Billing the CPU time used by system components on behalf of VMs

Boris Teabe, Alain Tchana, Daniel Hagimont
Institut de Recherche en Informatique de Toulouse (IRIT)
Toulouse, France
first.last@enseeiht.fr

***Abstract*—Nowadays, virtualization is present in almost all cloud infrastructures. In virtualized cloud, virtual machines (VMs) are the basis for allocating resources. A VM is launched with a fixed allocated computing capacity that should be strictly provided by the hosting system scheduler. Unfortunately, this allocated capacity is not always respected, due to mechanisms provided by the virtual machine monitoring system (also known as hypervisor). For instance, we observe that a significant amount of CPU is consumed by the underlying system components. This consumed CPU time is not only difficult to monitor, but also is not charged to VM capacities. Consequently, we have VMs using more computing capacities than the allocated values. Such a situation can lead to performance unpredictability for cloud clients, and resources waste for the cloud provider. In this paper, we present the design and evaluation of a mechanism which solves this issue. The proposed mechanism consists of estimating the CPU time consumed by the system component on behalf of individual VM. Subsequently, this estimated CPU time is charged to VM. We have implemented a prototype of the mechanism in Xen system. The prototype has been validated with extensive evaluations using reference benchmarks.**

***Keywords*-Cloud computing; Virtual machines; resources; computing capacities**

I. Introduction

Cloud Computing enables remote on-demand access to a set of computing resources through infrastructures, so-called Infrastructure as a Service (IaaS). The latter is the most popular cloud model because it offers a high flexibility to cloud users. In order to provide isolation, IaaS clouds are often virtualized such that resource acquiring is performed through virtual machines (VMs). A VM is launched with a fixed allocated computing capacity that should be strictly provided by the hosting system scheduler. The respect of the allocated capacity has two main motivations: (1) For the customer, performance isolation and predictability [7], [15], i.e. a VM performance should not be influenced by other VMs running on the same physical machine. (2) For the provider, resource management and cost reduction, i.e. a VM should not be allowed to consume resources that are not charged to the customer. Unfortunately, the allocated capacities to VMs are not always respected [1], [10], due to the activities of some hypervisor system components (mainly network and disk drivers) [21], [24]. Surprisingly, the CPU time consumed by the system components is not charged to VMs. For instance, we observe that a significant amount of CPU is consumed by the underlying system components. This consumed CPU time is not only difficult to monitor, but also is not charged to VM capacities. Therefore, we have VMs using more computing capacities than the allocated values. Such a situation can lead to performance unpredictability for cloud clients, and resources waste for the cloud provider. We have highlighted this issue in a previous work [24]. In this paper, we complete our previous analysis [24] by proposing a system extension which determines the CPU time consumed by the system components on behalf of the individual VM in order to charge this time to each VM. Therefore, we significantly reduce performance disturbance coming from the competition on the hosting system components. Note that most researches [12], [13] in this domain have investigated performance disturbance at a micro-architectural level (e.g. processor caches). Our extension is complementary to them since it addresses another level of contention. A prototype has been implemented in the Xen system and extensive evaluations have been made with various workloads (including real data center traces). The evaluations demonstrate that:

- without our extension, performance isolation and resource management can be significantly impacted
- our extension can enforce performance isolation for the customer and prevent resource leeks for the provider
- intrusivity of our implementation is negligible, near zero

The rest of the article is structured as follows. Section II introduces the necessary background for the paper. Section III presents the motivations for this work. Section IV overviews our contributions. The implementation is depicted in Section IV-C. An evaluation is reported in Section V. The latter is followed by a review of related work in Section VI, we present our conclusions and perspectives in Section VII.

II. Background

The contributions described in this paper have been implemented in the Xen system (the most popular open source virtualization system, used by Amazon EC2). This section presents an overview of virtualization technologies in Xen. This presentation only covers virtualization aspects which are relevant to our study: I/O virtualization, and CPU allocation and accounting.

A. I/O virtualization in Xen

In the Xen para-virtualized system, the real driver of each I/O device resides within a particular VM named "*driver domain*" (DD). The DD conducts I/O operations on behalf of VMs which run a fake driver called *frontend* (FE), as illustrated in Fig. 1. The frontend communicates with the real driver via a *backend* (BE) module (within the DD) which allows multiplexing the real device. This I/O virtualization architecture is used by the majority of virtualization systems and is known as the "*split-driver model*". It enhances the reliability of the physical machine (PM) by isolating faults which occur within a driver. It functions as follows.

The real driver in the DD can access the hardware but interrupts are handled by the hypervisor. The communication between the DD and the hypervisor, and between the DD and a domU are based on event channels (EC1 and EC2 in Fig. 1).

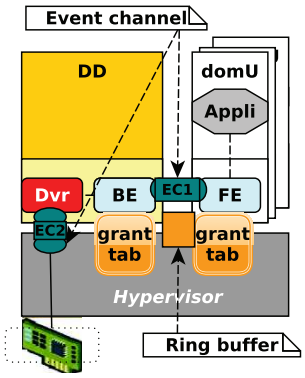

Figure 1: The split-driver model for I/O virtualizign

An I/O request issued by a guest OS (domU) creates an event on EC1. The data are transmitted to the BE in the DD through a ring buffer (based on memory pages shared between the domU and the DD). The BE is then responsible for invoking the real driver.

