Mitigating performance unpredictability in the IaaS using the Kyoto principle

Alain Tchana, Vo Quoc Bao Bui, Vlad-Tiberiu Nitu, Boris Teabe, Daniel Hagimont

Mitigating performance unpredictability in the IaaS using the Kyoto principle

Alain Tchana
University of Toulouse, France
alain.tchana@enseeiht.fr

Bao Bui
University of Toulouse, France
bao.bui@enseeiht.fr

Boris Teabe
University of Toulouse, France
boris.teabedjomgwe@enseeiht.fr

Vlad Nitu
University of Toulouse, France
vlad.nitu@enseeiht.fr

Daniel Hagimont
University of Toulouse, France
daniel.hagimont@enseeiht.fr

ABSTRACT

Performance isolation is enforced in the cloud by setting to each virtual machine (VM) a given fraction of each resource type (physical memory, processor, and IO bandwidth). However, microarchitectural-level resources such as processor's caches cannot be divided and allocated to VMs: they are globally shared among all VMs which compete for their use, leading to cache contention. Therefore, performance isolation and predictability are compromised. This situation is devastating for HPC applications. In this paper, we propose a software solution (called Kyoto) to this issue, inspired by the polluters pay principle. A VM is said to pollute the cache if it provokes significant cache replacements which impact the performance of other VMs. Henceforth, using the Kyoto system, the provider can encourage HPC cloud users to book pollution permits for their VMs. We have implemented Kyoto in several virtualization systems including both general purpose systems (Xen and KVM) and specialized HPC systems (Pisces).

1. INTRODUCTION

Nowadays, many organizations tend to outsource the management of their physical infrastructure to hosting centers. By this way, companies aim at reducing their cost by paying only for what they really need. This trend, commonly called cloud computing, is general and concerns all field of Information Technology. Notably, recent years have seen HPC application developers and industries thinking about the migration of their applications to the cloud [1].

In this context, the majority of platforms implements the Infrastructure as a Service (IaaS) cloud model where customers buy virtual machines (VM) with a set of reserved resources. The main benefit of virtualization is that it provides isolation among VMs running on the same physical machine. Isolation takes different forms, including security (sandboxing) and performance. Regarding security, isolation between VMs means that operating systems (and their applications) running in VMs are executing in separate address spaces and are therefore protected against illegal (bogus or malicious) accesses from other VMs. Regarding performance, isolation means that the performance of applications in one VM should not be influenced or depend on the behavior of other VMs running on the same physical machine.

This paper addresses an issue related to performance isolation. Performance isolation is enforced by giving to each VM a fraction of each resource type (physical memory, processor, and IO bandwidth). However, microarchitectural-level resources such as processors' caches cannot be divided. They are globally shared between all VMs. This situation can lead to cache contention. Therefore, performance isolation and predictability are compromised, as pointed out by several research [3, 8, 2]. Many microarchitectural-level resources may compromise performance isolation (QPI bus, Lx caches), but Last Level Cache (LLC) contention has been identified as one of the most critical component.

Several research have investigated the LLC contention issue. They can be organized into two categories. The first category includes research [7, 26, 28, 33, 31, 32] which proposes to intelligently collocating processes or VMs. Concerning the second category, it includes research [17, 19, 27] which proposes to physically or softly partition the cache. The main drawbacks of these solutions are the following: cache partitioning solutions require the modification of hardware (not yet adopted in today's clouds) while VM placement solutions are not always optimal (VM placement is a NP-hard problem). **Most important, these solutions are not in the spirit of the cloud** which relies on the payper-use model: why not each VM is assigned an amount of cache utilization in the same way as it is done for coarse-grained resource types?

In this paper, we propose Kyoto, a software solution to the issue of LLC contention. This solution is inspired by the polluters pay principle. A VM is said to pollute a cache if it provokes significant cache replacements which impact the performance of other VMs. We rely on hardware counters to monitor the cache activity of each VM and to measure each VM cache pollution level. Henceforth, using the Kyoto system, the provider may compel cloud users to book pollution permits for their VMs. Therefore, a VM which exceeds its permitted pollution at runtime has its CPU capacity reduced accordingly. We have implemented Kyoto in several virtualization systems including both general purpose systems (Xen and KVM) and specialized HPC systems

DOI: http://dx.doi.org/10.1145/2988336.2988342

(Pisces [4]). In summary, the main contributions we make in this paper are:

- We introduce a new VM configuration parameter which allows to book an amount of LLC pollution (a pollution permit).
- We implement Kyoto in three popular virtualization systems. Notice that our approach can easily be implemented within other systems.
- We perform several experimentation campaigns using micro and macro benchmarks (provided by SPEC CPU2006 [11]) in order to validate our approach. The results of these experiments validate Kyoto's effectiveness in terms of performance isolation and predictability. They also show that Kyoto introduces a negligible overhead.

The rest of the article is organized as follows. Section 2 presents both the problematic and the motivations of our work, including an experimental assessment of the addressed issue. Contributions are detailed in Section 3 while Section 4 presents evaluation results. Section 5 presents the application scoop. After a review of related works in Section 6, we conclude the paper in Section 7.

2. MOTIVATIONS

2.1 Problem statement

Virtualization has proved to be one of the best technology to isolate the execution of distinct applications in the same computer. The main feature which allows achieving this goal is resource partitioning. The analysis of today's hypervisors shows that only the partitioning of coarse-grained hardware resources (the main memory, the CPU, etc.) are allowed. The partitioning of microarchitectural-level components such as the Front Side Bus (FSB) and processors' caches are not taken into account, resulting in contention. This situation suits for some application types like network intensive applications. However, it is problematic for a non negligible proportion of application types. Several research have shown that contention on microarchitectural-level components is one of the main source of performance unpredictability [31]. The consequences of the latter are twofold. On the one hand, it could require supplementary tasks from cloud users. For instance, Netflix developers have reported [8] that they needed to redesign their applications to deal with this issue in Amazon EC2. On the other hand, some suggest that performance unpredictability contributes to brake the inroad of the cloud in some domains like HPC [1].

Contention on the LLC has been pointed by several research [33, 31, 32] as a critical issue. Therefore, this paper focuses on the problem of LLC contention.

