Application-specific quantum for multi-core platform scheduler

Boris Teabe Alain Tchana Daniel Hagimont
Toulouse University, France
first.last@enseeiht.fr

Abstract

Scheduling has a significant influence on application performance. Deciding on a quantum length can be very tricky, especially when concurrent applications have various characteristics. This is actually the case in virtualized cloud computing environments where virtual machines from different users are colocated on the same physical machine. We claim that in a multi-core virtualized platform, different quantum lengths should be associated with different application types. We apply this principle in a new scheduler called AQL_Sched. We identified 5 main application types and experimentally found the best quantum length for each of them. Dynamically, AQL_Sched associates an application type with each virtual CPU (vCPU) and schedules vCPUs according to their type on physical CPU (pCPU) pools with the best quantum length. Therefore, each vCPU is scheduled on a pCPU with the best quantum length. We implemented a prototype of AQL_Sched in Xen and we evaluated it with various reference benchmarks (SPECweb2009, SPECmail2009, SPEC CPU2006, and PARSEC). The evaluation results show that AQL_Sched outperforms Xen's credit scheduler. For instance, up to 20%, 10% and 15% of performance improvements have been obtained with SPECweb2009, SPEC CPU2006 and PARSEC, respectively.

Keywords quantum; multi-core; scheduler; virtual machine

1. Introduction

Cloud data centers are spreading very fast. Most of the time they are virtualized so that several user applications can be run on the same physical machine in isolated virtual machines (VM). Since such datacenters may have a high number of end-users, they may host many different application types with various characteristics. For instance, [1] reports that Amazon cloud runs a wide spectrum of applications including high traffic web sites such as reddit [2], Genome analysis platforms such as Illumina [3], or SAP applications. Consequently, it is difficult to design a resource management policy which satisfies every application. In this paper, we focus on processor scheduling in such virtualized environments.

Scheduling can have a significant influence on application performance [4]. An important parameter of a scheduler is the quantum length. Deciding on the latter can be problematic [5, 6], especially when various application types have to be managed as it is the case in cloud data centers. For example, a higher quantum length (e.g. 50ms) penalizes latency-critical applications while it favours memory intensive applications as it reduces cache contention. The issue is even more complex as different application types often run concurrently on the same server in the cloud, due to VM consolidation[1] (packing the maximum number of VMs atop the minimum number of servers). As reported in [8, 9], common scheduling algorithms such as the Linux Completely Fair Scheduler (CFS) make decisions which lead to frequent service level objective violations when latency-critical tasks are co-located with best effort tasks. In this paper, we subscribe to that conclusion and we claim that the use of a fixed quantum length for scheduling all application types as done by most popular virtualization systems (e.g. 30ms in Xen [10] and 50ms in VMware [12]) exacerbates this issue. An approach to this issue is to manage different quantum lengths for different application types. For example, we can improve the performance of a high traffic web site by about 62% if a quantum length of 1ms instead of 30ms (the default value) is used in Xen.

This issue is addressed by recent research works. [13] introduces the BOOST mechanism for improving the latency of IO applications in a Xen system. However, this solution is only efficient when applications exclusively run an intensive IO workload (see Section 4 for more details). [14] also focuses on the improvement of IO intensive applications. They propose to expose a processor pool (called turbo processor) to each VM as a "co-processor" which is dedicated to kernel

EuroSys '16, April 18-21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2901318.2901340
Reprinted from EuroSys '16,, [Unknown Proceedings], April 18-21, 2016, London, United Kingdom, pp. 1–14.

[1] For minimizing the energy consumed by servers (which is the largest fraction (50-70%)[7] of the total cost of ownership), maximizing server utilization by means of VM consolidation is common practice in the cloud.

threads that require synchronous processing such as I/O requests. These turbo processors are configured with a lower quantum length. In a VM all the kernel threads that are performing I/O requests are scheduled on these turbo processors. This solution is limited since the configuration should be done manually: the user should know in advance the kernel threads which perform IO requests in order to always schedule them on turbo processors. By using a lower quantum length, [6] improves the performance of VMs which perform a significant number of spin-locks. This solution penalizes last-level cache friendly applications (see Section 3) since it increases the probability of cache contention.

The common limitation of existing solutions is their restriction to a single application type. Putting all of them together in order to cover all application types is not straightforward. This is the purpose of our work. Taking into account application types at the scheduler level has already been proposed in Linux schedulers which make the distinction between real-time tasks and others. However, this distinction is elementary because a task type is dictated by its priority which is assigned by the user. We identified 5 main application types which are commonly deployed in the cloud: (1) IO intensive applications (they are latency-critical), (2) applications which run concurrent threads and rely on spin-locks for synchronization, (3) LLC[2] friendly applications (their working set size (WSS) fits within the LLC, thus they are very sensitive to LLC contention), (4) memory intensive applications whose WSS overflows the LLC, and (5) CPU burn applications whose WSS fits within low-level caches (e.g. L1/L2 in a 3-layer cache architecture). In this paper, we claim that a specific quantum length (the "best" one) should be associated with each application type, thus improving the performance of all applications wherever they run in the data center.

