A Multilevel I/O Tracer for Timing and Performance Analysis of Storage Systems in IaaS Cloud

Hamza Ouarnoughi*†, Jalil Boukhobza*†, Frank Singhoff*† *B-com Research Institute of Technology
F-29200 Brest, France
Email: Hamza.Ouarnoughi@b-com.com Stéphane Rubini† †Univ. Bretagne Occidentale UMR 6285, Lab-STICC
F-29200 Brest, France
Email: {boukhobza, singhoff, rubini}@univ-brest.fr

***Abstract*—Data centers are more and more relying on hybrid storage systems consisting of flash memory based storage devices and traditional hard disk drives. Optimal data placement in such hybrid storage systems is a very important issue in the domain of cloud computing and virtualization. This is specially the case when users need that storage systems enforce Quality of Service requirements on I/Os performed, for example for multimedia applications. To characterize Virtual Machine (VM) I/O workload properties such as timing predictability or throughput, monitoring services are necessary on such new architectures. This article presents a multilevel I/O tracer for virtual machines that relies on and complement different state-of-the-art tools. It produces I/O traces at different levels of the Linux I/O software stack. The I/O tracer gives an exhaustive information that allows administrators to precisely characterize virtual machine I/O behavior in terms of percentage of read/write I/Os, percentage of random/sequential, I/O request inter-arrival time, etc. This tool is the first piece towards a middleware whose purpose is to meet user QoS requirements thanks to optimal data placement and migration policies in a hybrid storage system in the context of an IaaS Cloud.**

I. Introduction

Infrastructure as a Service Cloud (IaaS) can be defined as the way of delivering hardware infrastructures (CPU, storage and network) and their associated software (virtualization technology and file system) to the cloud user [1]. In this context, the performance and the energy consumption characteristics of storage systems should be taken into account, in order to offer the best quality of service [2] [3].

One of the key technologies to cope with the storage system performance and energy consumption issues is flash based storage systems [4]. The main characteristics of this kind of storage system are their high throughput, low access latency and low energy consumption. However, they are more expensive then hard disk drives (HDD) and might have a limited lifetime. So, for cost consideration reasons, they cannot replace HDDs in the meantime [5]. Starting from this observation, designers have tried to integrate efficiently flash memory in the memory hierarchy mainly in three ways to propose *Hybrid Storage Systems* [6]: (1) as a main memory extension, (2) as a cache for HDDs, or (3) at the same level with HDDs in the storage system level.

The major issue in hybrid storage architectures is to define an optimal data placement policy. Data placement depends mainly on two parameters: (1) the characteristics of the storage devices, for instance: throughput, latency, power consumption, and lifetime, and (2) the characteristics of the accesses to the devices, for example: type of operations (read/write rate), and I/O pattern (sequential/random percentage). Combining both sets of parameters is important in order to achieve optimal data placement upon a hybrid storage system according to administrator constraints and user required Quality of Service. In the cloud context, the Quality of Service is expressed in the Service Level Agreement terms. Indeed, from a storage system point of view, mainly two metrics define the SLA (or QoS contract) for storage systems: throughput and access latency. Assessing those metrics is mandatory as a Cloud system may host applications with various QoS requirements. For example, a cloud can host at the same time non real-time applications but also real-time applications that require SLA enforcement. This is the case, for instance, when multimedia applications and/or database transactions are ran.

Characteristics of storage devices can be obtained from the corresponding manufacturer data sheets or through some benchmarking tools. However, it is difficult to get information about the processed I/O workload if we do not have a prior idea about the nature of the executed applications. This problem is even more complex in the context of IaaS cloud as we have an additional virtual machine (VM) layer. Indeed, the administrator is not always aware of the nature of applications executed by each VM. In case of multimedia applications, timing performances that are required might be unknown to the administrator. As a consequence, the only way to get the characteristics of the I/O workload of a given VM is through tracing its I/O access to the storage system device.

Many state-of-the-art Linux I/O tracers already exist. They are plugged at different levels of the I/O software stack. Blktrace [7] traces I/O activity at the block level (under the file system). Strace [8] is able to observe the system calls issued by a process. SystemTap [9] explores some specific system call execution and timing. However, each tool individually fails in delivering details about the whole I/O flow of VM executions. In fact, one needs to gather the following information : (1) identifying the VM files to know which users are handling which files, this is of help as the SLA is user defined, (2) identifying the access patterns from the user point of view (in high operating system level) in order to have a clear understanding of how the user accesses his files, and finally (3)

assessing the impact of the I/O access pattern on the underlying storage system to understand how the storage system and different operating systems levels behave according to the I/O workload extracted in (2).