An I/O request received by the hardware (as an interrupt) generates an event on EC2. From this event, the DD generates a virtual interrupt which is handled by the real driver. The kernel in the DD is configured so that the BE is the forwarding destination for any I/O request.

B. CPU allocation and accounting

In the Xen system, the Credit scheduler is the best fit for cloud platforms where a client books for an amount of computing capacity which should be guaranteed by the provider without wasting resources. Therefore, our contributions only consider this scheduler. The latter works as follows. A VM v is configured at the start time with a credit c which should be ensured by the scheduler. The scheduler defines *remainCredit*, a scheduling variable, initialized with c. Each time a v's virtual processor (vCPU) is scheduled on a processor, (1) the scheduler translates into a credit value (lets say *burntCredit*) the time spent by v on that processor. (2) Subsequently, the scheduler computes a new value for *remainCredit* by subtracting *burntCredit* from it. When *remainCredit* reaches a lower threshold, the VM is no longer allowed to access a processor. Periodically, the scheduler increases the value of *remainCredit* for each VM according to its initial credit c to give it a chance to become schedulable.

More formally, on a PM p, if v has n vCPUs (noted $vCPU_1$ to $vCPU_n$), $burntCredit(vCPU_i)$ and $pmProcessTime(p, v)$ provide respectively the aggregated processing time used by $vCPU_i$ and the time elapsed since the start-up of v, then the Credit scheduler goal it to satisfy the following equation at each scheduling time:

$$c = \frac{\sum_{i=1}^{n} burntCredit(vCPU_i)}{pmProcessTime(p, v)} \quad (1)$$

From the above description we can see that the processing time used by the DD to operate I/O requests on behalf of a VM is not charged to the VM by the scheduler. This is the source of several problems in the cloud.

III. Motivations

A. Problem statement

According to the split-driver model, we have seen that I/O requests are handled by both the hypervisor and the DD on behalf of VMs. Therefore, they use their own processing time (hereafter referred to as "*system time*") to perform operations on behalf of VMs. Current schedulers (including Credit) do not consider this system time when accounting a VM's processing time. This processing time is distributed as follows:

- T_1: within the hypervisor for handling hardware interrupts.
- T_2: within the DD's kernel for handling virtual interrupts and transferring I/O requests between the real driver and the backend.
- T_3: within the backend for multiplexing and delivering/receiving I/O requests to/from the frontend. It includes the processing time used at the hypervisor level for shared pages grant transfer (initiated with hypercalls).

More formally, the Credit scheduler does not take into account $T_1+T_2+T_3$ in equation 1. $T_1+T_2+T_3$ is significant when VMs perform intensive network or disk I/Os, which is the case for many cloud applications.
For illustration, we performed some experiments in our private cluster. It consists of HP machines with Intel Core i7-3770 CPUs and 8 Gbytes of memory. The PMs are linked to each other with a gigabyte switch. We used Xen version 4.2.0. The dom0 is used as the DD (with a credit of 25). We experimented with two benchmarks: (1) a web application based on wordpress for network evaluation, and (2) linux *dd* command for writing data to a portion of a VM disk. The VM we used is configured with a single vCPU (pinned to a dedicated processor, different from the DD's processor).

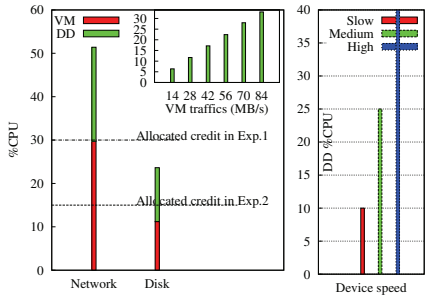

Figure 2: DD processing time used on behalf of VMs

The credit of the VM is set to 30 when running the network benchmark and 15 for the disk benchmark. Fig. 2 (left) plots the results of these experiments. We can see that an important CPU load (28% for network and 12% for disk) is used by the DD to perform I/O requests on behalf of the VM. The VM aggregated load (sum of its CPU load and the DD load) is significantly higher (CPU load over the horizontal dashed line) than the expected one (the VM credit). Fig. 2 (left/top-right-corner) shows the evolution of the DD load when the VM initiated network traffic varies, and Fig. 2 (right) shows its evolution when the speed of the disk device changes. These experiments confirm that the CPU time consummed in the DD is significant and may also vary significantly.

B. Consequences

The fact that, in today's schedulers, the CPU time consumed by system components is not charged to VMs can be problematic for cloud clients (performance predictability) and cloud provider (resource and money waste). If the DD's computing capacity is limited, the performance isolation is compromized as VM performance can be influenced by other VMs which shared the DD resources. Otherwise, if the DD's computing capacity is unlimited, the resource management (and especially VM consolidation) is affected. These two cases are considered in the following subsections.

Performance unpredictability: We consider here that the DD's computing capacity is limited. In a cloud, when a client books an amount of resource for his VM, he expects a predictible and reproductible performance level for his applications. As we have seen, the aggregated amount of resources effectively used by a VM depends on some shared components (the DD and the hypervisor) of the virtualization system. Knowing that the amount of resource available for the DD and the hypervisor is shared among tenant VMs, the client application performance is unpredictable.