Definition: LLC contention occurs when several VMs compete on the same LLC lines. It concerns both VMs which run in parallel (on distinct cores) or in an alternative manner on the same core. The former situation is promoted by the increase number of cores in today's machines while the latter situation comes from time sharing scheduling. The next section presents evaluation results which attest the need to handle LLC contention.

Main memory	8096 MB
L1 cache	L1_D 32 KB, L1_I 32 KB, 8-way
L2 cache	L2_U 256 KB, 8-way
LLC	10 MB, 20-way
Processor	1 Socket, 4 Cores/socket

Table 1: Experimental machine

2.2 Problem assessment

In order to provide a clear illustration of the issue we address, we consider the following assumptions: any VM runs a single application type and is configured with a single vCPU which is pinned to a single core.

2.2.1 Experimental environment

All experiments have been performed on a Dell machine with Intel Xeon E5-1603 v3 2.8 GHz processor. Its characteristics are presented in Table. 1. The machine runs a Ubuntu Server 12.04 virtualized with xen 4.2.0.

2.2.2 Benchmarks

Micro benchmark.

Micro benchmark applications come from [15]. In brief, a micro benchmark application creates an array of elements whose size corresponds to a specific working set size. Elements are randomly chained into a circular linked list. The program walks through the list by following the link between elements.

Macro benchmark.

We use both *blockie* [20] and applications from SPEC CPU2006 [11] as complex benchmarks. They are widely used to assess the processor and the memory subsystem performance.

2.2.3 Metrics

The two following metrics are used: cache miss ratio (cache misses per millisecond) and instruction per cycles (IPC). The latter is used to measure an application performance. To compute these metrics, we gathered statistical data from hardware performance monitoring counters (PMC) using a modified version of perfctr-xen [18].

2.2.4 Evaluation scenarios

Handling an intermediate level-cache (ILC) miss takes less time (the probability to find the missed data within the other cache is high) than handling an LLC miss (which always requires main memory accesses). In the case of our experimental machine, the time taken to access each cache level (measured with lmbench [14]) is the following (approximately): 4 cycles for L1, 12 cycles for L2, 45 cycles for LLC, and 180 cycles for the main memory. Therefore, VMs can be classified into three categories: C_1 includes VMs whose working set fits within ILC (including L1 and L2), C_2 includes VMs whose working set fits within the LLC (L3), and C_3 is composed of the other applications. For each category C_i ($1 \leq i \leq 3$), we have developed both a representative and a disruptive VM, respectively noted v^i_{rep} and v^i_{dis}. Each v^i_{rep} is executed in ten situations: alone (one situation), in an alternate manner with each v^i_{dis} (three situations), in parallel with each v^i_{dis} (three situations), and both in parallel and in an alternate way with each v^i_{dis} (three situations).

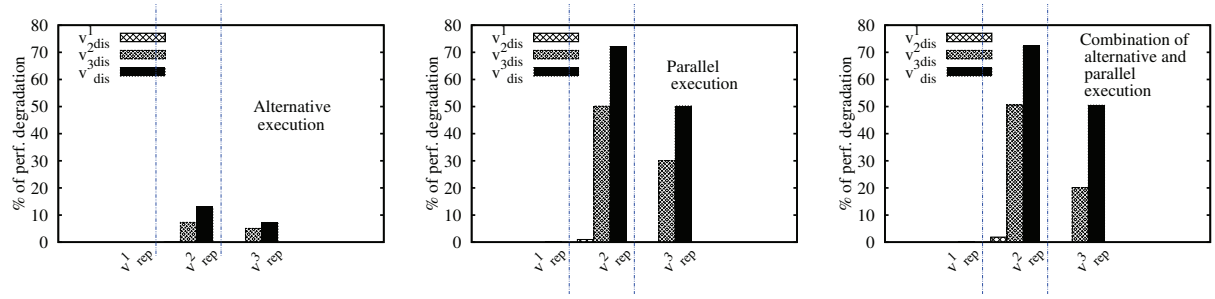

Figure 1: LLC contention could impact some applications.

2.2.5 *Evaluation results*

Fig. 1 presents the execution results of the above scenarios. Firstly, we can see that the competition on ILC is not critical for any VM type (all the first bars are invisible because the performance degradation percentage is almost nil). In addition, C_1's VMs are agnostics to both ILC and LLC contention (the three first bars of each curve is invisible because the performance degradation of v^i_{rep} is almost nil). Indeed, the cost needed to handle an ILC miss is negligible. Secondly, we can see that both C_2 and C_3's VMs are severely affected by LLC contention (the four visible bars in each curve show that the performance degradation percentage is not negligible). Thirdly, contention generated by a parallel execution is more devastating than the contention generated by an alternative execution: up to 70% of performance degradation in the former vs about 13% in the latter. In order to complete the analysis, let us zoom-in on the first six v^2_{rep}'s time slices[1] (v^2_{rep} is the most penalized VM type). We can see from Fig. 2 that when the VM runs alone, LLC misses occur only during the first time slice (data loading). It is not the case in the other situations because of the competition on LLC lines. This problem is well observed in the alternative execution which has a zigzag shape: the first tick of each time slice is used for loading data to the LLC (because the data have been evicted by the disruptive VM during the previous time slice). Concerning the parallel execution, the cache miss rate is very high because of data eviction. This is caused by the parallel execution with the disruptive VM.

In conclusion, sharing the LLC without any partitioning strategy under its utilization could be problematic for some VMs. In this paper we propose a solution in this direction, see the next section. In the rest of the article, C_2 and C_3's VMs are called sensitive VMs.

3. CONTRIBUTIONS

This section presents our solution (called Kyoto) to the LLC contention issue. After a presentation of the basic idea behind Kyoto (simple but powerful), a detailed description of its implementation within the Xen virtualization system is given (the patch can be downloaded at https://bitbucket.org/quocbaoit/xen-4.2.0-perfctr.git). We have also implemented Kyoto within KVM (the default Linux virtualization system) and Pisces [4] (a lightweight co-kernel for achieving performance isolation for HPC applications). An evaluation

[1]A time slice (30msec) is composed of 3 ticks (10msec) in Xen.