This paper presents an I/O tracer for an IaaS Cloud context. This I/O tracer monitors VMs I/O behavior on a given storage system. It combines and complements different state-of-the-art tracers that operate at different levels of the host operating system in a virtualized environment. This allows to gather detailed information on I/O access of each VM. Information gathered by this I/O tracer is mandatory to analyze resource requirements of the real-time applications ran on the VM.

Three I/O tracing levels are merged with our tool. (1) The first level is the Virtual Machine Monitor (VMM) using the libvirt API [10]. The libvirt API permits to manage VMs and to get information about resource usage. In our case, we use libvirt to get statistics about I/Os on virtual storage devices used by each VM (VM image, second virtual storage devices, etc). (2) The second trace level is the Linux Virtual File System (VFS). The used monitoring tool for this level is Strace [8]. At this level, we can retrieve information about the file system calls executed by each VM monitor running in the system. In the virtualization using Linux as host operating system, a VM appears as a classical Linux process [11]. (3) The last trace level is the block I/O layer. The used tool to trace I/O operations at this level is blktrace [7]. It allows users to get detailed information about the I/O requests received by the block devices. The developed I/O tracer returns VM I/O traces by aggregating heterogeneous results produced in each trace level, in order to give detailed and homogeneous information on the I/O workload patterns related to each VM and user.

The rest of the paper is organized as follows. In the next section, we present the general context of our work, background, and related works. Then, the architecture of the I/O tracer and the different components are described in section 3. In section 4, we present a use case to evaluate our I/O tracer. Finally, we conclude and give perspectives on future work.

II. Background and Related Works

In this section, we present the general context of our contribution. Then we introduce the background of our work, and finally some related works are given.

A. General Context

This work is part of a project that aims to propose a middleware to optimize performance and energy consumption of a hybrid storage system in the context of a distributed Cloud. This middleware will be integrated in an Open Source IaaS cloud manager, such as OpenStack [12] to manage transparently the storage system. The storage management middleware is autonomic and obeys the MAPE-K (Monitor, Analyze, Plan, Execute - Knowledge) reference model [13]. It consists of four MAPE-K components that are: 1) I/O tracer, 2) the I/O workload analyzer, 3) data placement decision maker, and 4) data dispatcher, see Figure 1. The first component provides the I/O traces of a workload executed by a virtual machine on physical storage devices. This first component is the subject of this article. The second component analyzes the produced traces and gives the main characteristics and profile of the studied I/O workload/user in terms of percentage of read/write, random/sequential requests, request average size, etc. The third component decides the optimal placement of data on the different available storage devices according to different objectives, device, and workload characteristics. The objectives and constraints may consist in reducing the energy bill, respecting the SLA and reducing penalties, while device and workload characteristics can be declined in terms of size of the flash device pool, performance characteristics, and I/O workload patterns. The last component dispatches data according to the decided plan.

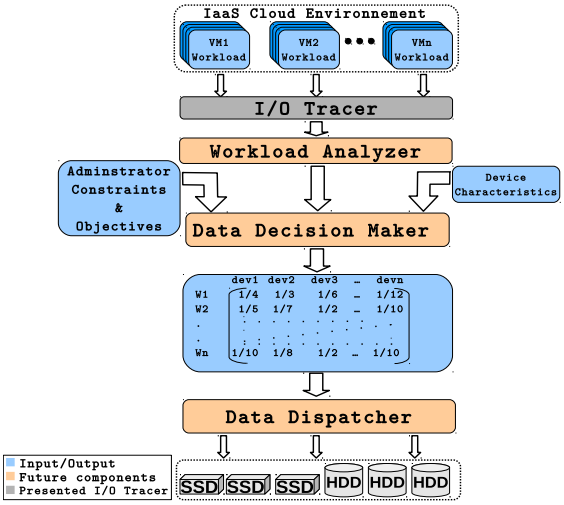

Fig. 1. Hybrid Storage Optimization Middleware

B. Background

As it has been previously underlined, the IaaS cloud offers virtualized hardware infrastructure to the users. In this section, we briefly introduce general concepts of virtualization and Linux I/O software stack.

1) Linux I/O Software Stack: Linux is the most used operating system as a host for virtualized environments [11]. Figure 2 (a) shows the I/O paths in a Linux software stack. Application processes execute I/O operations on files, which passes through three main software levels before reaching the storage device: 1) VFS, 2) file-system, 3) block I/O layer.

