification with Intel X25-M G2.

machine, which is a 160 GB SATA II hard drive with 7200 RPM and 64 MB on-board disk cache.

We install an agent inside Dom 0 to collect the guest disk I/O traces at block level. To rule out the interference of local disk I/O, all logs from the agent will be transferred into another special Intel blade server. These logs will be organized as the form of raw trace files and each of them corresponds to a certain Dom U. Every trace file uses four fields to identify a unique guest read/write: (1) *stime*, it denotes the system time when a request arrived on Dom 0; (2) *fd*, it refers to the file description of a certain guest image file; (3) *offset*, points to the data location in the image file; (4) *resp*, the request response time, indicates the necessary time to complete this request. We record guest requests as appending manner, and so *eight* raw trace files, which marshal the read/write records with ascending order based on the *stime* field, can be finally obtained.

Algorithm 1 Reducing raw trace file

```
Input: F
Output: Outputfile

top = 0;
WriteLatency = 0;
OutputArray = NULL;

while Not at end of F do
    WriteLatency = GetNextWriteLatency(F);
    if top=0 or ABS(WriteLatency − OutputArray[top − 1]) > 200μs then
        OutputArray[top++] = WriteLatency;
    end
end

for i=0 to top do
    Write(Outputfile,OutputArray[i]);
    i++;
end

return;
```

We first conduct this process upon the HDD device. This process lasts about one week. At the end, we have gathered 18,475,296 write records and 25,339,195 read records, respectively.

Then we conduct the same process upon the SSD device, which lasts about three weeks. *In the first two weeks* we collect a total of 16,890,614 write records and 20,408,037 read ones. *In the third week* we also obtain a log containing a total of 9,638,711 write records and 14,724,955 read ones.

To clearly present the overall view of disk I/O as *system wide*, in each case we merge all collected guest raw trace files into a single textual file, denoted by F, in which sorting these I/O records based on their *stime* field.

Due to large volume data, we only choose write requests from F[4] and then reduce them into a smaller set by using Algorithm 1. This reduced trace file uses *write segment* as its unit.

[4]To clarify the point that how a virtualized SSD device affects guest write in this paper, we ignore all read requests on figure plotting. However, we will include them into the trace analysis by strictly referencing the original trace file.

2.3 Problem description

Figure 1 shows our trace results. In the case of *HDD*, most write latencies, which are logarithmic, varied between the y-axis range of 10,000 and 20,000 with some abnormal points even exceeding 100,000. By investigating the involved write segments and raw trace files, we find this fluctuation is caused by concurrent disk I/O flows during a short period. Taking one of them that rounds the 12,500th scale on x-axis as an illustrative example (we marked a label 1 in Fig. 1), a time interval, only across about five seconds, is filled up with 3,753 raw write requests from five guest instances and 10,049 raw read ones from six guest instances. Not surprisingly, this congestion arouses random disk I/O and thus degrades their write latencies.

In the case of *SSD* is collected in *the first two weeks* (as the blue symbols in Fig. 1), SSD provides great improvement to the write latency: its performance is generally over two orders of magnitude higher than that of HDDs. Note that several abnormal points with high latency are being dotted sporadically along the x-axis (we marked a label 2 in Fig. 1). By investigating the raw trace files, we find that there are intensive random-write flows nearby these involved write segments. Therefore, we speculate the corresponding abnormal points incurring a mild WA. Nevertheless, SSD still greatly benefits the performance of concurrent I/O flows in our system.

However, in the case of *SSD* collected *in the third week* (as the symbols with light green color in Fig. 1), we can see that there is a quite large fluctuation on guest write performance. It spans four orders of magnitude of write latencies on y-axis. From the label 3 in Fig. 1, we can clearly see that the fluctuation of write latency happens so often and so severely that end-users even can perceive the delay when they are interacting with their own desktops. According to the user-feedback survey report from this stage, this high delay directly affects the user experience if end-users watch their P2P streaming videos, browse web pages, and even operate a file.

By investigating the current status of experimental machine, we determine that the situations are caused by WA issue in a virtualized SSD system. The 80 GB capacity on the shared SSD is almost exhausted. Therefore, the coping mechanisms at device-level, such as garbage collector and over-provision [13], are difficult to run efficiently to receive rewrite requests.

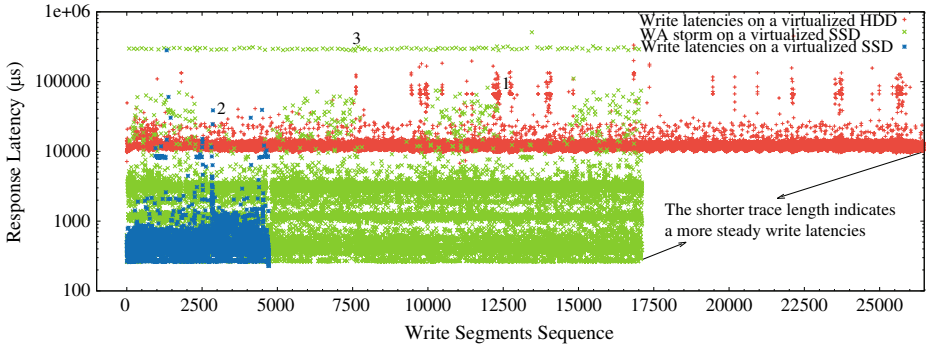


Fig. 1 Write latency on ClouDesk, the y-axis is logarithmic

2.4 Discussion on potential solutions

According to the storage protocol stacks of a typical virtualized system, there are three layers can be embedded with the related improvements to relieve above WA issue: *guest* layer, *hypervisor* layer, and *device* layer (or their collaboration).