We introduce AQL_Sched, an Adaptable Quantum Length Scheduler which follows that direction. In contrast with existing solutions [6, 14, 15], AQL_Sched covers a wide range of application types. AQL_Sched dynamically associates an application type with each virtual CPU (vCPU) and schedules vCPU on processor pools according to their type and their memory activity (to reduce LLC contention). Processor pools are configured with the best quantum length for the associated application type. By scheduling vCPU not only according to their type, but also according to their memory activity, AQL_Sched also addressed the LLC contention problem. AQL_Sched implements three main features, each of them addressing key challenges:

- A vCPU type recognition system (vTRS for short). The hypothesis of a fixed type for a VM vCPU during its overall lifetime is not realistic. Several different thread types can be scheduled by the guest OS on the same vCPU. Therefore, vTRS should be accurate, i.e dynamically identify the right type for each vCPU while minimizing both overhead (use the minimum CPU time) and intrusivity (avoid as far as possible both the intervention of cloud users and the modification of the guest OS).
- The identification of the "best" quantum length. AQL_Sched should know the best quantum length to use for each application type. The best quantum lengths are obtained through an offline calibration. The latter should be done on a representative application set in order to cover all existing applications.
- The clustering of vCPUs. It consists in mapping vCPUs of the same type (cluster) to a pool of physical processors (pCPUs). This clustering also addresses the LLC contention issue by taking into account vCPU memory activity. This should be done while ensuring fairness as common cloud schedulers do: each VM should receive its booked CPU resources. Since processors are organized in pools, fairness could be difficult to achieve when the number of processors is limited (the distribution unit in pools is the processor, which may be too coarse-grained with a small number of processors).

We implemented an AQL_Sched prototype in the Xen virtualization system as an extension of its Credit scheduler [10]. We performed extensive performance evaluations of our prototype, relying on both micro-benchmarks and reference benchmarks (SPECweb2009 [16], SPECmail2009 [17], SPEC CPU2006 [18], and PARSEC [20]). The results show that adapting the quantum length according to application types leads to significant benefits in comparison with the native Xen scheduler: up to 20% for SPECweb2009, 25% for SPECmail2009, 15% for PARSEC applications, and 10% for SPEC CPU2006 applications. The results also demonstrate both the scalability of the prototype and its negligible overhead. In addition, we compare our prototype with existing solutions (vTurbo [14], vSlicer [15], and Microsliced [6]).

The rest of the article is organized as follows. Section 2 presents both the background and the motivations of our work. Contributions are presented in Section 3 while Section 4 presents evaluation results. A review of related works is presented in Section 5 and the conclusion is drawn in Section 6.

2. Motivations

VM scheduling is carried out through vCPUs assignation to physical processors (hereafter pCPUs). This scheduling could be a source of performance improvement in virtualized systems as well as it is the case in native systems. However, we notice that all improvements which have been obtained through scheduling in native OSes are ineffective when the OS runs as a VM. The reason is that in a virtualized system, the hardware is controlled by the hypervisor, but not the guest OS. Fig. 1 shows an illustration where VM1's

[2] LLC stands for Last Level Cache.

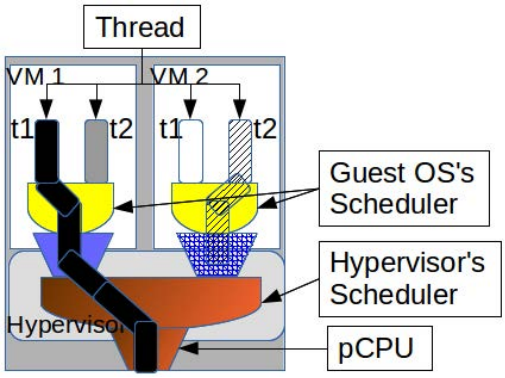

Figure 1. Scheduling in a virtualized system: This figure shows that the scheduling in a virtualized system is performed at two levels: guest OS level and hypervisor level.

thread t1 accesses the processor, leaving VM2's thread (t2) out, whereas it has been scheduled in by VM2's scheduler. Studying scheduling means answering two fundamental questions:

- (Q_1) which vCPU should acquire the pCPU (at a given moment)?
- (Q_2) for how long (also called quantum length) it can use the pCPU without pre-emption?

Answering correctly both Q_1 and Q_2 allows the scheduler to give to any vCPU the opportunity to use a pCPU, while avoiding starvation and ensuring fairness. The next section presents how Q_1 and Q_2 are answered in Xen [11], one of the most popular virtualization system.