2) Virtualization: The concept of virtual machine was developed in the sixties to provide a sharing system of mainframe computer [14]. Real-time cloud applications depend mainly on the timing constraints of virtual machines [15]. A virtual machine is seen as an instance of the physical machine on which it runs by using an abstraction layer called Virtual Machine Monitor (VMM) positioned directly on the hardware or on the host operating system [16]. In the case of Linux hosted virtual machines, the common method to interact with the virtual machine monitor, is to use the libvirt API [10]. This API can be used to develop some virtual machine management

applications. Figure 2(b) shows the architecture of a virtualized environment with a virtual machine management application that uses the libvirt API where the components are: (1) **Node:** it is the physical host system that supports the hypervisor. (2) **Hypervisor:** it is the utility that gives an isolated environment to run virtual machines. (3) **Domains:** domain 0 refers to the host operating system, and the other domains are instances of virtual machines attached to a specified hypervisor.

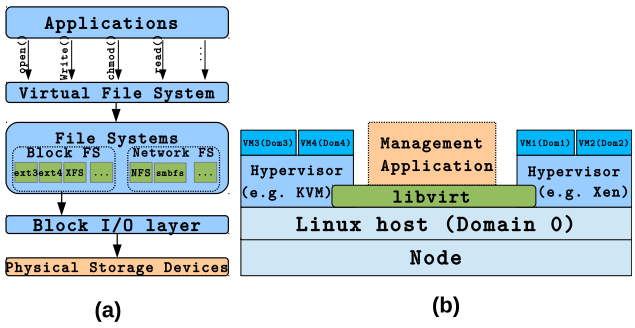

Fig. 2. Linux I/O stack and libvirt-based Management Application

C. Related Works

In the context of IaaS Cloud, monitoring is highly important for both providers and customers. It is mainly used for billing and cloud orchestration [17]. The Cloud monitoring tools such as Ceilometer [18] for OpenStack and Nagios [19] give general quantitative information about the state of IaaS cloud (network activities, system load, etc). To optimize the storage system usage, one needs more details about the I/O operations performed. There are also different monitoring tools at different levels of the I/O software stack system. **blktrace** [7] is a block layer I/O tracing tool which acts in the driver I/O request queue level. It provides detailed information about the state of the I/O request during its execution on the physical storage device. **iostat** [20] is a Linux command that gives statistics about I/O device load and information by observing the activity time of the monitored devices according to their average transfer rate. **iotop** [21] is an I/O monitoring program written in Python. It gives a user interface similar to the top Linux command output, by showing per-process I/O rates. Many other generic tracing frameworks exist, such as **Strace** [8] and **SystemTap** [9] ; they provide trace facilities of user and kernel level functions.

The tools presented above are not dedicated to monitor virtualized environments. Each tool works independently on a predefined system layer. The I/O tracer presented in this work combines and increments different tools at different system levels. The goal is to have a global view of each I/O request path and benefits from the advantages of each tracing level.

III. I/O Tracer Architecture

An I/O request executed by a virtual machine goes through several software layers before reaching the physical storage device. In order to have a precise idea about the I/O access pattern of a given VM, we have to extract information from every layer that requests cross:

At the hypervisor level: we look for the type and the size of I/O requests issued to the virtual storage devices. We also need to identify the files accessed by the different VMs in order to capture access patterns according to user's behavior. This is extracted in our tracer thanks to the libvirt API (see Figure 3).

At the host level: we need to know which I/O operations are generated on the host Virtual File System level once the I/O workload of the VM is applied. For such a purpose, we have used Strace to monitor all system calls executed by the hypervisor. The results are then parsed and filtered to only select I/O related system calls.

At the file-system level: I/O system calls are performed on files that are stored on physical storage devices, and organized using a specific file system. We do not trace at this level, but we need to perform a mapping between files captured in the preceding levels, and block numbers on the storage device. This is done in order to to bridge the gap between the high level file view and low level block device view. We use libext2fs library [22] to extract the structure of each file in the physical storage device

At the I/O block layer: before reaching the storage device, I/Os can be merged and/or scheduled in the block I/O layer, then forwarded to the storage device. At this level, we use blktrace to monitor the block I/O requests. Its output is parsed and filtered to only select the I/O operations performed on the virtual machine storage devices.

The final output is obtained by combining different tracer outputs for a given time interval. This is detailed in the next sections.

Figure 3 presents the detailed architecture of I/O tracer with associated input/output. This figure shows the different levels in the tracing environment. The right hand side shows a layered vision of the virtualized system while the left hand side part shows the different tracing levels.