2.4.1 Guest layer

By tweaking its inherent configuration, a guest can reduce the number of rewrite that must touch the SSD's surface. For example, delaying the execution of periodical flush routine could effectively absorb rewrites inside guest file system. However this kinds of method is exposed to a guest and thus it is also visible to the upper users. This may motivate a guest, who can arbitrarily modify its default configurations and intentionally spawn massive writes to disturb write performance of the shared SSD device. This result is disastrous to the virtualized system based on end-users' experience, such as ClouDesk, in which the user experience in a same physical machine would become bad and even hard to use. Therefore, from security perspective, we leave the improvements implemented inside guest.

2.4.2 Device layer

Some methods at device-level have been proposed recently. For example, CAFTL [5], a content-aware flash translation layer, can remove the duplicated content inside SSD and thus produce more clean erase-blocks to receive rewrite requests. Since a part of guest instances shares the same kind of OS, this novel technology may suit ClouDesk well. Displacing current PC-level SSD devices with these novel ones is a potential solution, in which the later often has the larger internal cache to relieve the WA issue.

Nevertheless, this kind of improvement inside device would largely increase our production costs. We conduct a survey on the recent Amazon online shopping [2], finding that the newly released SSD products, which contain novel technologies or large on-board cache, are usually launched with double or triple price over popular ones. Furthermore, these novel techniques based on hardware level are usually designated at the manufacturing process. Some of them are only verified in the simulated environment, such as *DiskSim* [1, 7]. Therefore, we do not expect SSD manufacturers willing to redesign or reproduce the SSD devices for our virtualized system now.

2.4.3 Hypervisor layer

Limiting improvements within hypervisor layer has two advantages: (1) it is unperceived to the upper guest OSes; (2) it is transparent to the bottom storage device. Both of them make improvements inside hypervisor layer presenting a *one-size-fits-all* style. This convenience feature is quite suitable to ClouDesk since we provide end-users with a lot of choices on their preferred OSes. For example, ClouDesk has two kinds of OS template: Linux and Windows. Each of them further consists of different distributions and combinations with different file systems, such as `Fedora 14` with `ext3`, `Windows XP` with `NTFS` and `Ubuntu 11` with `btrfs`. An improvement inside hypervisor layer could get them all in a general way, largely reducing our intrusion cost to original systems.

Like *Google File System* (GFS) [8] running on inexpensive commodity hardware, another benefit from this kind of improvements is that ClouDesk can keep inexpensive commodity SSD device to act as its back-end storage, as the performance level of these inexpensive storage devices are transparently enhanced. Therefore, we can avoid an extra expenditure on special hardware such as the `OCZs Z-Drive R4 CloudServ` [20] and `Fusion I/O`[5] [4].

However, importing these improvements into original system complicates the hypervisor layer, where is the kernel component in a typical virtualized system. Therefore, implementing any extraneous components into this layer should undergo a careful design.

3 Solution

In this section, we first describe the overview of our solution. Then we give a set of specific design considerations that have guided our solution and present its structure—VFlashCache. Finally, we describe the detail caching behaviors inside VFlashCache.

3.1 Overview

We interpose a block level write cache into hypervisor layer, called VFlashCache. This cache is used for absorbing the duplicated guest rewrites while presenting the asymmetric write performance to defend the upper applications from directly suffering WA storm.

VFlashCache intercepts guest write into hypervisor's address space and allows this I/O operation to be returned in advance, rather than directly driving it into the SSD device. When a duplicated rewrite is arrived in hypervisor, VFlashCache is capable of overlapping the early intercepted data with the newest one. In such a way, it reduces the number of rewrite that necessarily touch the SSD surface, thus indirectly improves the SSD's write performance and even endurance. Even if a virtualized SSD is incurring WA storm, on the other hand, VFlashCache raises opportunities for interleaving the flush flows from different guest OSes, and provides SSD device with a breathing space to receive rewrite request with higher-efficiency. Therefore, guest writes in VFlashCache can be flushed into SSD as a more elegant manner. We call this SSD-aware flush mechanism in VFlashCache as *FlushCtrl*.

3.2 Design considerations

VFlashCache's design is based on three high-level themes.

Performance How to improve WA in a virtualized SSD system is our main concern. Meanwhile, the read performance is also needed to be kept.

[5]This kind of SSD devices usually have a good performance on handling rewrite operations but often trading with expensive price.

Generality Virtualization technology can simultaneously host a variety of guest OSes in a single machine, thus VFlashCache should be compatible to these various guest instances and even different hypervisors. Furthermore, a write cache may violate the guest write semantic. VFlashCache should provide an alternative disk I/O path to these guests who require strict write semantic. Finally, VFlashCache needs to provide agnosticism and isolation to the bottom SSD device, which may spans dozens of manufactures and hundreds of products.

Simplicity Due to server consolidation, a virtualized system is usually resource-restrained. Therefore, VFlashCache's design should follow the lightweight pattern as much as possible. A compact design can also benefit the system's reliability.