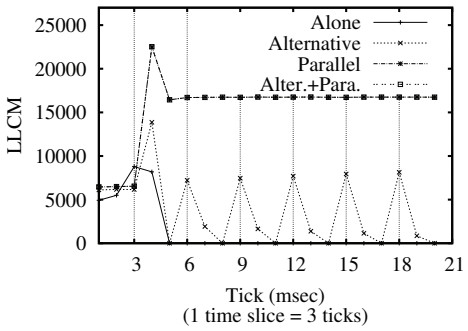

Figure 2: Impact of LLC contention explained with LLC misses

of the latter is presented in Section 4. In order to respect the page length, we only present the Xen implementation.

3.1 Basic idea: "polluters pay"

We propose a software solution whose basic idea is the same as the "polluters pay" principle of the Kyoto protocol [9]. This solution relies on the following assumptions. (1) A VM execution time results in the pollution of the LLC at a certain level. (2) Therefore, a VM which generates a high pollution level is likely to cause more contention (thus aggressive against other VMs) when it is collocated with other VMs. Under these assumptions, if one is able to instantiate a VM with a booked pollution level, and to enforce that pollution level during the overall VM lifetime, then he will have defined a solution to the problem of cache partitioning, thus cache contention. Therefore, the utilization of the LLC could be charged to cloud users in the same way as coarse-grained resources (e.g. processor, disk, main memory). This is the main idea we follow in this paper. This idea raises two main challenges:

- How to monitor a specific VM pollution level at runtime?
- How to enforce a booked pollution level at runtime?

The first challenge can be achieved using hardware performance monitoring counters (PMCs). The latter allow to gather information about the utilization of the majority of microarchitectural-level components such as the LLC. Section 3.3 presents which metrics Kyoto uses to compute a VM pollution level. Concerning the second challenge, Kyoto relies on the processor, which is the central resource in

a computer: a VM is only able to pollute the LLC when it is scheduled on a processor. Therefore, the processor can service as a lever to enforce a pollution level (this is illustrated in Section 4.1). A VM whose actual pollution level exceeds the booked one sees its computing capacity reduce. Therefore, handling the second challenge requires the extension of the hypervisor component which is responsible to schedule VM on processors. The next section presents an implementation of the Kyoto's scheduler within Xen.

3.2 The Kyoto's scheduler within Xen

The Kyoto's scheduler (hereafter noted KS4Xen) enforces each VM's booked pollution level during the overall lifetime of the VM. Before presenting KS4Xen, we firstly gives a quick description of the Xen credit scheduler (hereafter noted XCS), knowing that further details could be found in [16].

XCS.

It is the default scheduler in Xen. It is suitable for cloud platforms since the customer books for an amount of computing capacity which should be ensured without wasting resources. XCS works as follows. A VM v is configured at start time with a credit c which should be ensured by the scheduler. To this end, the latter defines *remainCredit*, a scheduling variable, which is initialized with c. Each time a v's vCPU is scheduled on a processor, (1) the scheduler translates into a credit value (let us say *burntCredit*) the time spent by v on that processor. (2) Subsequently, the scheduler computes a new value for *remainCredit* by subtracting *burntCredit* from *remainCredit*. When the latter reaches a lower threshold, the VM is no longer allowed to get the processor. We can say that the VM is "blocked". Periodically, the scheduler increases the value of *remainCredit* for each VM blocked VM according to its initial credit c. This allows the VM to become schedulable.

KS4Xen.

We propose KS4Xen as an extension of XCS. The former works as follows. In addition to c (introduced above), a VM is configured (booked by its owner) with the pollution level (noted *llc_cap*) it intends to generate during a time slice. At runtime, a scheduling variable named *pollution_quota* is assigned to each VM. As well as XCS ensures the respect of c, KS4Xen does the same for *llc_cap*. This is achieved by periodically monitoring LLC related statistics for each VM. From these collected data, the actual *llc_cap* (noted llc_cap_{act}) of each VM is computed (see Section 3.3). The scheduler then debits the VM's *pollution_quota* according to this llc_cap_{act}. If a VM's *pollution_quota* goes negative, that VM will be in priority OVER, meaning that it cannot use the processor any more. At the end of each time slice, VMs earn a specific amount of pollution quota based on their booked *llc_cap*. If a *pollution_quota* is positive, the VM is marked UNDER, meaning that it can use the processor. There are also some codes we have introduced in order to provide a way to set a VM's *llc_cap* as a Xen command line parameter. In summary, apart from the code provided by perfctr-xen [18], which is used to collect PMCs, we made our modifications in 8 files of Xen source codes, representing about 110 LOCs.

3.3 Computation of llc_cap_{act}

The computation of llc_cap_{act} is periodically performed (e.g. each 100 million of instructions) for all active vCPUs. We assume that vCPUs of the same VM have the same behaviour. Therefore, only one vCPU of each VM is considered. Kyoto relies on two performance metrics: LLC Misses and UnHalted Core Cycles. Subsequently, the llc_cap_{act} is estimated using equation 1.

$$llc\_cap_{act} = \frac{llc\_misses \times cpu\_freq\_khz}{unhalted\_core\_cycles} \quad (1)$$

Being able to collect LLC related statistics is not sufficient to compute llc_cap_{act} for each specific VM. A crucial question goes unresolved: How to rightly identify PMCs of a specific VM knowing that several VMs may run in parallel atop the same LLC[2]? The Kyoto monitoring system is able to use two solutions. The first solution consists in dedicating the use of the LLC to the vCPU whose llc_cap_{act} needs to be computed. In other words, only one core in the socket is activated during the sampling time (about one billion of cycles). The other vCPUs are migrated to another socket. This solution could impact migrated vCPU performance (as shown in Section 4.5). The second solution comes as a response to this limitation.

The second solution relies on the use of a microarchitectural-level simulator. We have used the McSimA+ [12] simulator in our prototype. McSimA+ [12] is able to be configured to reflect a specific hardware (including processor caches, pipelines, etc.). Using a pin tool [13], the instructions generated during the execution of an application can be concurrently replayed within the simulator. McSimA+ returns PMCs related to the architecture of the machine given as the input. Relying on such a simulator, which runs atop a dedicated machine, the computation of each VM's llc_cap_{act} can be achieved following these steps:

1. KS4Xen asks the simulator to start the pin tool for a sampling period,
2. the simulator replays instructions and sends PMCs back to KS4Xen,
3. and KS4Xen computes the llc_cap_{act} based on the collected PMCs.