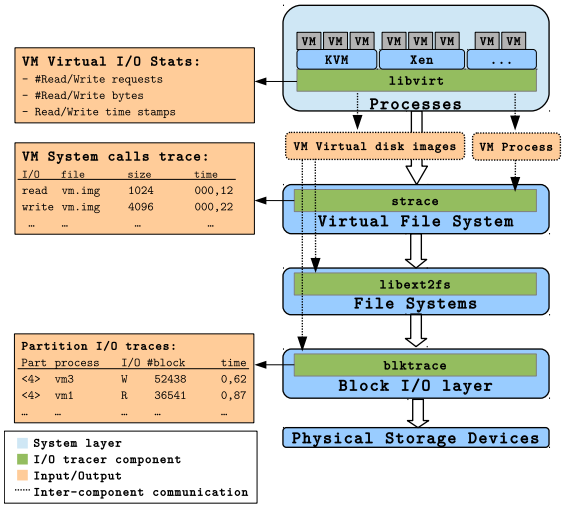

Fig. 3. Architecture of the I/O tracer

The developed I/O tracer is composed of two main parts: (1) a part that synchronizes the different tracers, merges the results using the libvirt API etc, and (2) a part that bridges the information gap between files and blocks relying on the file system library libext2fs.

A. Hypervisor Level

The I/O tracer tracks the path of the I/O request from the virtual machine down to the host physical storage device. The first layer through which I/O requests pass is the hypervisor. By using the libvirt API utilities, one can get statistics about the number of read/write requests executed on the virtual storage devices, and the amount of bytes read or written for each request. Another important information is to track the virtual device image files of virtual machines. I/O requests are issued on files that represent virtual disks. We give these files as an input for the lower levels: file system and block I/O layers. At this layer, we can get the hypervisor process identifier that runs each virtual machine.

Figure 4 shows an example of output file at the hypervisor level, obtained using the first component we developed. Note that the output file does not log events (stamped I/O calls) but gives a summary of performed operations (type of operation, count and size). For the example shown in the figure, the traced virtual machine executes a simple loop of 'dd' Linux command that reads from '/dev/urandom' and writes blocks of 4096 bytes in a file. Each line represents the current state of the virtual devices. In this case, the virtual machine has one virtual storage device (vda) which is the virtual machine image file on the host operating system. The first column represents the virtual device name. The second one is the number of read requests. The third is the total number of read bytes.The forth is the number of write requests followed by the total number of written bytes. Finally the last column is the snapshot time stamp.

Device	#Read	R_Byte	#Write	W_Byte	Time
vda	0	0	0	0	1405521774.02
vda	24	749568	9	475136	1405521775.02
vda	0	0	0	0	1405521776.03
vda	24	749568	9	475136	1405521777.04
vda	0	0	0	0	1405521778.05
vda	7	204800	0	0	1405521779.06
vda	17	544768	9	475136	1405521780.06
vda	0	0	0	0	1405521781.07
vda	24	749568	9	475136	1405521782.08
vda	0	0	0	0	1405521783.09

Fig. 4. libvit stat output file

Another important output at this level is the process identifier (see Table I) and the host target file (see Table II). These are inputs for lower I/O tracer components.

VM Name	Process Name	PID
vm1	kvm	123
vm2	qemu-system-x86_64	456

TABLE I. VM-PROCESS LOOK-UP TABLE

B. VFS Level

System calls executed by each process passes first through the Virtual File System (VFS). As each running VM is a system process (see Table I), one can trace system calls issued by each VM. In our work, VM system calls are captured by Strace. Each process to trace is obtained using the libvirt API at the hypervisor level. The Strace output is parsed in order to obtain the desired I/O information. An intermediate trace file is generated, it contains the I/O system calls traced for a given virtual machine. A trace example is shown in Figure 5.

VM Name	Virtual device	Host target file
vm1	vda	/var/lib/libvirt/images/vm1.img
vm1	vdb	/var/lib/libvirt/images/dev1.img
vm2	sda	/var/lib/libvirt/images/vm2.img

TABLE II. VIRTUAL DEVICE-TARGET LOOK-UP TABLE

SYS_CALL	File	Size	TimeStamp
read	anon_inode:[eventfd]	512	1405680107,28
write	/var/lib/libvirt/qemu/vm1.monitor	57	1405680107,52
read	anon_inode:[signalfd]	128	1405680107,69
read	pipe	16	1405680107,77
write	anon_inode:[eventfd]	8	1405680107,78

Fig. 5. VFS level component output file

The trace file shows the system calls related to the same virtual machine previously investigated (see Figure 4). The first column shows system calls executed on the file given in the second column. The third column gives the request size in bytes, followed by the time stamp.

C. File System Level

In a virtual environment, I/O operations are executed on files that represent virtual disks. In the host operating system, these files are managed by a specific file system. By default, Linux uses ext2/3/4 file systems for the local block storage devices [23]. As shown in Figure 3, the virtual disk files are obtained using libvirt API. So, in order to trace those files at a block level, one needs to perform a mapping between the file references and the block device numbers. To do so, we developed a mapping tool based on libext2fs [22] which allows us to explore the virtual disk file structure on the physical storage device for ext file systems. This tool provides the following information: block numbers, block size and allocated/free blocks for a given file on a given partition.