Resource and money waste: Giving an infinite resource to the DD without accounting and charging its processing time to client VMs is like giving a blank cheque to clients. This is a shortfall in terms of money for the provider. In addition, without accounting $T_1 + T_2 + T_3$, the scheduler accelerates the VM de-consolidation (moving a VM from an overloaded PM to an underloaded PM) rate, which results in the use of more resources than needed.

IV. Contributions

In this paper, we propose a solution which overcomes the problem identified in the previous section. This solution, implemented at the scheduler level (in the virtualization system), aims at satisfying the following equation, instead of equation 1:

$$c = \frac{\sum_{i=1}^{n} burntCredit(vCPU_i) + T_1 + T_2 + T_3}{pmProcessTime(p, v)} \quad (2)$$

Although the solution is relatively easy to label, its implementation should face the following challenges:

- Accuracy. How to accurately account per VM system time knowing that both the hypervisor and the DD are shared among several tenants?
- Overhead. The processing time needed to run the solution should be negligible.
- Intrusion. The solution should require as few modification as possible within the client VM kernel.

A. General approach

We propose an implementation which takes into account all the challenges listed above. This implementation mainly relies on calibration. It is summarized as follows. First of all, the provider measures for each PM type the system time (noted t=$T_1 + T_2 + T_3$) needed to handle each I/O request type (see section IV-B). This step is carried out once and the measurements are made available to the scheduler (see below). The DD is modified in order to count per VM and per I/O request type, the number of I/O requests (noted nb_{req}) it has handled. This modification is located in the backend, which is the ideal place to track all I/O requests. Subsequently, the DD periodically sends the collected information to the scheduler. This is realized using an existing hypercall ($gnttab\_batch\_copy$), which is invoked by the backend when dealing with grant tables). Using an existing hypercall avoids the introduction of any overhead. When the scheduler receives the collected information, it computes (based on t, evaluated during the calibration phase) the CPU time used by the DD on behalf of each VM: $nb_{req} \times t$. This system time is then charged to the VM. This is done by balancing the system time among all VM's vCPUs (to avoid the penalization of a single vCPU). The next sections give more details about the calibration and the implementation in the Xen Credit scheduler.

B. Calibration

The calibration phase consists in evaluating $t = T_1 + T_2 + T_3$, the CPU time needed to handle each I/O request type. To this end, we place sensors both at the entry and the exit of each component. We decided to ignore T_1 since it is very slight (showed by [3]). Evaluations in section V confirm that

ignoring T_1 is acceptable. Therefore $t \approx T_2 + T_3 = T_{DD}$. We implemented a set of micro-benchmark[1] applications to accurately calibrate T_2 and T_3. The next sections present the rigorous and exhaustive investigations we carried out. For each I/O device type, we consider all factors which could impact the system time. The most important factors are:

- the virtualization approach. For each I/O device, Xen provides several ways to configure how its real driver interacts with the backend.
- the I/O request type (e.g read/write, send/receive). I/O requests do not follow the same path according to their type.
- the request size. I/O requests have different size.

1) Network calibration: We implemented in C a sender/receiver application based on UDP to calibrate the cost of handling a network operation in the DD. The sender always sends the same frame to the receiver. Both the sender and the receiver are within the same LAN.

Virtualization configuration: Xen provides 3 possible network configurations for the DD. These are bridging, routing and NATing. Bridging is the most used configuration. The path (between the real driver and the backend) taken by frames through the Linux network stack differs for each mode. Routing and NATing use a very similar path while it is different for bridging. As illustrated in Fig. 3 (leftmost), the Routing requires more CPU time (e.g. for frame header analysis, packets fragmentation/defragmentation) than Bridging (which does not go beyond the second level of the ISO model). Bridging is used in the rest of the article, unless otherwise specified.

The I/O request type: Packet transmission (when a VM sends a packet) and packet reception (when a VM receives a packet) do not follow the same path in the backend [3]. Therefore, the processing time needed to handle a network packet within the DD depends on its direction. As illustrated in Fig. 3, transmission is less expensive than reception. This difference is located in the backend which performs notably one more hypercall when a packet is received. A worrying observation is that even if a VM does not host a network application, it could be concerned by a load generated within the DD. In fact, since the DD is not aware of the applications within a VM, it always handles incoming requests regardless if the VMs are expecting I/O activities or not. This can be used by malicious users to saturate the DD [4].

Packet size: As shown in the four rightmost curves of Fig. 3, the cost of handling a network packet in the DD varies with its size (even if it is within the MTU). This variation comes from the real driver which includes data copies. By considering packet size, our calibration is also effective with other network technologies such as jumbo.

[1]Note that, once both the hypervisor and the DD are patched (for including sensors), a calibration round does not take a lot of time (about 5 minutes).

2) Disk calibration: We used the Linux *dd* command to calibrate disk operations.

Virtualization configuration: Xen provides different ways to configure how a VM disk is managed in the DD. These are *tap*, *qdisk*, and *phy*. The latter is the most used configuration. The configuration mode influences the processing time used by the DD. *tap* as well as *qdisk* requires much more processing time than *phy*. We use *phy* in the rest of the article.

The I/O request type: Disk operations are read and write. Unlike the network, they are always initiated by the VM. According to the Linux and the backend source code, their processing follows the same path and they require the same processing time. Therefore, our implementation does not distinguish the disk operation type.