The next section presents the evaluation results of all KS4Xen aspects.

4. EVALUATIONS

After the presentation of the results which justify our choices, the evaluation of both KS4Xen's effectiveness and overhead are presented. Unless otherwise specified, any VM uses a single vCPU (having the computing capacity of a core) and runs either a SPEC CPU2006 application or *blockie*. The latter is one of the most contentious application from the contention benchmark suite developed in [20]. To make reading easier, we use the following notations: v^i_{sen} and v^i_{dis} respectively identify a sensitive and a disruptive VM, ${}^{lc}v$ means that the VM v is configured with a booked *llc_cap* value equals to lc. Table 2 shows the name of the application which corresponds to each v^i_{sen} and v^i_{dis} $(1 \leq i \leq 3)$. Throughout the rest of the article, the expression "we ran an application x" is equivalent to "we ran a VM hosting application x".

[2]A VM should not be punished for the pollution of another VM.

VM name	Applications
v_{sen^1}, v_{sen^2}, v_{sen^3}	respectively gcc, omnetpp, soplex
v_{dis^1}, v_{dis^2}, v_{dis^3}	respectively lbm, blockie, mcf

Table 2: Experimental VMs

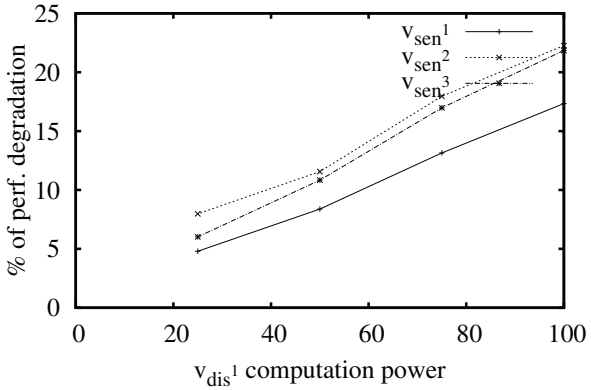

Figure 3: The processor is a good lever for punishing polluter/disruptive VMs

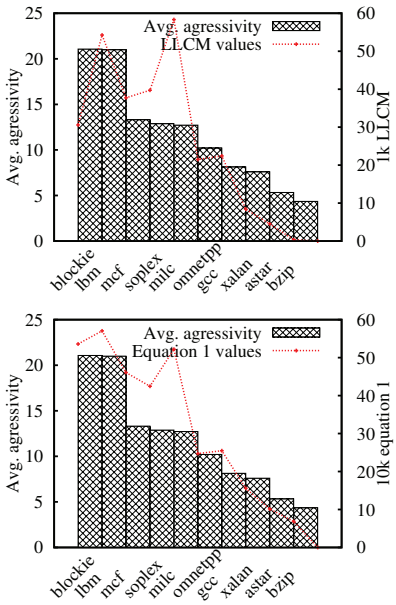

Figure 4: Equation 1 vs LLCM

4.1 The processor is a good lever

KS4Xen uses the processor as the lever to enforce an assigned *llc_cap*. The first experiment type confirms a strong relationship between a VM's computing capacity and its aggressiveness, which is correlated to its pollution level. The scenario we use for these experiments is the following. We run each v_{sen^i} in parallel with a v^i_{dis} (let us say v_{dis^1} (lbm)) while varying the computing capacity of the latter. Fig. 3 shows the results of these experiments. We can see that each v_{sen^i}'s performance degradation percentage linearly increases with v_{dis^1}'s. Indeed, increasing v_{dis^1}'s computing capacity increases its scheduling frequency, which in turn increases its aggressiveness.

4.2 Equation 1 vs LLC misses (LLCM): which indicator as the *llc_cap*?

This section presents evaluation results which confirm the better accuracy of equation 1 (introduced in [7]) in comparison with LLCM for the estimation of each VM pollution level. The latter can be seen as the aggressiveness level of the VM. We use the following scenario. We evaluate the aggressiveness of 10 applications (astar, blockie, bzip, gcc, lbm, mcf, milc, omnetpp, soplex, and xalan) as follows. Each application is firstly executed alone and its *llc_cap* is computed in two manners: using LLCM and using equation 1. Subsequently, each application is executed in parallel with each of the other applications to evaluate its real aggressiveness. The latter corresponds to the performance degradation level the causes. The average aggressiveness of each application is computed. The results of these experiments are presented in Fig. 4 in a descending order regarding real aggressiveness values. The latter lead to the order o_1=(blockie, lbm, mcf, soplex, milc, omnetpp, gcc, xalan, astar, bzip) while the order obtained with LLCM is o_2=(milc, lbm, soplex, mcf, blockie, gcc, omnetpp, xalan, astar, bzip) and the one obtained with equation 1 is o_3=(lbm, blockie, milc, mcf, soplex, gcc, omnetpp, xalan, astar, bzip). Relying on the Kendall's tau [36] method, we can see that o_3 is more closer to o_1 than o_2. In conclusion, equation 1 is a better indicator for *llc_cap* than LLCM.

4.3 KS4Xen's effectiveness

This section evaluates the benefits of KS4Xen in terms of LLC contention limitation. This can be judged by the ability of KS4Xen to ensure performance predictability. This evaluation is straightforward. We run in parallel ${}^{250k}v_{sen^1}$ (gcc) with different ${}^{250k}v_{dis^i}$ (lbm, blockie, and mcf). Fig. 5 shows the results of these experiments. We can see that the performance of v_{sen^1} is almost kept whatever the aggressiveness of the concurrent VM (Fig. 5 top left). Fig. 5 top right shows respectively the number of times where v_{sen^1} and v_{dis^i} have been punished. All v_{dis^i} (disturber VMs) have received more penalties than v_{sen^1}. To complete the analysis, curves in Fig. 5 bottom plot for v_{dis^1} (lbm) respectively the variation of both measured *llc_cap* and the processor utilization. Contrary to XCS (the red line), we can see that in KS4Xen, the VM is deprived of the processor for long moment every time the measured *llc_cap* exceeds the booked *llc_cap* (the zigzag line).