D. I/O block layer

The last software layer that issues I/O operations before the physical device driver, is the block I/O layer. At this level, an I/O queue is implemented to buffer incoming I/O requests. blktrace permits to have the trace of all I/O requests executed on the physical block device. The obtained trace is filtered to just keep events of the virtual machines and their files. To get a human-readable output, the result output is parsed using blkparse [7].

At the block layer, an I/O request can be remapped (for stacked devices), split (on RAID or device mapper setups), added to the request queue, or merged with a previous entry on the queue [7]. As shown in Figure 6, the block I/O level

Device	Event	IO	StartBlock	Time	Process
/dev/sda6	Q	WS	42728093	0,022144804	kvm
/dev/sda6	Q	WS	42728073	0,022197447	kvm
/dev/sda6	Q	WS	42728599	0,02465816	kvm
/dev/sda6	Q	WS	43250139	0,027069399	kvm
/dev/sda6	Q	WS	51479776	0,664746176	jbd2/sda6-8
/dev/sda6	Q	WS	51479777	0,664758356	jbd2/sda6-8
/dev/sda6	Q	WS	51479778	0,664761627	jbd2/sda6-8
/dev/sda6	Q	WS	51479779	0,664762753	jbd2/sda6-8

Fig. 6. Block I/O level component output file

permits to trace all events that happened on the block device. In this case, blktrace output is parsed then filtered to get only one type of I/O request that is queued I/O requests (Q in the second column). This is achieved in order to avoid duplication of the I/O request and to get the trace of the I/O request sequence before potential scheduling by the I/O scheduler. This allows us to have an idea about the I/O workload rather than its management by the driver. The I/O column gives the request I/O type: 'R' for read, 'W' for write optionally followed by 'S' for synchronous or 'B' for barrier operations.

The developed I/O tracer returns a statistics file that shows the type and the number of I/O traces recorded at each level. Statistics are grouped by time stamp. To get a detailed I/O information about each level, we need to give a time stamp (in seconds) to a post processing function. This is required because the time granularity is different at each level (microsecond in Strace, nanosecond in blktrace, second in libvirt). All tools at different trace level get the time stamps from the same system clock: *getnstimeofday* in blktrace, *gettimeofday* in Strace, and *time()* python function that is based on *gettimeofday* in hypervisor level. To save traces for each level, we use a dictionary of all traces, where the keys are times in seconds.

Times (s)	Read_libvirt	Write_libvirt	Read_strace	Write_strace	Block_Read	Block_Write
1406056802	4071	110427	198	1008	2	17
1406056803	4071	114920	612	2251	0	36
1406056804	4071	120414	263	2899	0	9
1406056805	4071	126953	176	3172	0	25

Fig. 7. General output stats

Figure 7 shows an example of an output file obtained after tracing a VM. To unify statistics for all levels, I/O operations stamped in nano second and micro second at block I/O and VFS level respectively, were grouped by seconds. Each line presents the state of the I/O system at the given second. One has to note that, for each line, the underlying operations are not necessarily the consequences of the operations of higher levels. We have to take into account the presence of caches at different levels (e.g. VFS cache of guest and host system).

Another important result obtained by the I/O tracer is the access pattern of virtual machines to the physical device (see next section). The block I/O level permits to get a precise idea about the I/O request type (read/write), and blocks targeted by each request. This information is highly important to characterize a virtual machine I/O workload. The access pattern can be obtained for a specified time stamp in second (given as a key), or for a time interval. First we convert all time stamps to the biggest time granularity (seconds on hypervisor level), then we give a second as a key to get the global view of an I/O path form the hypervisor level to the block I/O level. This method does not take into account the caches of different levels, thus we miss several I/O operations in low levels. The second method gives a time interval that includes cache flush time of different levels [24]. More detailed results about the access pattern are given in the next section.

IV. Experiments

Our objective is to characterize a virtual machine I/O workload, by tracing its I/O requests on the physical storage device. This is mandatory for the support of applications that require QoS enforcement. Some factors were identified to measure performances of real-time applications in a virtualized environments [25], and to compare virtualization technologies [26]. In this section, we present some case studies where I/O benchmarking tools are executed in order to test the I/O tracer and to show its usefulness.

A. I/O Benchmarking tools and setup

To evaluate the I/O tracer, we have used popular storage related benchmarking tools. Some of these tools offer ready-to-use I/O workload scenarios (e.g. web server, mail server). We used both FileBench [27] and Postmark [28]. For FileBench, we ran four I/O workload models: web server, mail server, file server, and OLTP. The I/O request size was fixed to 1MB and file sizes between 16KB and 2MB. For the Postmark configuration workload, we ran experiments with files whose size ranges between 1KB and 10KB and the I/O request size was 512 bytes. The total number of files was fixed to 50000, with 30000 transactions performed. The Table III shows the used configuration for each benchmark.