Packet size: A disk request size is upper bounded by the memory page size (e.g. 4KB). A request which arrives at the backend is fragmented into several *bio* data structures, the processing unit in the Linux kernel. Any bio is of the same size, configured at compilation time. Thus, our calibration is performed at the bio granularity. Therefore, we evaluate on the one hand the time used to build a bio (noted $T_{bio\_compute}$) from a disk request. On the other hand, we evaluate the time needed to submit a bio (noted $T_{bio\_handling}$). In order to avoid disk speed effects (shown in Section III-A), our calibration takes place when the bio is placed within the real driver buffer. Fig. 4 shows the calibration results for our experimental environment.

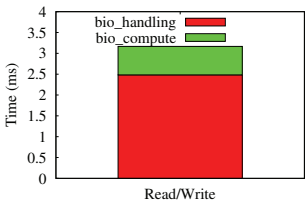

Figure 4: Disk calibration

C. Implementation

This section presents the implementation of our solution in both the DD's kernel (version 3.13.11.7) and the Xen hypervisor (version 4.2.0). This implementation is not intrusive for client VMs since it only requires the modification of the backend and the hypervisor. Regarding the former, new data structures have been introduced for storing information on the number of handled I/O requests. Benefiting from the existence of the $gnttab\_batch\_copy$ hypercall performed at the end of each I/O request, the backend sends the content of its data structures to the hypervisor. This is done periodically after a configured number of handled requests. Regarding the hypervisor, we have modified the VM data structure (*struct domain*) so that it contains the CPU time used by the DD on behalf of a VM: net_{debt} and $disk_{debt}$. These variables give the debts that the VM must repay. We have also added a linked list of VM debts which buffers incoming information from the DD (see below). The new

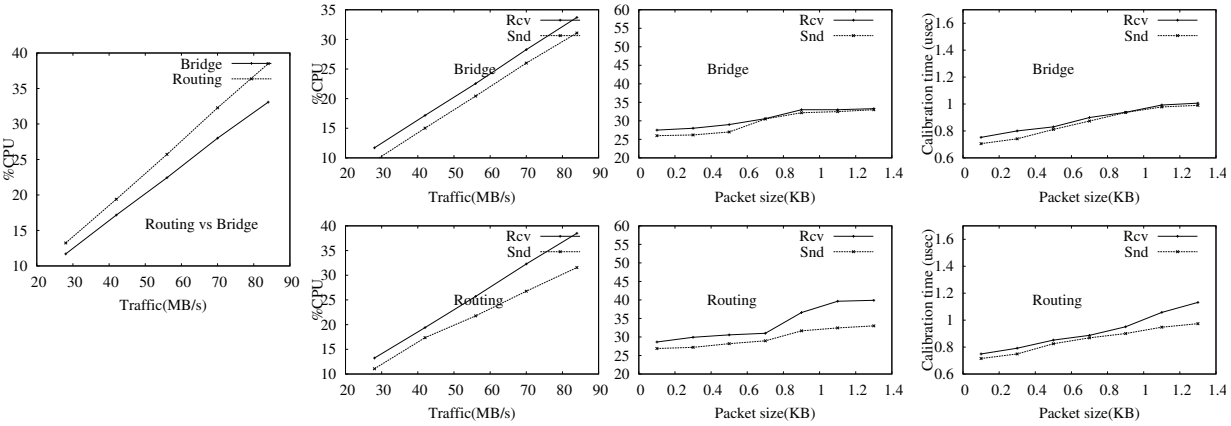

Figure 3: Network calibration

Credit scheduler is implemented in schedule.c. From the calibration results, it knows the cost of performing any I/O request type from different size[2] (in the case of the network) in different situations (routing, NATing, and bridging for the network; qdisk, tap, and backend for the disk). The new scheduler works as follows. The information sent by the backend is buffered in the linked list presented above. When the scheduler wakes-up, it parses the linked list and for each VM, it distributes its debts (by subtraction) to all its vCPUs. This is done before invoking the scheduling function which chooses the eligible vCPU. Note that distributing a VM's debts could require many scheduling round since Xen imposes a lower threshold for vCPU credit. We have also provided a new version of procps [6] so that cloud clients can know within their VM the amount of load used by the DD on behalf of their VMs.

V. Evaluations

This section presents the evaluation results of our solution. We evaluate the following aspects: (1) the overhead of the solution, (2) its efficiency regarding performance predictability, (3) its efficiency regarding resource waste minimization, and (4) its application to other hypervisors.

A. Experimental environment

The experimental environment is the same as in Section III-A. The DD's computing capacity is configured to 30% of the processor (credit 30). VMs are configured with a single vCPU (pinned to a dedicated processor, different from the one used by the DD).

B. Overhead and scalability

As its description suggests, our solution introduces a negligible overhead (near zero). It could have been otherwise if for example we had introduced a new hypercall for informing the hypervisor level (about the number of I/O requests handled by the DD). We avoided this approach by using an existing hypercall.

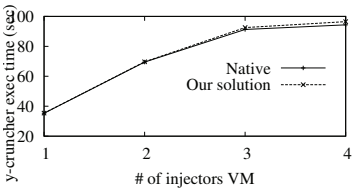

Figure 5: Overhead evaluation.