We have also evaluated KS4Xen scalability. To this end, we execute ${}^{250k}v_{sen^1}$ while varying the number of colocated ${}^{50k}v_{dis^i}$ (from 1 to 15 vCPUs[3]). KS4Xen is scalable if v_{sen^1}'s performance is kept. From Fig. 6, we can see that KS4Xen always keeps the performance of the sensitive VM whatever the number of colocated disturbers.

4.4 Comparison with existing systems

The previous section have presented the Kyoto's effectiveness in comparison with the Xen system, a general purpose

[3]According to [10], the average number of vCPUs sharing the same core is about 4. Having 4 cores in our socket, we can colocate up to 16 vCPUs (remember that v_{sen^1} is already assigned one vCPU).

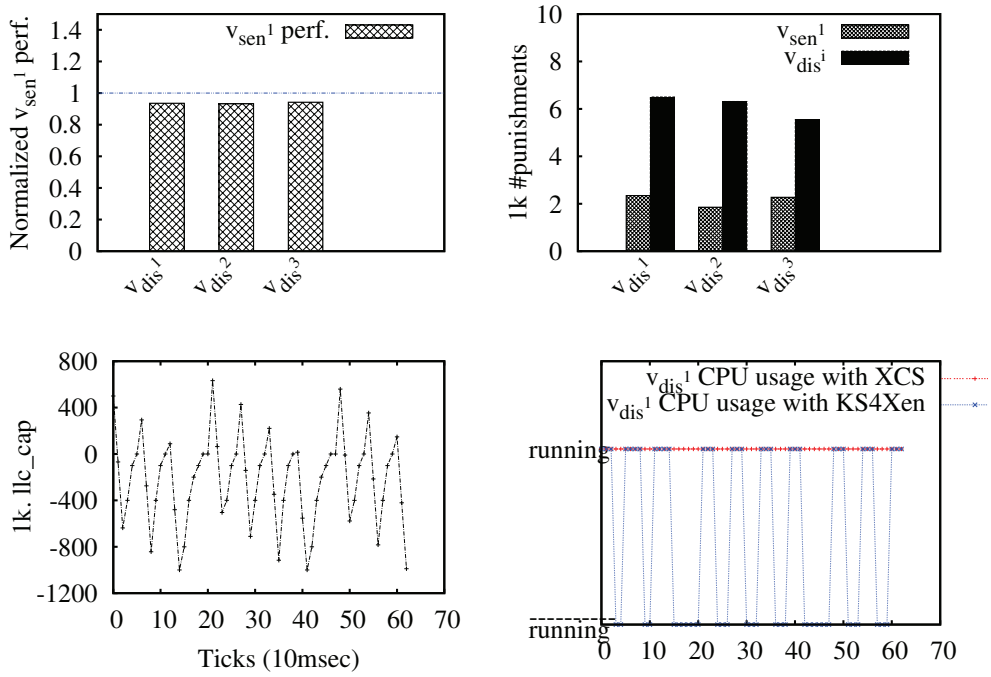

Figure 5: KS4Xen minimizes LLC contention, thus avoids performance variations

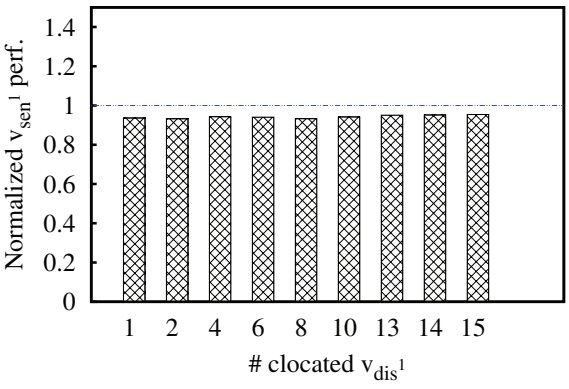

Figure 6: KS4Xen's scalability

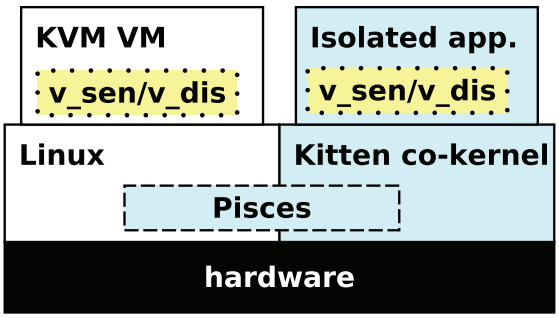

Figure 7: Pisces architecture

virtualization system. We have also compared Kyoto with Pisces [4], a co-kernel [5] which allows building strongly isolated HPC applications (see Fig. 7). To guarantee performance isolation, a Pisces application runs in a VM which has the entire control of its assigned resources, without the intervention of an hypervisor. By doing so, Pisces avoids the contention within the hypervisor and other virtualization components (such as driver domains), which is known to be source of performance interference [6]. We have evaluated the Pisces capability to (1) isolate a sensitive application (v_{sen^1}) and to limit the negative effect of a disruptive application (v_{dis^1}). Subsequently, we have implemented and evaluated the effectiveness of two other Kyoto versions: one for the Linux virtualization system (via the CFS scheduler, noted KS4Linux) and the other for Pisces (noted KS4Pisces). Fig. 8 (the first two bars) shows that