3.3 System structure

Figure 2 shows an overview of VFlashCache's architecture. Generally, VFlashCache adopts a *local-based* design. Namely, VFlashCache allocates an exclusive cache space, called `VDiskcache`, to each guest instances, used for buffering guest write I/O. A combination of these cache spaces forms our VFlashCache in hypervisor layer. In each VDiskcache, a flush handler, called `FlushCtrl`, will flush these buffered writes back into SSD device, depending on specific situations described in Table 1.

VFlashCache is located in the hypervisor layer, where has two potential benefits for implementing our design considerations. First, due to being required to multiplex the shared I/O resource, hypervisor needs to validate each I/O instruction from guest OS. This can expose a drop-in interface to VFlashCache, allowing VDiskcache to intercept guest write requests without intrusive modification on an original system. Second, since certain isolation has been enforced by hypervisor, VFlashCache does not need to involve any correlations with specific guest or SSD device. All of design of VFlashCache is encapsulated at the hypervisor layer and hidden from guest and SSD device.

Not like a *system-wide* method that uses the file cache in a host file system to buffer guest writes, VFlashCache's design follows a simpler manner which is *local-based*.

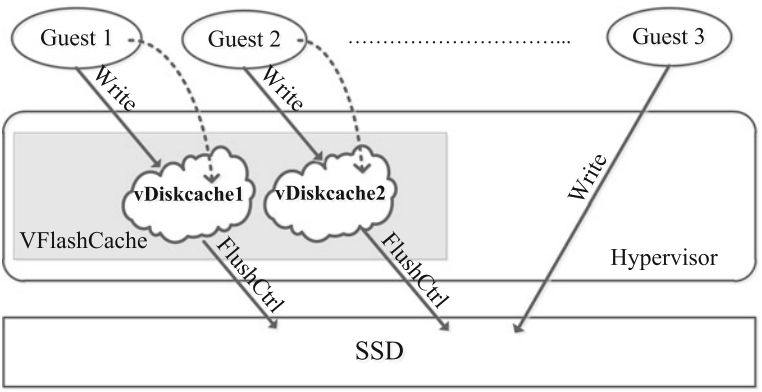


Fig. 2 Structure of VFlashCache. A flush handler inside VFlashCache, called FlushCtrl, will flush these buffered writes back into SSD device, depending on specific situations described in Table 1

Table 1 Trigger conditions of FlushCtrl

Event	Description
Timer	FlushCtrl will be invoked periodically to flush guest data back into persistent storage.
Space management	When a VDiskcache is full, FlushCtrl must invoke a flush handler to conduct a write-back process.
Guest migration	When a guest is being migrated to other host, FlushCtrl should flush its dirty-data back into SSD device.
Guest close	When a guest is closed by user or administrator, FlushCtrl is invoked to flush its dirty-data.
Guest crash	The associated VDiskcache in VFlashCache can catch the event of guest crash by a heartbeat mechanism.

Namely, each VDiskcache's management only focuses on a single guest. Consider hypervisor is the kernel component in a virtualized system, this simplicity is important to maintain the reliability and succinctness of whole system.

More importantly, this local-based design can provide an alternative disk I/O path for those guests which require strict write semantic, to bypass VFlashCache and then to directly talk with SSD device, such as the *Guest* 3 in Fig. 2.

Finally, as the dotted arrow showing in Fig. 2, the memory resource used by VFlashCache comes from guest itself, rather than being directly allocated from hypervisor. Thus, it has less influence on the scalability of original virtualized system.

Algorithm 2 Write behavior of VFlashCache

Input: W_i, denotes a block-level write request
Output: A_i, indicates the acknowledgment of W_i

```
1  Let N be the set of old versions of W_i in VDiskcache;
2  Let θ be the flush threshold of VDiskcache;
3  Let c be the current number of write request in VDiskcache;
4  N = NULL;
5  N = SearchInVDiskcache(W_i);
6  if IsHit(N,W_i) then
7  |  InsertVDiskcache(W_i);
8  |  ClearFlushFlag(N);
9  |  SetFlushFlag(W_i);
10 end
11 else if NotHit(N) then
12 |  InsertVDiskcache(W_i);
13 |  SetFlushFlag(W_i);
14 end
15 c++;
16 if c == θ then
17 |  FlushVDiskcache();
18 |  /*only flush the write request with flush flag.*/;
19 |  c = 0;
20 end
21 return (A_i);
```

3.4 Caching behavior

Generally, VFlashCache uses basic FIFO (*First-In-First-Out*) replacement policy to manage its space. While a replacement algorithm based on LRU (*Least-Recently-Used*) may obtain better performance, VFlashCache must keep the constraints of guest I/O ordering. For example, when guest is writing to an unallocated region of a file, we must flush the file data before writing the meta data, otherwise it risks exposing uninitialized data to users.

To reduce the number of guest rewrites that would flush into SSD's flash memory, FlushCtrl only flushes the newest data, if an incoming write in the current VDiskcache has any old versions. In this way, we not only keep the basic write ordering between meta data and regular data in a guest file system, but also provide a critical chance to improve WA issue by reducing the number of rewrites. Algorithms 2 and 3 describes the caching behavior of VDiskcache for dealing with the guest write and read requests respectively. It should be noted that the read request from a guest must find its target in VDiskcache first, to keep data consistency.