Benchmark	Mean File size	I/O Request size
Postmark	1KB-10KB	512B
FileBench File Server	2MB	512KB
FileBench Mail server	16KB	16KB
FileBench Web Server	16KB	1MB

TABLE III. Workloads Configurations

We ran three virtual machines that executed the benchmarks simultaneously, with the same workload configuration. The resource configuration was the same for each virtual machine: 1 VCPU, 1GB of memory, and the size of the virtual storage device was 8GB. All guests and the host ran Linux operating system with kernel version 3.0.2. The hypervisor used for virtual machines was KVM [11]. The test benches were performed with a Dell Precision T3610 Workstation with 4 Cores 3.7 GHz Intel Xeon processor, 16GB of RAM memory and 1TB Hard Disk Drive.

B. Results and discussion

Table IV presents I/O statistics at each I/O trace level as retrieved by our tracer. We have taken a snapshot on the first 20 seconds to include the time at which VFS caches are flushed for both guest and host operating systems.

Benchmark	Trace level	#I/O	VM1	VM2	VM3
Postmark	Hypervisor	Read	56014	51562	66864
		Write	1143842	1134154	1167091
	VFS	Read	306	367	345
		Write	304	856	732
	Block I/O	Read	3400	3028	4204
		Write	46	46	49
FileBench-web	Hypervisor	Read	52010	51674	52360
		Write	640	594	610
	VFS	Read	1598	1511	1865
		Write	487	454	571
	Block I/O	Read	3588	3191	3321
		Write	23	18	23
FileBench-mail	Hypervisor	Read	52388	51898	51912
		Write	718	713	724
	VFS	Read	346	354	294
		Write	173	174	173
	Block I/O	Read	840	837	842
		Write	49	51	50
FileBench-OLTP	Hypervisor	Read	52783	51492	52248
		Write	634	632	602
	VFS	Read	799	809	783
		Write	201	199	200
	Block	Read	3587	3093	3229
		Write	51	48	46

TABLE IV. I/O STATISTICS

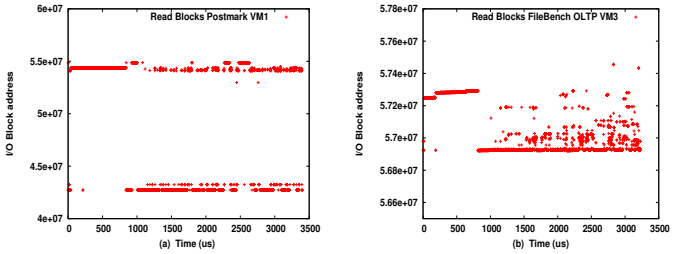

Fig. 8. Accessed I/O Block addresses for read operations

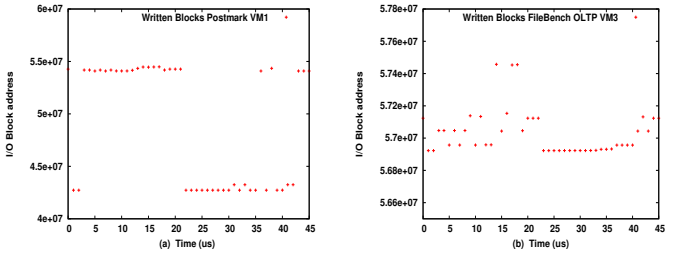

Fig. 9. Accessed I/O Block addresses for write operations

The obtained statistics file shows that we have a write/read ratio close to 1% in all benchmarks at the block I/O level, except the FileBench mail model (line 3) that has a 6% write/read ratio. One can also notice cache effects as the number of I/Os at the block level is reduced as compared to the hypervisor level. The write/read ratio can be a very good indicator when one needs to decide upon integrating a flash memory. Indeed, flash memories Achilles' heel is random writes, and read intensive workloads can highly benefit from the flash performance.

The following figures focus on the I/O patterns as extracted from the block I/O traces for two virtual machines executing two benchmarks. Figure 8(a) shows the read pattern of VM1 executing Postmark. It shows that read pattern of VM1 starts sequentially in the first interval (from the beginning up to $\sim$800 μsec), then it becomes random (more precisely interleaved) targeting two groups of sequential block addresses. This means that the image representing the virtual storage device was fragmented (probably two or three fragments). Figure 9 (a) shows the write I/O pattern of the same virtual machine executing Postmark. It reveals a less random access pattern than the read pattern. We notice an important distance between the block groups (about 10^7 blocks), which might generate a high activity on the HDD. This is also a very interesting result when thinking about flash integration.