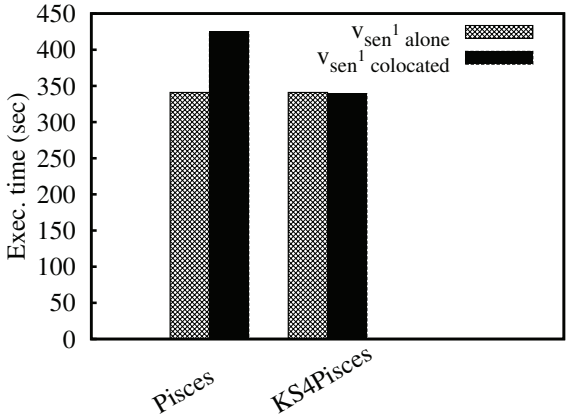

Figure 8: Comparison of Kyoto with Pisces

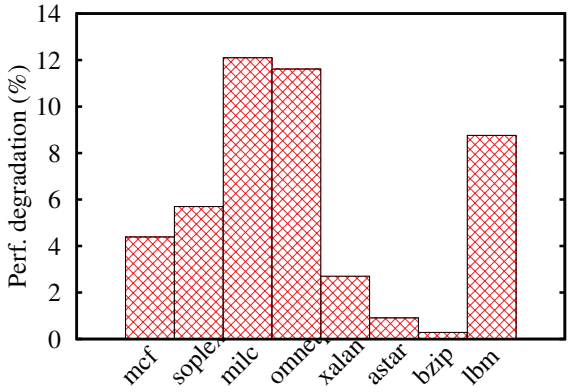

Figure 9: Migrating vCPU could impact VMs which host memory bound applications

Pisces does not ensure performance predictability when the LLC is shared between a sensitive and a disruptive VM (the performance difference is about 24%). This is explained by the fact that the performance interference issue considered by Pisces is the one which comes from shared virtualization components (such as the driver domain) and coarse-grained resources (such as processors). Microarchitectural-level components like the LLC are not considered. Fig. 8 (the last two bars) also shows that when the previous experiment is played in a Kyoto environment, performance predictability is achieved (notice that we use the same *llc_cap* value presented in the previous section).

4.5 Kyoto's overhead

The complexity of Kyoto is $\mathcal{O}(n)$, where n is the number of VMs (about a hundred) in the physical machine. This section evaluates Kyoto's overhead by relying on KS4Xen knowing the lessons learned here are applicable to other Kyoto's implementations. The execution of KS4Xen can introduce two overhead types: (1) from the solution used to identify LLC statistics related to a specific vCPU (to compute its llc_cap_{act}, see Section 3.3), and (2) from the monitoring system (PMCs gathering). This section evaluates the impact (if ever exists) of these overheads.

llc_cap_{act} computation.

Recall that one of the solutions used by KS4Xen to identify the LLC statistics related to a specific vCPU relies on the dedication of a socket to that vCPU for the duration of the sampling. This requires the migration of not concerned vCPUs to another socket. We evaluate the impact of this migration using the following scenario. We experiment 8 SPEC CPU2006 applications atop a NUMA machine (PowerEdge R420) composed of 2 sockets (noted $numa_0$ and $numa_1$). Each experiment uses a single VM composed of a single vCPU which starts its execution on $numa_0$. KS4Xen is configured to periodically migrate the vCPU between $numa_0$ and $numa_1$. The return migration from $numa_1$ to $numa_0$ is performed after a random period in order to mimic the time taken by KS4Xen to compute all vCPUs' llc_cap_{act}. Fig. 9 presents the results of these experiments. We can see that all VMs are not impacted at the same level. We have observed that the most affected applications (milc, omnetpp, lbm) are those which run memory intensive applications (up to 12% overhead). This is explained by the fact that when the vCPU is migrated to $numa_1$, all memory accesses are done remotely.

This degradation can be minimized by reducing the number of migrations. We have identified two situations in which vCPU isolation is not mandatory. These situations are:

- A vCPU which generates a very low level of LLC misses (let us say lower than a configurable threshold) will not be isolated. Indeed, such vCPUs are neither disturbers nor sensitive. The first two bars in Fig. 10 shows the value of llc_cap_{act} for a VM running hmmer (known to generate low LLC misses) when its vCPU is isolated and not isolated (colocated with several disturbers vCPUs). We can see that the difference is almost nil.
- A vCPU which shares the LLC only with vCPUs which generate low level LLC misses will not be isolated. Indeed, since colocated vCPUs are not disturbers, it is most likely that the obtained llc_cap_{act} is not far from the correct value. The last two bars in Fig. 10 shows bzip's *llc_cap* is almost the same when it is colocated with several hmmer applications.

PMCs gathering.

We have also evaluated KS4Xen's overhead in terms of the amount of resources it consumes. Concerning the main memory, KS4Xen extends two data structures (*structcsched_vcpu* and *structcsched_dom*) to record PMCs for each VM. This extension is about 72 bytes, which is negligible. Concerning the processor, the execution of perfctr-xen (for gathering PMCs) is the only source of processing time consumption. To evaluate the latter, we ran in parallel two VMs which host the same CPU bound application (the SPEC CPU2006 application povray) atop the same processor. KS4Xen and XCS are experimented with different time slices (scheduling periods) to vary the intervention delay (thus the execution of the monitoring system, the potential source of overhead). Fig. 12 presents the results of these experiments. We can see that both KS4Xen and XCS lead VMs to the same performance level. In other words, the monitoring system used by KS4Xen does not introduce an overhead.