Algorithm 3 Read behavior of VFlashCache

```
Input: R_i, denotes a block-level read request
Output: A_i, indicates the acknowledgment of R_i
Let N be the hit set of R_i in VDiskcache;
N = NULL;
N = SearchInVDiskcache(R_i);
if IsHit(N,R_i) then
    R_i=ReadtheNewestOneFrom(N);
    return (A_i);
end
ReadFromSSD(R_i);
return (A_i);
```

To FlushCtrl, there are five types of event will launch it to do a real write-back process in a VDiskcache, as Table 1 summarizing. During each flush process, FlushCtrl always starts from the head of a VDiskcache and then scans forward. If a cached write request has been cleared its flush flag, which indicates that it will be rewritten later, FlushCtrl just ignores it and keeps on going.

Finally, a mutex is used in VDiskcache for ensuring only one flush handler can operate at a certain time.

4 Implementation

We have implemented VFlashCache in Xen hypervisor, an open-sourced and popular virtual machine monitor. Although our work is focused on a specific virtualized SSD system, the main idea can be easily applied to other systems.

We add a new protocol, named *ssd*, into Blktap driver, which uses Tapdisk as VDiskcache for caching the guest writes. Our implementation on Xen hypervisor totally consists of 990 lines of code on the user-space of Dom 0.

There are two kinds of meta-data used to manage each VDiskcache: a linked-list and a hash table. The linked-list is used to store information on write requests.

Each node in the list corresponds to a specific write. Data nodes in the linked-list are arranged in an order from head to tail according to their arrival times. The hash table is used for effective data searching according to the *offset* field,[6] in which the nodes with same value on *offset* field in the linked-list can be quickly identified.

For the implementation of FlushCtrl, we interpose those flush handlers into the related functions or routines in Tapdisk. For an example, to trigger the flush procedure when Dom U is terminated or migrated, we insert the flush handler into `unmap_disk()` in Tapdisk and `xc_domain_save()` in Xen tools.

5 Evaluation

In this section, we present the evaluation of VFlashCache. By using real user workloads on virtual desktop, we first show its direct improvement on ClouDesk. Next, we show the performance of VFlashCache under several synthetic workloads and typical realistic workloads. Finally, we discuss the overhead that VFlashCache enforces on driver domain.

We compare VFlashCache with the original Blktap driver with `qcow` protocol. To have an apple-to-apple comparison, we allocate each guest on VFlashCache with only 448 MB memory and enforced an upper limit of 64 MB (namely θ described on Algorithm 2) on their own VDiskcaches. The timer of periodical flush handler on VFlashCache is set to 10 min. On the flipped side, the memory size of each guest on original Blktap is set to 512 MB.

5.1 VFlashCache on ClouDesk

We use the added protocol *ssd* in Blktap to boot all *eight* Dom Us in our experimental node, where succeeds the context from Fig. 1 that has suffered WA storm. We finally get eight raw trace files after one week, which contains a total of 8,902,217 write records and 29,744,419 read ones. By excluding read requests and using Algorithm 1 to reduce their datasets, a processed trace file with only 1,237 write segments is obtained. Figure 3 shows the result.

VFlashCache greatly relieves the WA storm. This improvement is reflected by the elimination on those write requests with high delay, which exceed the scale 100,000 on y-axis. The distribution of write latency on VFlashCache mainly concentrated on four areas in Fig. 3, and we have marked the corresponding labels.

Write segments around label 1 refers to those guest writes only touching the VFlashCache. Since they are returned before entering into SSD surface, all of them avoid the possible WA storm, and obtain an extra performance improvement in contrast to the case without our cache.

Label 2 represents the write segments who trigger a flush handler in VFlashCache. They are required to wait for the completion of flush procedure and thus incur a performance penalty.

Write segments around label 3 and 4 denote another type of write segments. They are involved in the concurrent foreign flushes, which are derived from another guests.

[6]The *offset* field refers to an offset from the starting position of a guest image file.

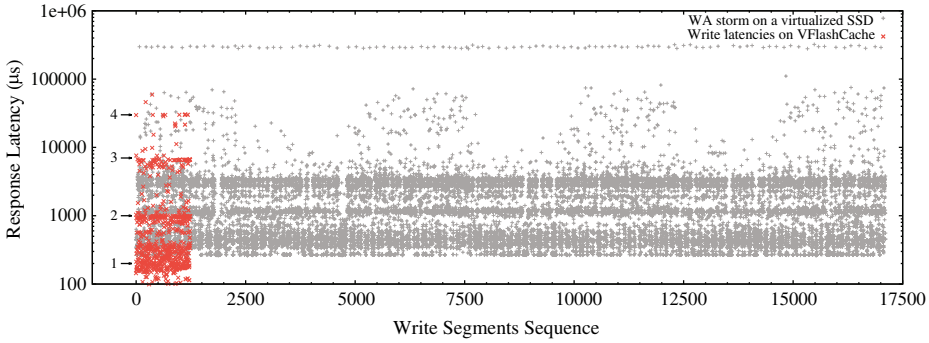


Fig. 3 Write latency on ClouDesk, the y-axis is logarithmic

Note that the points with a higher latency around label 4 implicate they are suffering a more concurrent foreign flush flows. Therefore, due to bandwidth congestion inside SSD, the relevant write segments still incur a relatively severe WA. However, they are minor, and part of them will be defused by the background writes inside a guest file system, such as `pdflush` thread. Therefore, users may not directly perceive this delay during their interactions. According to the user-feedback survey report recently, most users have not observed the influence of WA, even in the multimedia scenario.