Figure 8(b) shows the read pattern of VM3 when executing FileBench OLTP model. We have multiple levels that represent different groups of blocks accessed randomly. We can observe that the image file of the virtual machine storage device is more fragmented than VM1 as block addresses are more scattered. We note the same observation about the randomness of the write access pattern (see Figure 9(b)).

As seen in this section, by extracting information at different levels, the developed tracer can be of a precious help for characterizing VM I/O requests to gather information on the I/O workload patterns and on the system behavior (to what extent does the cache absorb write operations). It allows us to have both a quantitative and a qualitative view of the I/O workload: quantitative in terms of I/Os performed and qualitative in terms of access patterns.

C. Evaluation

To evaluate the I/O tracer, we performed a set of tests using the previous benchmarks and we extracted information about the execution time for one and multiple virtual machines with and without I/O tracer. We ran mail, web, and file server of Filebench by one and three virtual machines using the I/O tracer and without using it. The test with only one virtual machine allowed us to see the overhead of the tracer on the latency. We also tested with three virtual machines to see if the impact of the tracer is increased, for instance as it might modify the I/O request ordering. The following table V shows the average and the standard deviation of the execution time for the different test benches. Indeed, each test was executed 5 times.

Table V shows the evaluation statistics obtained when executing three benchmark models of FileBench on one and three virtual machines by setting the I/O tracer ON and OFF. The results for three virtual machines show the statistics per VM. The obtained results show a small Throughput and CPU overheads except in the Mail server case (14% throughput and 21% CPU time) where the main cause is the high activity in this server (1000 files, 16 threads, and 16K for the I/O request size) compared to the other servers (500 files, 10 threads, and 1MB I/O request size for the file server). This implied a higher tracer activity. In addition, one can observe that for this test, the standard deviation is very high showing an unstable behavior of the storage system facing this workload (stdev of 0.32 for a

Benchmark	Tracer Status	Stats	One VM	Three VMs
Mail Server	Tracer ON	Throughput AVG(MB/s)	4,43	1,4
		Throughput STDV	0,15	0,3
		CPU Time AVG(μs/op)	353	244,66
		CPU Time STDV	18,02	4,72
	Tracer OFF	Throughput AVG(MB/s)	4,56	1,63
		Throughput STDV	0,15	0,32
		CPU Time AVG(μs/op)	324,66	309,66
		CPU Time STDV	9,01	15,30
		Throughput Overhead(%)	2,91	14,28
		CPU Time Overhead(%)	-8,72	20,99
Web Server	Tracer ON	Throughput AVG(MB/s)	146,66	118,23
		Throughput STDV	0,86	2,26
		CPU Time AVG(μs/op)	90	116,66
		CPU Time STDV	0	3,21
	Tracer OFF	Throughput AVG(MB/s)	146,66	125,3
		Throughput STDV	1,02	2,45
		CPU Time AVG(μs/op)	90,33	112
		CPU Time STDV	0,57	2,64
		Throughput Overhead(%)	0	5,63
		CPU Time Overhead(%)	0,36	-4,16
File Server	Tracer ON	Throughput AVG(MB/s)	210	79,03
		Throughput STDV	4,95	1,9
		CPU Time AVG(μs/op)	509,66	488,66
		CPU Time STDV	10,11	5,85
	Tracer OFF	Throughput AVG(MB/s)	211,23	79,96
		Throughput STDV	3,51	1,49
		CPU Time AVG(μs/op)	522,33	514
		CPU Time STDV	4,5	6,24
		Throughput Overhead(%)	0,58	1,16
		CPU Time Overhead(%)	2,42	4,92

TABLE V. EVALUATION STATISTICS

throughput of 1.4MB/s). We also notice a negative overhead for CPU time when running Web server benchmark with the tracer activated. This value stays very small as compared to the mean values meaning that the impact of the tracer is negligible. In fact, except those values, the overhead of the tracer is always smaller than 10% and most of the time smaller than 5%.

V. CONCLUSION

This paper presents a new I/O tracer to be used in IaaS Cloud. The objective behind the design of this tracer is to monitor the user and the VM I/O accesses on multiple levels of the system I/O software stack. This is performed in order to design efficient data placement methods on hybrid storage systems. Data placement in an IaaS Cloud should take into account many factors related to both the client side (performance predictability, SLA, penalties, etc) and the administrator side (cost of the storage system, energy efficiency, storage device heterogeneity and characteristics, etc). The developed I/O tracer links high level I/Os performed on the VMs and I/O blocks sustained by the storage device. It allows users to identify individually the I/O patterns of VMs and applications. It is based on two state-of-the-art tracers (blktrace and Strace) in addition to two developed tools: one that is built with the libvirt API to extract the VM usage, and a second that links high level file identifiers to low level block numbers. A subset of tools was also developed to merge the results of all the components of the tracers and to produce relevant information for the Cloud administrators. We plan to integrate this tool to the Openstack IaaS Cloud management system. Future works should now investigate how performance information gathered by the proposed I/O tracer can be used to build resource management mechanisms to enforce deterministic performances such as timing constraints or throughput.