Fig. 5 presents the results of the experiments we have performed to validate our assertions. We run a witness VM (noted $v_{witness}$) hosting an application (y-cruncher [17]) which is both CPU and memory bound. $v_{witness}$ is configured with a single vCPU pinned to the same processor as the DD (having also a single vCPU). Both the DD and $v_{witness}$ have access to the entire processor capacity. The PM also hosts a set of client VMs (called injectors) whose number varies during the experiments (to increase the traffics within the DD). Each client VM runs the same web application based on wordpress. Experiments were carried out in two contexts. The first context is based on native systems. It is the baseline. The second context uses our solution in which the mechanism of charging debts to VMs is disabled (in order to keep injectors with the same behaviours as in the baseline context). Fig. 5 shows that our solution does not introduce overhead since the $v_{witness}$ performance is the same in the two contexts.

C. Accuracy

1) Micro-benchmark evaluation: This section presents the evaluation results of the accuracy of our solution for both network and disk workloads using micro-benchmark. We experimented two benchmarks: (1) a web application based on wordpress for network evaluation, and (2) linux dd command for writing data to a portion of a VM disk. The VM credit is set to 30 when running the network benchmark and 15 for the disk benchmark. The experiment is realized in two contexts: with our solution and with the native Xen system. We show the ability of our approach to ensure that

[2]The cost of not calibrated packet sizes is obtained by interpolation.

the aggregated CPU consumed by a client VM remains within its booked credit, which is not the case with the native system. This also allows to guarantee performance predictability. The leftmost curve in Fig. 6 presents the results of these experiments. We can see that using our solution, the aggregated CPU load of the client VM is about the value of its credit (30 or 15). The margin of error is negligible. The three rightmost curves in Fig. 6 focus on the network case. They present results for performance predictability. The second curve highlights the issue of performance unpredictability in the Xen system when two VMs share the DD (the throughput of the indicator VM goes from 1200req/sec when the VM is alone to 800req/sec when it is colocated). The third curve shows the results of the same experiment when our solution is used. We can see that the VM has the same performance, about 800req/sec. The latter represents the throughput the VM should provide regarding its booked credit. Indeed, our implementation avoids the saturation of the DD since its allocated credit was enough for handling VMs traffics when their aggregated CPU load stay within their booked credit. The rightmost curve presents the evaluation of our solution when several VMs performing network requests are colocated. We can observe that the indicator VM performance is always the same regardless the number of colocated VMs.

2) Complex benchmark evaluation: This evaluation demonstrates the effectiveness of our solution on a set of realistic complex workload provided by SPECvirt sc2010 [29] (SPECvirt for short). The latter is a reference benchmark which is widely used by cloud providers for evaluating their platform. It implements the vast majority application types which run in the cloud. It is composed of three benchmarks: SPECweb2005 (web application), SPECjAppserver2004 (JEE application), and SPECmail2008 (mail application). Each benchmark defines its performance indicator: average response time for both SPECweb2005 and SPECmail2008, and JIOPS (Java Operation Per Seconds) for SPECjAppserver2004. We executed each benchmark within a dedicated VM having a booked credit of 30. We have experimented two colocation scenarios: (1) each benchmark running alone atop the physical machine, and (2) all benchmarks running concurrently. Fig. 7 shows the results (performance at the top and CPU load at the bottom) of these experiments. Unlike SPECweb2005 and SPECmail2008, SPECjAppserver2004 generates a negligible load within the DD (green bars in the first two bottom curves are significantly more higher than those in the bottom rightmost curve). This is because SPECjAppserver2004 does not perform a lot of IO requests in comparison with SPECweb2005 and SPECmail2008. Therefore, the native implementation of Xen (as well as our solution) provides almost the same performance for SPECjAppserver2004 when it runs either alone or together with other benchmarks (all bars in the top rightmost curve have almost the same height).

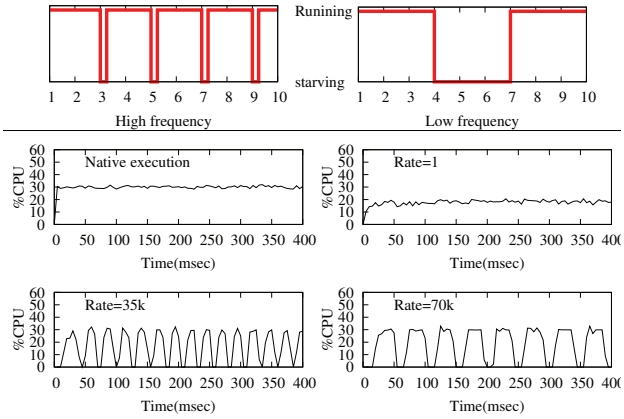

Figure 8: Reporting rate impact

In contrast, Xen does not ensure performance predictability neither for SPECweb2005 (the second and the fourth bars in the top leftmost curve does not have the same height) nor for SPECmail2008 (the second and the fourth bars in the top middle curve doest not have the same high). With a negligible margin of error, our solution guarantees performance predictability (e.g. the first and the third bars in the top leftmost curve have the same high). This is achieved by enforcing each benchmark to only consume its booked capacity (both the first and the third bars in the bottom curves are close to 30% CPU).