Figure 4 shows the corresponding CDF plot of write delay for ClouDesk. Compared with a virtualized SSD incurring WA storm, it can be clearly seen that the proportion of write requests with high latency, which exceeds 1,000, are substantially reduced on VFlashCache.

On the other hand, VFlashCache can reduce the number of rewrites that necessarily touching the SSD surface. Denoting the total number of guest writes intercepted in VFlashCache as W, the total hit number of guest write as w, the write hit rate is defined as $\frac{w}{W}$. Figure 5 presents the write hit rate of the eight user workloads,

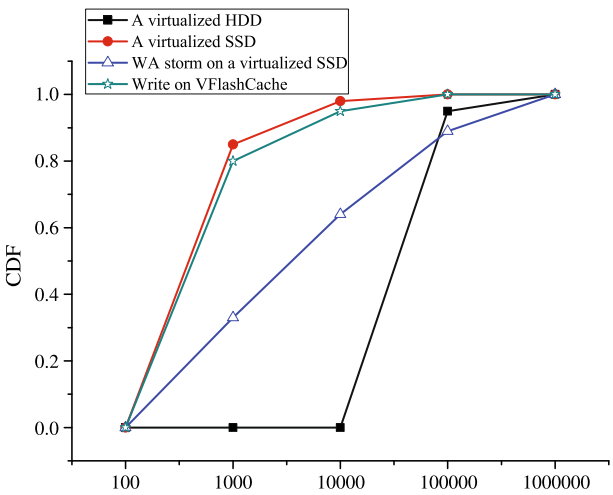


Fig. 4 CDF plot of write latencies on ClouDesk, which is compiled by the raw trace files. The x-axis is logarithmic

with their corresponding read hit rate on VFlashCache. The write hit rates across the 8 workloads range from 14 % (VM 7) to 31 % (VM 4). We also can see that the read one ranges from 4 % (VM 1) to 11 % (VM 5). The low hit rate of read request is caused by the fact that most hot user files are cached in the guest page cache, and make the flushed guest data in VFlashCache to be read less frequent. On the contrary, a higher write hit rate indicates a feature on ClouDesk: a few files are being operated or edited by users for a time, and they are saved more often.

5.2 Synthetic workloads

For comprehensive evaluating the performance of VFlashCache, we also use some typical benchmarks to simulate the common workloads. We first boot three Dom Us on our experimental node, and then in turn run the same benchmark on them simultaneously.

Figure 6 shows their results. The length of each bar indicates the relative performance between original Blktap driver and VFlashCache, which is based on the average score of three Dom Us. The first group of bars shows the result of `DBench` benchmark. Every Dom U has 20 virtual user connections. It clearly shows that VFlashCache can bring double performance in contrast to the original one. This is caused by the fact that the flush operation in `DBench` has a large proportion of overall score. VFlashCache just allows most of these flush operations to return in advance, thus avoids the interference of WA and acquires the extra performance benefit. On the other hand, VFlashCache absorbs some rewrites that necessarily touch the SSD surface. This workload in VFlashCache has a read/write hit rate of 8.4 % and 15.5 %.

In the second group, the performance of Linux kernel compilation (version 2.6.38) is evaluated. This workload reads massive source files, creates many object files, and thus generates a large number of operations for meta data update and journal synchronization to the file system. The result shows that VFlashCache has almost identical performance with its counterpart. While VFlashCache can relieve WA for these update operations on guest file system, the smaller memory size, which leads Dom Us to a smaller working set, makes this benefit to be somewhat

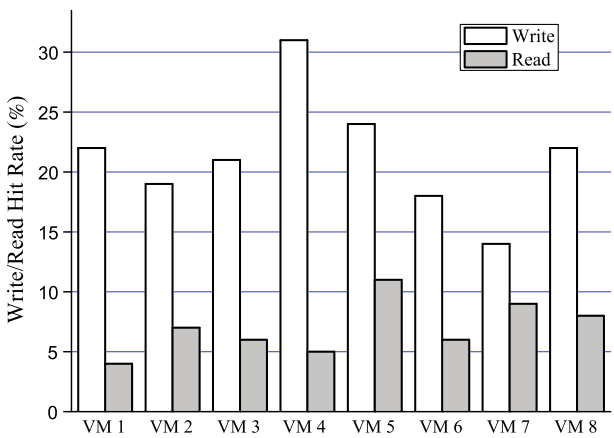


Fig. 5 Read-Write hit rate on ClouDesk with VFlashCache

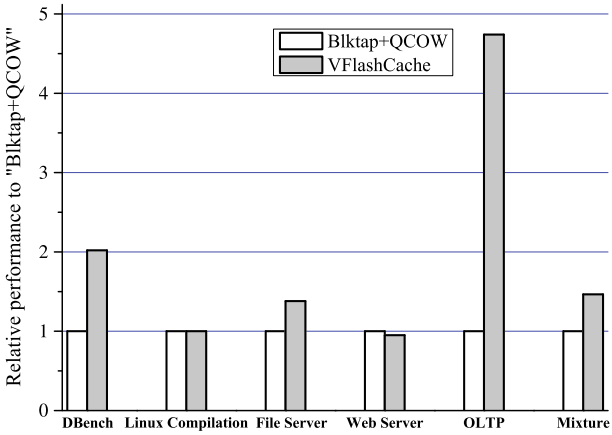


Fig. 6 Relative performance of original Blktap with `qcow` protocol and VFlashCache

counterbalanced by the event of read missing in guest page cache. Moreover, because this workload contains the process of computer intensive compilation, the proportion of I/O operations is consequently faded. VFlashCache totally achieves a read/write hit rate of 0.7 % and 7.9 % in this workload, respectively.