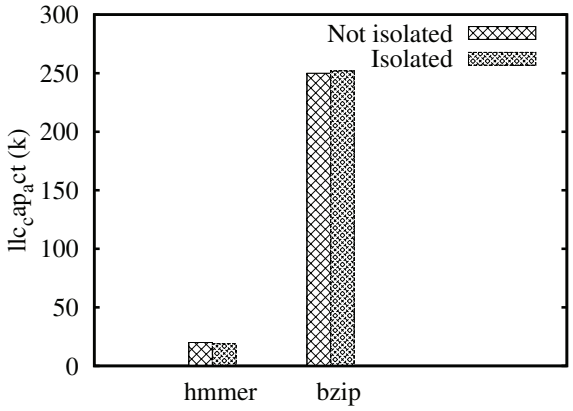

Figure 10: vCPU isolation could be avoided in some situations

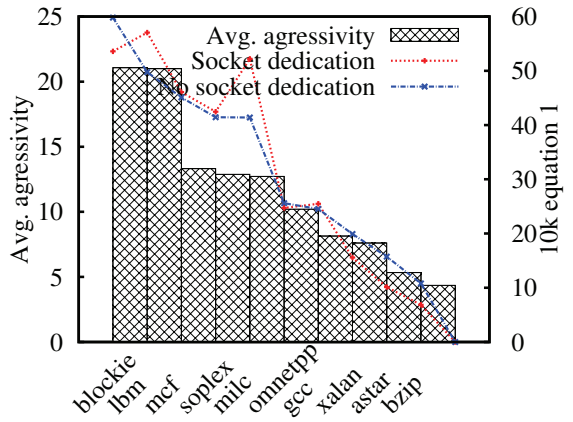

Figure 11: Socket dedication could be avoided when computing llc_cap_{act}

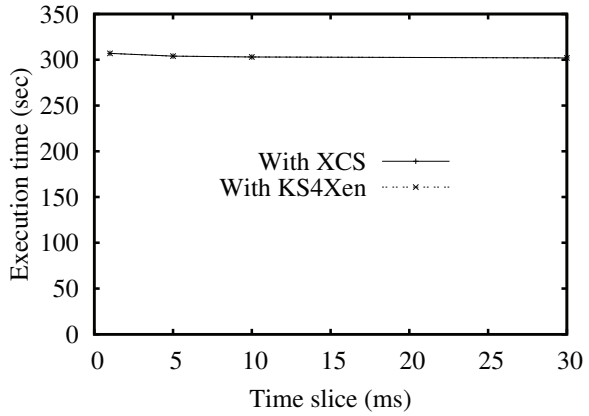

Figure 12: The overhead incurs by KS4Xen is near zero.

5. DISCUSSION

The contribution of this paper does not target all cloud types. It is suitable for HPC clouds since they run applications which are very sensitive to microarchitectural-level components behavior (such as LLC contention). Therefore, we assume that users of such clouds are able to deal with the new parameter we have introduced: the *llc_cap*. A question that one could ask is how the user chooses a VM's *llc_cap* value? We answer this question as follows. A cloud platform often defines a set of bookable instance types (e.g. Amazon EC2 proposes 38 instance types[4]) which are different each other by the amount of resource they are assigned regarding each resource type. For instance in Amazon EC2, the particularity of a R3 instance is the fact that it is assigned a lot of memory in comparison with the computing capacity. Therefore, relying on VM typed, the provider can associate to each instance type a *llc_cap* level. We can assume that the latter is proportional to the amount of memory assigned to the instance. For instance, R3's instances will be assigned much more *llc_cap* than C3's instances since the primary needs of the latter is the computing capacity.

6. RELATED WORK

Existing solutions can be organized into two categories: placement algorithms and cache partitioning.

Placement algorithms.

Several prior work have proposed cache aware scheduling algorithms to address the problem of LLC contention. In the context of non-virtualized environments, [7, 26, 28, 38, 39, 40] presented some methods to evaluate the sensitivity and the aggressiveness of an application. Our Kyoto system uses one of these approaches, particularly the one presented by [7]. [21] proposed ATOM (Adaptive Thread-to-Core Mapper), a heuristic to find the optimal mapping between a set of processes and cores such that the effect of cache contention is minimized. [24] is situated in the same vein. It proposed two scheduling algorithms to distribute processes across different cores such that miss rate is fairly distributed. [25] presented a cache aware scheduling algorithm which awards more processing time to a process when it suffers from cache contention. Therefore, [25] confirms in some way the fact that the processor can serve as a lever for controlling LLC utilization as we did.

Several researches [33, 31, 32, 38, 39, 40] have pointed the problem of LLC contention in the context of virtualized environments. However, very few of them have proposed a solution to this problem. [30] studied the effects of collocating different types of VMs under various VM to processor placement schemes to discover the best placement. The main limitation of this solution is the fact that it needs to know the applications which are running within VMs (to evaluate the collocation effects). [37] proposed a cache aware VM consolidation algorithm which chooses the consolidation plan so that the overall LLC misses are minimized in the IaaS. This solution considers the entire IaaS, not a single machine as we did.

Cache partitioning.

In this category we can distinguish two main approaches. The first approach is based on cache replacement policies.

[4]https://aws.amazon.com//ec2/instance-types/

It is independent from the execution environment (virtualized or not). According to this approach, [17, 19] proposed a dynamic insertion policy (DIP) which adapts the insertion policy (LRU or BIP) according to process memory activities. By doing so, DIP avoids to keep in the cache data of a VM which is parsing a large working set (a kind of disruptive VM). This solution is limited to a single category of disruptive VMs. [26] trends in the same direction by proposing PD (Protecting Distance), a cache replacement policy which protects cache lines that may be reused. [29] proposes a cache management policy called PIPP (Promotion/Insertion Pseudo-Partitioning). The latter partitions the cache by managing both cache insertion and promotion policies. [27] presents UCP (Utility-based Cache Partitioning), a runtime mechanism for partitioning the cache between multiple applications. UCP monitors each application using a cost estimation hardware circuit. Collected data are used by a partitioning algorithm to decide the amount of cache resources to allocate to each application. The policy is implemented through hardware and software modifications. [34] presented a QoS enabled cache architecture which enables more cache resources for high priority applications. Applications are assigned a priority level (this is comparable to our *llc_cap*). Then each cache line is tagged with a priority level.

The second approach addresses the cache contention issue using software based cache partitioning. Our solution uses this approach. [22, 23] proposed to partition the cache using page coloring [35]. Each VM is reserved a portion of the cache, and the physical memory is allocated such that a VM cache lines map only that reserved portion. This idea is very nice but difficult to implement. It depends on both the architecture of the cache and the replacement policy. Moreover, allocating physical pages to enforce the use of a specific place of the cache could be difficult to implement without wasting memory resources. For these reasons, [22, 23] only presented preliminary results.

Positioning of our work.

The main drawbacks of the above solutions are the following: cache partitioning solutions require the modification of hardware while VM placement solutions are not always optimal (VM placement is a NP-hard problem), **most important these solutions are not in the spirit of the cloud** which relies on the pay-per-use model: why not each VM books for an amount of cache utilization such that the virtualization system ensures that in the same way as it does for other coarse-grained resource types (CPU, memory, etc.). In this paper, we have proposed the Kyoto system which is a step in that direction.