D. DD to hypervisor: reporting rate

The DD regularly informs the hypervisor level after a (configurable) number of I/O requests. The choice of this number is important for interactivity. If this number is too high, repaying debts on vCPUs burnt credits will require several scheduling round since Xen credit scheduler imposes a low level threshold for burnt credits. It can impact the interactivity of VMs with latency sensitive applications (I/O intensive workloads), alternating between long phases of activity and starving (debts charging) as illustrated in Fig. 8. We evaluated three arbitrary rates (1, 35000, and 70000) and compared the fluctuation of the CPU when using a native systems (unmodified Xen). We can see that a small value such as 1 is ideal, especially as it does not incur any overhead. Note that regardless this phenomenon, the VM does receive its entire credit.

E. Resource and money saving

To evaluate the benefits that our solution brings in terms of resource saving, we performed an experiment using real traces of two data centers named respectively UNIV1 and UNIV2 [8]. These traces contain network traffic statistics of the data centers during a period of time. A machine is composed of several network devices. Thus, the same PM can be involved in several networking operations at the same time. We used these traces to simulate network traffics of

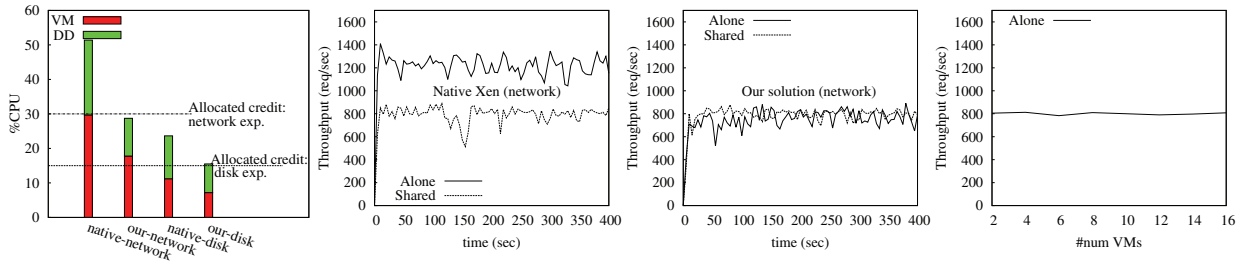

Figure 6: Accuracy of our solution using micro-benchmark

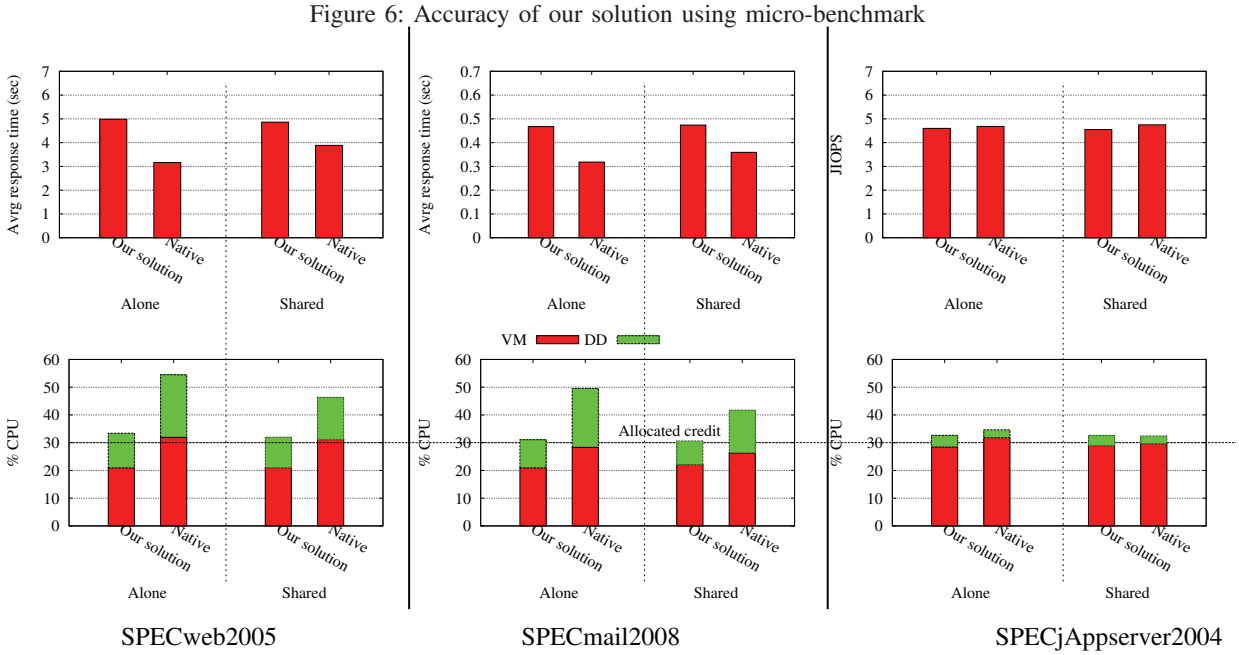

Figure 7: Accuracy of our solution using a complex

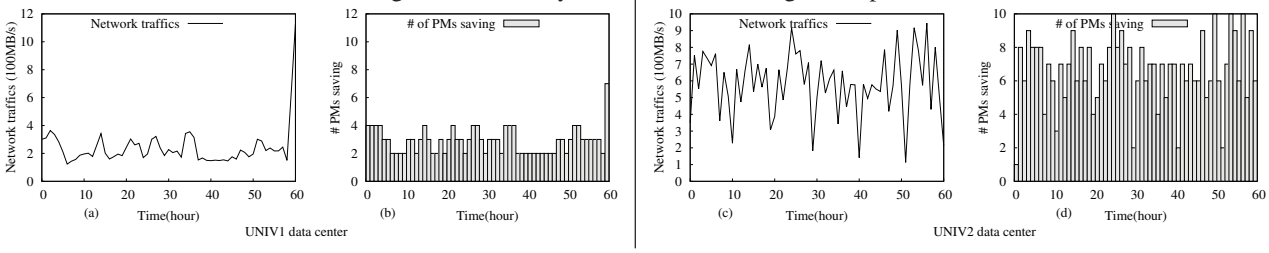

Figure 9: Resource and money saving

two cloud platforms hosting several VMs as follows. We consider that a PM has the same characteristics as a p_1 in our private cluster. Each IP address found within the traces is seen as a VM of type m3.medium from Amazon EC2 nomenclature. Thus, we have up to 6035 VMs for UNIV1 and 5461 for UNIV2 in the entire traces, each PM providing a hosting capacity of up to 8 VMs (a PM has 8 processors). From these assumptions and using our calibration results, we have simulated the amount of resources saved (respectively wasted) by our solution (respectively by the native implementation) in the simulated clouds. Fig. 9 presents the results of these simulations. Fig. 9 (a) and (c) respectively present the intensity of network traffics in UNIV1 and UNIV2. The latter contains the most important traffic. From this traffic information, we computed (using calibration results) the total amount of CPU load they induced in the DD. Lets say t_{DD} represents this number at a given time. Therefore, the amount of resources (in terms of PM) saved by our solution is given by the following formula: $\frac{t_{DD}}{800}$ (the maximum CPU load of a PM is 800%). Fig. 9 (b) and (d) respectively present the resource saving in UNIV1 (an average of about 2 PM/hour) and UNIV2 (an average of about 6 PM/hour). Still on the basis of t_{DD}, we can evaluate the benefits in terms of money. Without our solution, t_{DD} can be seen as what