Then we use `IOmeter` to simulate four kinds of workload: file server, web server, OLTP, and their mixture. Each testing file during this experimental group is set to 1 GB.[7]

We first run file server workload in all three Dom Us simultaneously, the transfer request size is set to 64KB and the read/write ratio is 4:1 with 100 % random manner. The result shows VFlashCache can bring about 38 % improvement to the IOPS of server. A read/write hit rate of 19.2 % and 15.5 % is obtained on VFlashCache.

To web server workload, we set the transfer request size to 512KB and all of them are read operations with 100 % random manner. The result shows that VFlashCache incurs an approximately 5 % performance penalty on server IOPS. It is caused by the read intensive feature and a smaller page cache. VFlashCache achieves a read/write hit rate of 21.2 % and 1.8 % in this workload, respectively. While part of read requests are hit in VFlashCache, they are required to experience a context switch between guest and hypervisor layer, thus suffers a longer I/O path.

To OLTP case, the transfer request size is set to 8 KB, and read/write ratio is 3:1 with 100 % random manner. The result shows that VFlashCache obtains up to 4.75 times improvement over its counterpart on the IOPS. Considering this workload has the read/write hit rate of 7.2 % and 22.7 % in our cache, respectively, we conclude this bigger improvement mainly comes from the higher reduction of rewrites.

The next one is *Mixture*. We deploy the above three configurations of `IOmeter` into the equal number of Dom Us and run them simultaneously. The result shows that VFlashCache has about 45 % improvement over original `Blktap` driver.

[7]We also conduct a series of experiments by using 2 GB and 4 GB testing file size. But we find that the result of relative performance between VFlashCache and its counterpart is quite similar with the case of 1 GB size (the margin of error is about 1.5 percentage points). Therefore, we ignore them in this paper.

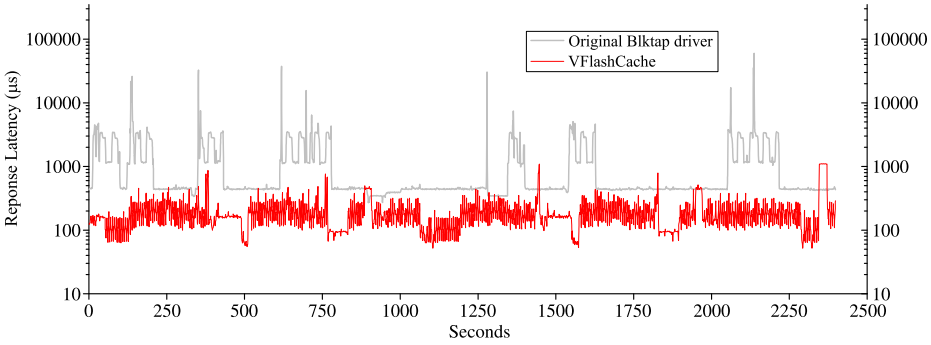


Fig. 7 Variation of write latency on synthetic multimedia workload, the y-axis is logarithmic

VFlashCache in the mixed workload has a read/write hit rate of 14.2 % and 6.7 %, respectively.

Finally, we use a synthetic multimedia workload to measure VFlashCache. In detail, we deploy *PPStream*, a popular P2P video software, on our all three experimental guests. Then we launch these applications and connect them to an available program source which lasts about 40 min. During this process, *PPStream* will respectively create a 1 GB temporary file, *ppsdg.pgf*, on their virtual disks to act as buffer role, thereby producing many read and write requests upon it. Similar with section II.B, we collect write requests at block level in driver domain, but without using Algorithm 1 to reduce the final raw trace file. According to the *stime* filed, these write request from three guests will be arranged in chronological order in raw trace file. If multiple write requests share the same value of *stime*, we merge them and take their median. Figure 7 shows the variation of write latency across this duration of 40 min (2400 s), on original `Blktap` driver and VFlashCache respectively. It can be seen clearly that the write latency on VFlashCache has the lower delay and better stability.