ACKNOWLEDGMENT

This work has been achieved within the Institute of Research & Technology b<>com, dedicated to digital technologies. It has been funded by the French government through the National Research Agency (ANR) Investment referenced ANR-A0-AIRT-07.

REFERENCES

[1] S. Bhardwaj, L. Jain, and S. Jain, "Cloud computing: A study of infrastructure as a service (iaas)," *International Journal of Engineering and Information Technology*, pp. 60 – 63, 2010.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[3] J. B. Carter and K. Rajamani, "Designing energy-efficient servers and data centers," *IEEE Computer*, vol. 43, no. 7, pp. 76–78, 2010.

[4] D. Roberts, T. Kgil, and T. Mudge, "Integrating nand flash devices onto servers," *Commun. ACM*, vol. 52, no. 4, pp. 98–103, Apr. 2009.

[5] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to ssds: Analysis of tradeoffs," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 145–158.

[6] J. Boukhobza, *Data Intensive Storage Services for Cloud Environments*. GI Global Editor, 2013, ch. Flashing in the Cloud: Shedding some Light on NAND Flash Memory Storage Systems.

[7] A. Brunelle and J. Axboe. (2007) blktrace user guide. [Online]. Available: http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html

[8] D. V. Levin, R. McGrath, and W. Akkerman. (2014) strace. [Online]. Available: http://sourceforge.net/projects/strace/

[9] "IBM redbooks SystemTap: Instrumenting the linux kernel for analyzing performance and functional problems," Aug. 2009. [Online]. Available: http://www.redbooks.ibm.com/abstracts/redp4469.html

[10] D. Coulson, D. Berrange, D. Veillard, C. Lalancette, L. Stump, and D. Jorm, "libvirt 0.7.5, application development guide," Red Hat, Inc, Tech. Rep. 02/680r0, Jul. 2010.

[11] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, Ottawa, Ontario, Canada, Jun. 2007, pp. 225–230.

[12] R. C. Computing. (2014) Openstack: Open source software for building private and public clouds. [Online]. Available: https://www.openstack.org/

[13] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing –degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 7:1–7:28, Aug. 2008.

[14] S. Nanda and T. cker Chiueh, "A survey of virtualization technologies," Tech. Rep., 2005.

[15] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *Journal of Systems Architecture*, vol. 60, no. 9, pp. 726 – 740, 2014.

[16] S. T. King, G. W. Dunlap, and P. M. Chen, "Operating system support for virtual machines," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '03. Berkeley, CA, USA: USENIX Association, 2003, pp. 6–6.

[17] G. Aceto, A. Botta, W. De Donato, and A. Pescape, "Survey cloud monitoring: A survey," *Comput. Netw.*, vol. 57, no. 9, pp. 2093–2115, Jun. 2013.

[18] OpenStack. (2014) Ceilometer. [Online]. Available: https://wiki.openstack.org/wiki/Ceilometer

[19] N. Enterprises. (2014) Nagios overview. [Online]. Available: http://www.nagios.org/about/overview/

[20] O. Cordes. (2014) iostat. [Online]. Available: http://iostat.sourceforge.net

[21] G. Chazarain. (2014) iotop. [Online]. Available: http://guichaz.free.fr/iotop/

[22] T. Tso. (2005) The ext2fs library. [Online]. Available: http://www.giis.co.in/libext2fs.pdf

[23] R. Card, T. Ts'o, and S. Tweedie, "Design and implementation of the second extended filesystem," in *Proceedings of the First Dutch International Symposium on Linux*, vol. 1, Amsterdam, Dec. 1994.

[24] D. A. Rusling, *The Linux Kernel*, G. P. Licence, Ed. The Linux Documentation Project, 1999.

[25] M. García-Valls and P. Basanta-Val, "Benchmarking communication middleware for cloud computing virtualizers," in *REACTION'13*, 2013, pp. 13–18.

[26] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machinesand linux containers," IBM Research Division, Tech. Rep., 2014.

[27] P. Sehgal, V. Tarasov, and E. Zadok, "Evaluating Performance and Energy in File System Server Workloads," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. San Jose, CA: USENIX Association, February 2010, pp. 253–266.

[28] A. Koto, H. Yamada, K. Ohmura, and K. Kono, "Towards unobtrusive vm live migration for cloud computing platforms," in *Proceedings of the Asia-Pacific Workshop on Systems*, ser. APSYS '12. New York, NY, USA: ACM, 2012, pp. 7:1–7:6.