7. CONCLUSION

We presented in this paper a new approach to address the issue of performance unpredictability due to LLC contention in a virtualized cloud environment. Our approach is inspired by the polluters pay principle which is applied as follows: any VM should pay for the amount of pollution it generates in the LLC. To implement it, we relied on hardware counters to monitor the utilization of the LLC by VMs, and we implemented a new vCPU scheduler which enforces at runtime a booked pollution level of a VM. We have presented a prototype for Xen system, KVM and Pisces. These prototypes have been evaluated using reference benchmarks (SPEC CPU2006), showing that they can enforce performance isolation between VMs even in case of LLC contention.

8. REFERENCES

[1] Dejan Milojicic (HP Labs), 'High Performance Computing (HPC) in the Cloud', Computing Now journal, September 2012.

[2] MicrosoftâĂŹs Top 10 Business Practices for Environmentally Sustainable Data Centers, 'http://www.microsoft.com/environment/news-and-resources/datacenter-best-practices.aspx'.

[3] Xi Chen, Chin Pang Ho, Rasha Osman, Peter G. Harrison, and William J. Knottenbelt, 'Understanding, modelling, and improving the performance of web applications in multicore virtualized environments', ICPE 2014.

[4] J. Ouyang, B. Kocoloski, J. Lange and K. Pedretti, 'Achieving Performance Isolation with Lightweight Co-Kernels', HPDC 2015.

[5] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira. 'Designing and Implementing Lightweight Kernels for Capability Computing'. Concurrency and Computation: Practice and Experience, 21(6), 2009.

[6] Boris Teabe, Alain Tchana, and Daniel Hagimont. 'Billing system CPU time on individual VM'. CCGRID, 2016.

[7] Lingjia Tang, Jason Mars, and Mary Lou Soffa, 'Contentiousness vs. Sensitivity: improving contention aware runtime systems on Multicore architecture', EXADAPT 2011.

[8] 5 Lessons We've Learned Using AWS: http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html

[9] Kyoto Protocol, http://unfccc.int/kyoto_protocol/items/2830.php

[10] Lingfang Zeng, Yang Wang, Wei Shi, Dan Feng, 'An Improved Xen Credit Scheduler for I/O Latency-Sensitive Applications on Multicores',CLOUDCOM 2013

[11] SPEC CPU2006, 'https://www.spec.org/cpu2006/.

[12] Jung Ho Ahn, Sheng Li, O. Seongil, and Norman P. Jouppi, 'McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling', ISPASS 2013.

[13] Prashanth P. Bungale and Chi-Keung Luk, 'PinOS: a programmable framework for whole-system dynamic instrumentation', VEEE 2007.

[14] lmbench, 'http://www.bitmover.com/lmbench/.

[15] Ulrich Drepper. What every programmer should know about memory; http://people.redhat.com/drepper/cpumemory.pdf, 2007.

[16] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat, 'Comparison of the Three CPU Schedulers in Xen', SIGMETRICS Performance Evaluation Review, 35(2) 2007.

[17] Moinuddin K., Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer, 'Adaptive insertion policies for high performance caching', ISCA 2007.

[18] Ruslan Nikolaev and Godmar Back, 'Perfctr-Xen: a framework for performance counter virtualization', VEE 2011.

[19] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer, 'Adaptive insertion policies for managing shared caches', PACT 2008.

[20] Jason Mars and Mary Lou Soffa, 'Synthesizing contention', WBIA 2009.

[21] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa, 'The Impact of Memory Subsystem Resource Sharing on Datacenter Applications', ISCA 2011.

[22] Xinxin Jin, Haogang Chen, Xiaolin Wang, Zhenlin Wang, Xiang Wen, Yingwei Luo, and Xiaoming Li, 'A Simple Cache Partitioning Approach in a Virtualized Environment', ISPA 2009.

[23] Xiaolin Wang, Xiang Wen, Yechen Li, and Yingwei Luo, 'A Dynamic Cache Partitioning Mechanism under Virtualization Environment', TrustCom 2012.

[24] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova, 'Addressing shared resource contention in multicore processors via scheduling', ASPLOS 2010.

[25] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith, 'Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler', PACT 2007.

[26] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum, 'Improving Cache Management Policies Using Dynamic Reuse Distances', MICRO 2012.

[27] Moinuddin K. Qureshi and Yale N. Patt, 'Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches', MICRO 2006.

[28] Gaurav Dhiman, Giacomo Marchetti, and Tajana Rosing, 'vGreen: a system for energy efficient computing in virtualized environments', ISLPED 2009.

[29] Yuejian Xie and Gabriel H. Loh, 'PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-core Shared Caches', ISCA 2009.

[30] Indrani Paul, Sudhakar Yalamanchili, and Lizy K. John, 'Performance impact of virtual machine placement in a datacenter', IPCCC 2012.

[31] Younggyun Koh, Rob C. Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu, 'An Analysis of Performance Interference Effects in Virtual Environments', ISPASS 2007.

[32] Padma Apparao, Ravi R. Iyer, and Donald Newell, 'Implications of Cache Asymmetry on Server Consolidation Performance', IISWC 2008.

[33] Natalie Enright Jerger, Dana Vantrease, and Mikko Lipasti, 'An Evaluation of Server Consolidation Workloads for Multi-Core Designs', IISWC 2007.

[34] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt, 'QoS policies and architecture for cache/memory in CMP platforms', SIGMETRICS 2007.

[35] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen, 'Towards practical page coloring-based multicore cache management', EuroSys 2009.

[36] Mirella Lapata, 'Automatic Evaluation of Information Ordering: Kendall's Tau,' Comput. Linguist. 32, 4, 2006.

[37] Jeongseob Ahn, Changdae Kim, Jaeung Han, Young-Ri Choi, and Jaehyuk Huh, 'Dynamic virtual machine scheduling in clouds for architectural shared resources', HotCloud 2012.

[38] Richard West, Puneet Zaroo, Carl A. Waldspurger, Xiao Zhang, "Online cache modeling for commodity multicore processors", SIGOPS 2010

[39] Marco Caccamo, Rodolfo Pellizzoni, Lui Sha, Gang Yao, Heechul Yun, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms", RTAS 2013

[40] Abhishek Gupta, Laxmikant V. Kale, Dejan Milojicic, Paolo Faraboschi, Susanne M. Balle,"HPC-Aware VM Placement in Infrastructure Clouds", IC2E 2013