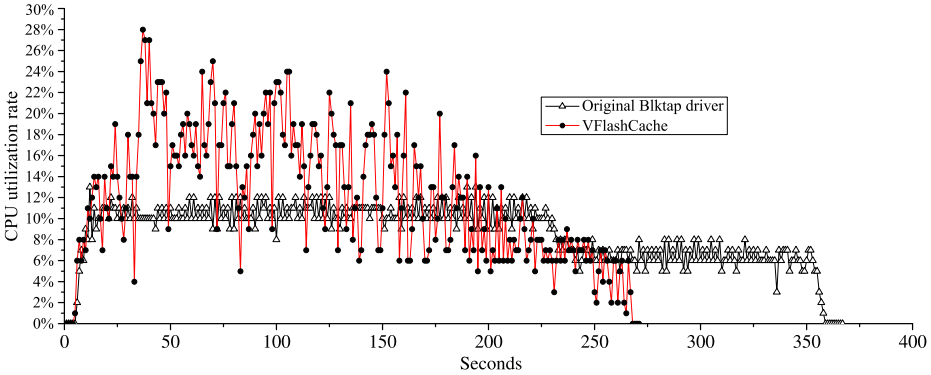


Fig. 8 CPU overhead in driver domain, the size of each testing file is set to 1 GB

5.3 Overhead in driver domain

To measure the necessary overhead of VFlashCache in Dom 0, we boot all *eight* Dom Us in our experimental node. We run `IOzone` benchmark on these guest file systems simultaneously and produce numerous small writes (4 KB) with synchronous manner. In such a way, VFlashCache in Dom 0 is required to deal with the highly concurrent writes. Since this situation is almost impossible to happen on realistic user workloads, we treat this experimental scenario as the *upper limit* of overhead in VFlashCache. The result is shown in Fig. 8. Generally, VFlashCache consumes about twice CPU resource to maintain *eight* vContext structures running, but the peak value is still resided below the scale 30 % on y-axis.

6 Related work

It is commonly known that SSD has the potential to replace traditional rotating disk device, especially for its read performance. However, it falls into a dilemma due to the limitation of P/E cycles. Recently, Grup et al. [9] further indicates that NAND flash memory has a bleak future due to the WA issue and the marketing strategy of manufacture. Therefore, currently we do not lay our hopes on the SSD itself. Instead, we try to seek a solution from the software layer, to strike a balance between its impressive read and apprehensive write.

There are many previous works at device level to study and improve the WA's problem of flash memory on a SSD device. A survey in [6] has summarized these works. We here only describe the works which are most related to a virtualized SSD.

Chen et al. have proposed CAFTL [5], a device-level method used to reduce write traffic to flash memory by removing unnecessary duplicate writes. In this way, some space consumptions on a flash memory can be saved, allowing the inherent coping mechanisms in a SSD controller, such as garbage collector and over-provision, can be run as optimal state and thus maintain the write performance to upper workloads. This method may suit our system well, since some guest instances in ClouDesk share the same kind of OS, in which a majority of data redundancies may happen on their image files. However, this technique requires SSD to be redesigned and reproduced. Consider currently CAFTL is only be implemented and verified in a simulated environment (e.g. `DiskSim` [7]), this kind of device-level methods are inapplicable to be used in practice.

More recently, Kim et al. have devised FTRIM [16], a support for running TRIM command in a virtualized environment, which makes SSD device be aware of the file deletion inside guest OS. As a result, the real SSD device in host can utilize the space of the deleted files and thereby maintain the rewrite performance from guest instances. Compared with VFlashCache, FTRIM requires the guest kernel to be modified, in which needs adding TRIM support into the specific virtual block driver. Therefore, this method violates our design consideration based on generality to various guest OSes.

Jo et al. have presented a hybrid virtual disks (HVD) that combines SSD and HDD for virtualization [15]. In this hybrid architecture, HVD places a read-only template disk image on a SSD, while write requests are isolated to the HDD. In this way, the disk I/O in a guest OS benefits from the fast read access of the SSD and precludes

write operations from degrading flash memory performance. It differs from our work in that VFlashCache focuses on a single storage structure based on SSD.

7 Conclusion

Through a case study of the practical virtual desktop system, we first confirm that the problem of write amplification in a virtualized environment will become severe due to the characteristic of virtual disk device. By interposing a block-level write cache into hypervisor layer to reinforce the limited on-board cache in a shared SSD device, our method can effectively improve the WA problem on a realistic workload. We also evaluate our method with some typical synthetic workloads, the result shows that it can improve the read-write mixed workloads and maintain the performance of read-intensive ones. We finally measure the inherent overhead enforcing on hypervisor layer, which is brought by our method. The result shows that this cost is reasonable.

Acknowledgements This work is supported by China National Natural Science Foundation (NSFC) under grants 61272408, 61133006, National High-tech R and D Program of China (863 Program) under grant No. 2012AA010905 and Hubei Funds for Distinguished Young Scientists under grant No. 2012FFA007.