we have called the "blank cheque" given to clients. If we convert t_{DD} into a number of m3.medium VM instances, the "blank cheque" can be evaluated by the following formula: $\frac{t_{DD}}{100}$ (a m3.medium VM is allocated a unique processor). Knowing that a m3.medium VM instance is $0.070 hourly in Amazon at the time of writing, the operating loss for our simulation period is amounted to about $114 for UNIV1 and $249 for UNIV2. The extrapolation of these results in a full scale cloud (thousands of machines hosting millions of VMs) would show very significant benefit. Without a precise accounting of system time, the provider has to integrate these costs in the global operating budget of the data center.

F. Other hypervisors

We discuss in this section the possibility of applying our work to other hypervisors. Although the split-driver based architecture is not used by all hypervisors, they are subject to have the problem raised in this paper. In fact, all components outside the VM where I/O requests transit consume CPU time on behalf of the VM. Microsoft Hyper-V hypervisor [25] uses the same split-driver model as Xen. Therefore our work can easily be applied to it. In KVM [26] (Kernel-based Virtual Machine), a well identified Linux kernel module handles all VMs I/O operations. This module is similar to the backend in Xen. Thus, it can host the implementation of our solution. OpenVZ [27] design is similar to KVM. In Xen HVM mode, each VM is linked with a Qemu device emulator, lying in the dom0. This brings us back to the KVM design. About VMware ESXi [28], a device emulator which resides inside the hypervisor handles all VMs I/O operations. Our solution can be implemented in that place.

VI. Related work

The main motivation in this paper is performance unpredictability and resource waste in the cloud. Several works have investigated I/O virtualization issue [20], [19], [11], [16], [18]. Nonetheless, several studies [7], [15] have highlighted performance unpredictability in the cloud because of the VMs competition on shared resources. Works in this field can be classified in two categories. The first category consists of studies that proposed solutions at a micro architectural level (e.g. cache contention effects). This approach consists in placing VMs intelligently on machines in order to avoid compete workloads atop the same machine [12]. Studies [10], [9], [14], [5] of the second category have addressed the problem at the software level. In this category, they advocate for bandwidth allocation to VMs. Each VM is allocated a minimum bandwidth, which is guaranteed. [5] goes in the same vein by limiting a VM bandwidth proportionally to its booked CPU capacity. As we said in our previous work [24], all these studies do not accurately guarantee performance predictability, even less CPU time charged on clients VMs as we do in this paper. Concerning the second category, the approach could be efficient if a given bandwidth always leads to the same CPU time in the DD. As we have shown, this is not true since several factors intervene. For instance, Fig. 10 presents the DD CPU load when a VM uses its entire network bandwidth (48MB/s) for sending/receiving packets of different sizes. We can see a significant difference, up to twice the load for smaller packets. Our solution tackles all these issues at a scheduler level.