References

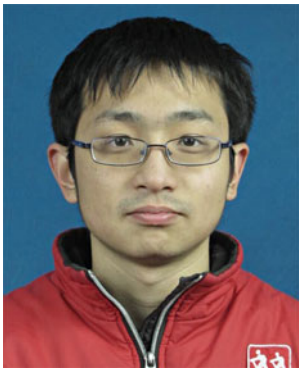
1. Agrawal N, Prabhakaran V, Wobber T, Davis JD, Manasse M, Panigrahy R (2008) Design trade-offs for SSD performance. In: Proceedings of the USENIX 2008 annual technical conference (USENIX 2008). USENIX Association, Berkeley, CA, pp 57–70
2. Amazon.com (2012) http://www.amazon.com
3. Apple—iPhoto (2012) http://www.apple.com/ilife/iphoto/
4. Application Acceleration Enterprise Flash Memory Platform (2012) http://www.fusionio.com/solutions/virtualization/
5. Chen F, Luo T, Zhang X (2011) CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In: Proceedings of the 9th USENIX conference on file and storage technologies (FAST 2011). USENIX Association, Berkeley, CA
6. Gal E, Toledo S (2005) ACM Comput Surv (CSUR) 37(2):138
7. Ganger G (2011) The DiskSim simulation environment (v4.0). http://www.pdl.cmu.edu/DiskSim/
8. Ghemawat S, Gobioff H, Leung ST (2003) The Google file system. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003). ACM, New York, NY, pp 29–43
9. Grupp L, Davis J, Swanson S (2012) The bleak future of NAND flash memory. In: Proceedings of the USENIX conference on file and storage technologies (FAST)
10. Gulati A, Merchant A, Varman P (2010) mClock: handling throughput variability for hypervisor IO scheduling. In: Proceedings of the 9th USENIX conference on operating systems design and implementation (OSDI 2010). USENIX, Vancouver, BC, pp 437–450
11. Harter T, Dragga C, Vaughn M, Arpaci-Dusseau AC, Arpaci-Dusseau RH (2011) A file is not a file: understanding the I/O behavior of apple desktop applications. In: Proceedings of the 23th ACM symposium on operating systems principles (SOSP 2011). ACM, New York, NY, pp 71–83
12. Hildebrand D, Povzner A, Tewari R, Tarasov V (2011) Revisiting the storage stack in virtualized NAS environments. In: Proceedings of the 3rd conference on I/O virtualization (WIOV 2011). USENIX Association, Berkeley, CA
13. Hu XY, Eleftheriou E, Haas R, Iliadis I, Pletka R (2009) Write amplification analysis in flash-based solid state drives. In: Proceedings of the Israeli experimental systems conference (SYSTOR 2009). ACM, New York, NY, pp 10:1–10:9
14. Intel Solid-State Drive Optimizer (2009) http://download.intel.com/design/flash/nand/mainstream/Intel_SSD_Optimizer_White_Paper.pdf

15. Jo H, Kwon Y, Kim H, Seo E, Lee J, Maeng S (2009) SSD-HDD-Hybrid virtual disk in consolidated environments. In: Proceedings of the 2009 international conference on parallel processing (EuroPar 2009), pp 375–384
16. Kim S, Kim J, Maeng S (2012) Using Solid-State Drives (SSDs) for virtual block devices. In: Proceedings of the runtime environments, systems, layering and virtualized environments (RESoLVE'12)
17. Le D, Huang H, Wang H (2012) Understanding performance implications of nested file systems in a virtualized environment. In: Proceedings of the USENIX conference on file and storage technologies (FAST)
18. Liao X, Jin H, Hu L, Liu H (2010) Towards virtualized desktop environment. Concurrency and Computation: Practice and Experience 22(4):419–440
19. Narayanan D, Thereska E, Donnelly A, Elnikety S, Rowstron A (2009) Migrating server storage to SSDs: analysis of tradeoffs. In: Proceedings of the 4th ACM European conference on computer systems (EuroSys 2009). ACM, New York, NY, pp 145–158
20. OCZ LAUNCHES Z-DRIVE R4 CLOUDSERV (2012) http://www.ocztechnology.com/aboutocz/press/2012/481
21. Open Source P2P video Streaming Software (2012) http://www.scvi.net/stream/soft.htm
22. Rosenblum M, Waldspurger C (2011) I/O virtualization. ACM Queue 9(30):30–39
23. Shimpi A (2010) Kingston SSDNow V+100 review. http://www.anandtech.com/show/4010/kingston-ssdnow-v-plus-100-review
24. Warfield A, Hand S, Fraser K, Deegan T (2005) Facilitating the development of soft devices. In: Proceedings of the annual conference on USENIX annual technical conference (USENIX 2005). USENIX, Anaheim, CA, pp 379–382

Dingding Li is a Ph.D student working with Prof. Hai Jin in the Services Computing Technology and System Laboratory (SCTS) at Huazhong university of Science and Technology (HUST). His research is focused around I/O virtualization and cloud computing.

Hai Jin received his B.S., an M.A. and a Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST) in 1988, 1991 and 1994, respectively. Now he is a Professor of Computer Science and Engineering at HUST in China. He is now the Dean of School of Computer Science and Technology at HUST. In 1996, he was awarded German Academic Exchange Service (DAAD) fellowship for visiting the Technical University of Chemnitz in Germany. He worked for the University of Hong Kong between 1998 and 2000 and participated in the HKU Cluster project. He worked as a visiting scholar at the University of Southern California between 1999 and 2000. He is the chief scientist of the 973 project "ChinaV" and the largest grid computing project, "ChinaGrid", in China. His research interests include virtualization technology for computing system, cluster computing and grid computing, Peer-to-Peer computing, network storage, network security, and high assurance computing. He is the member of Grid Forum Steering Group (GFSG). He is a senior member of IEEE and member of ACM.

Xiaofei Liao received his Ph.D. degree in computer science and engineering from Huazhong University of Science and Technology (HUST), China, in 2005. He is now an associate professor in the school of Computer Science and Engineering at HUST. He has served as a reviewer for many conferences and journal papers. His research interests are in the areas of virtualization technology for computing system, P2P system, cluster computing and streaming services. He is a member of the IEEE and the IEEE Computer Society.



Jia Yu gets his master's degree in Computer Science at Huazhong university of science and Technology (HUST) in June 2012. His research is focused around virtualization and computer architecture.