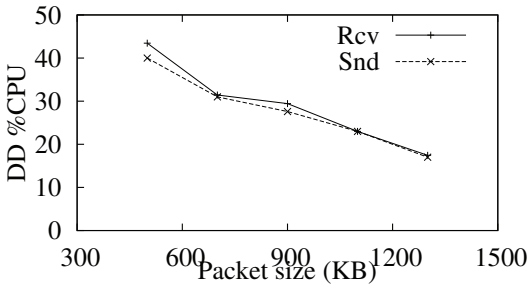

Figure 10: Bandwidth selling

[21] is the only study close to what we propose. They propose both ShareGuard (a bandwidth delimiter system) and SEDF-DC (a scheduler which takes into account CPU time used by the DD on behalf of VMs) to improve performance predictability. As we said in our previous work [24], [21] has the following weaknesses. (1)Their study focuses on network devices, therefore ignores disk operations. (2) The proposed SEDF-DC scheduler can only be applied to mono processor machine. (3) ShareGuard is intrusive because, its drops network packets for VMs whose CPU load within the DD is above the configured threshold. (4) SEDF-DC and ShareGuard use XenMon which requires to be constantly activated, thus consuming a non negligible CPU time.

VII. Conclusion

This paper has proposed a complementary solution to two relevant problems in the cloud: performance unpredictability and resource waste. We have addressed them using a new light solution. Roughly, instead of investigating micro-architecture components (for performance unpredictability) or proposing yet-another consolidation algorithm, we have proposed an orthogonal solution based on an efficient charging of CPU time used by the system components to VM. The article describes our solution, including a prototype. The latter has been evaluated with various workloads (including real data center traces). It has demonstrated its ability to accurately overcome the initial problems without an overhead.

Acknowledgements

This work benefited from the support of the French "Fonds national pour la Société Numérique" (FSN) through the OpenCloudware project.

REFERENCES

[1] Daniel Hagimont, Christine Mayap, Laurent Broto, Alain Tchana, and Noel De Palma, "DVFS aware CPU credit enforcement in a virtualized," Middleware 2013.

[2] Credit Scheduler, 'http://wiki.xen.org/wiki/Credit_Scheduler.

[3] J. Renato Santos, G. (John) Janakiraman, and Yoshio Turner, 'Xen Network I/O Performance Analysis and Opportunities for Improvement," Xen summit 2007.

[4] Qun Huang and Patrick P.C. Lee, 'An Experimental Study of Cascading Performance Interference in a Virtualized Environment," SIGMETRICS Perform. Eval. Rev. 2013.

[5] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea, 'Chatty Tenants and the Cloud Network Sharing Problem," NSDI 2013.

[6] Procps: http://procps.sourceforge.net/

[7] Has Amazon EC2 become over subscribed?http://alan.blogcity.com/has _amazon_ec2_become_over_subscribed.htm.

[8] Data Set for IMC 2010 Data Center Measurement. http://pages.cs.wisc.edu/ tbenson/IMC10_Data.html

[9] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron, 'Towards Predictable Datacenter Networks", SIGCOMM 2011.

[10] Jorg Schad, Jens Dittrich, and Jorge-Arnulfo Quiane-Ruiz, 'Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance", VLDB 2010.

[11] Yaozu Dong, Zhao Yu, and Greg Rose, 'SR-IOV networking in Xen: architecture, design and implementation", Usenix WIOV 2008.

[12] George Kousiouris, Tommaso Cucinotta, and Theodora Varvarigou, "The effects of scheduling, workload type and consolidation scenarios on virtual machine performance and their prediction through optimized artificial neural networks," Journal of Systems and Software 84(8) 2011.

[13] Joydeep Mukherjee, Diwakar Krishnamurthy, Jerry Rolia, and Chris Hyser, "Resource contention detection and management for consolidated workloads," IM 2013.

[14] Lucian Popa, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica, 'FairCloud: Sharing the Network in Cloud Computing", HotNets-X 2011.

[15] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao, 'Who Is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds", TRANSACTIONS ON SERVICES COMPUTING 2013.

[16] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee, 'Task-aware virtual machine scheduling for I/O performance", VEE 2009.

[17] y-cruncher: A Multi-Threaded Pi-Program. http://www.numberworld.org/y-cruncher/.

[18] Denghui Liu, Jinli Cao, and Jie Cao, 'FEAS: a full-time event aware scheduler for improving responsiveness of virtual machines", ACSC 2012.

[19] Dimitris Aragiorgis, Anastassios Nanos, and Nectarios Koziris, 'Coexisting scheduling policies boosting I/O Virtual Machines", Euro-Par 2011.

[20] Xiao Ling, Hai Jin, Shadi Ibrahim, Wenzhi Cao, and Song Wu, 'Efficient Disk I/O Scheduling with QoS Guarantee for Xen-based Hosting Platforms", CCGRID 2012.

[21] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat, "Enforcing performance isolation across virtual machines in Xen', Middleware 2006.

[22] Diwaker Gupta and Ludmila Cherkasova, "XenMon: QoS Monitoring and Performance Profiling Tool', HPL-2005-187 2005.

[23] Amit Kumar, Rajeev Rastogi, Avi Silberschatz, and Bulent Yenerc, "Algorithms for provisioning virtual private networks in the hose model', SIGCOMM 2001.

[24] Boris Teabe, Alain Tchana, and Daniel Hagimont, "Billing System CPU Time on Individual VM", CCGRID 2015 (short paper/poster)

[25] Hyper-V Architecture: http://msdn.microsoft.com/en-us/library/cc768520.aspx.

[26] KVM: http://www.linux-kvm.org.

[27] OpenVZ: http://openvz.org/Main_Page.

[28] VMware ESXi: www.vmware.com/products/vsphere/esxi-and-esx/index.html

[29] https://www.spec.org/virt sc2010/, visited on January 2015