2.1 Scheduling in Xen

Xen [11] is a popular open-source virtualization system widely used by cloud providers such as Amazon EC2. It supports three schedulers which are: Borrowed Virtual Time (BVT), Simple Earliest Deadline First (SEDF), and Credit. [28] provides a detailed description of these schedulers. In this paper, we focus on the Credit scheduler which is the default and the most frequently used scheduler. The Credit scheduler allocates CPU in proportion to VM assigned weights. A second parameter (called *cap*) allows limiting the amount of computation power received by a VM. The Credit scheduler works as follows. For each VM v, it defines *remainCredit* (a scheduling variable) initialized with *cap*. Each time a v vCPU is scheduled on a pCPU, (1) the scheduler translates into a credit value (let us say *burntCredit*) the time spent by v on that pCPU (this is performed every tick, typically 10ms). (2) Subsequently, the scheduler computes a new value for *remainCredit* by subtracting *burntCredit* from the *remainCredit* previous value. If the computed value is lower than a threshold, v enters the *OVER* state, i.e it cannot access a pCPU. Otherwise, v enters the *UNDER* state and is queued. VMs with the *OVER* state have their *remainCredit* periodically increased, according to their initial *cap*, in order to give them the opportunity to become schedulable. VMs whose state is UNDER are scheduled in a round-robin manner (answer to Q_1). Regarding Q_2, Credit uses 30ms as the quantum duration.

2.2 The problem

From the previous section, the following issues can be highlighted:

- Regarding (Q_1). vCPUs are scheduled in a *round-robin* way. This is known to disadvantage VMs which often underuse the CPU (as IO intensive applications tend to do) in favour of VMs that use their full quantum. This issue can be handled by using a lower quantum length (related to Q_2).
- Regarding (Q_2). The quantum length is given by a *fixed* value. Knowing that VMs with different characteristics could run in the cloud, using a fixed quantum length may be beneficial for some VMs while harmful for others (see Section 3.4).

We can conclude that providing a correct answer to Q_2 is crucial since, as a spin-off, it allows addressing issues related to Q_1. Therefore, we focus in this paper on the issues related to Q_2: the use of a fixed quantum length for scheduling all application types. This problem is not specific to Xen. For example, VMware, a proprietary virtualization solution which is the leader in the domain, also uses a fixed quantum length [12] which is 50ms. Identifying the best quantum length for each application type is crucial for improving the performance of all cloud applications at the same time. In this paper, we propose a new way for scheduling VMs, called AQL_Sched (stands for Adaptable Quantum Length Scheduler), which goes in that direction. The next section presents AQL_Sched.

3. AQL_Sched

In this section, we start with the presentation of the basic idea behind AQL_Sched. Afterwards, we detail each AQL_Sched scheduler design dimension. Although we rely on Xen for illustration, our contribution is quite general and is applicable to any virtualization system.

3.1 The basic idea

As discussed in the previous section, performance heavily depends on the quantum lengths used to schedule vCPUs. We define a vCPU type at a given instant as the thread type using the vCPU within the VM at that instant. **The basic idea behind AQL_Sched is the exclusive scheduling of the same vCPU type atop a dedicated pool of pCPUs, using the "best" quantum length (the quantum which leads that type to its best performance).** To this end, the AQL_Sched scheduler includes three prominent features:

- A vCPU type recognition system (vTRS for short): we identified all vCPU types (at least the most prevalent)

which could be run within the cloud. Section 3.2 presents the various vCPU types. Therefore, the AQL_Sched scheduler implements an online vTRS (presented in Section 3.3) which periodically evaluates the actual type of a vCPU.

- The identification of the best quantum length: based on extensive experiments (presented in Section 3.4), we identified the best quantum length for each application type.
- The clustering of vCPUs: Once all vCPU actual types are identified, vCPUs are organized in clusters according to their type and their memory activity. A pCPU set is associated to each cluster, while ensuring fairness. pCPU schedulers which belong to the same cluster are configured with the same quantum length, the best one according to the calibration results.

The next sections detail each dimension of our scheduler.

3.2 Application types

We identified three main application characteristics: CPU burn, IO intensive, and concurrent. In this section, we present the various application types and we also show how they can be impacted by the use of an inappropriate quantum length.

CPU burn applications and the cache contention problem. These are applications which intensively use the processor, as well as the main memory. We classify CPU burn applications into three sub-types according to the use of CPU caches[3]:
(i)-Last-level cache friendly applications (noted $LLCF$): their working set size (WSS for short) fits within the last-level cache (LLC). Therefore, they are very sensitive to LLC pollution [30, 31]. A lower quantum length leads to an increase of the number of context switches, thus reducing the probability for $LLCF$ applications to find their data in the LLC.
(ii)-Trashing applications (noted $LLCO$): their WWS overflows the LLC. They do not suffer from cache pollution. However, they could act as disturbers for LLCF, according to the cache replacement policy.
(iii)-Low-level cache (LoLC) friendly applications
(noted $LoLCF$): their WWS fits within LoLC (e.g. L1 cache). Such applications are agnostic to cache pollution since handling LoLC misses are much less expensive than LLC misses.

IO intensive applications (noted $IOInt$) and the interrupt handling problem. These are applications which intensively generate IO traffic. In this paper, we consider both disk and network traffic. In a native system, IO requests are materialized by interrupts which are immediately handled by the OS. Therefore, having the entire control of the hardware and knowing which process is waiting for an IO request, the OS is able to immediately give the CPU to a previously blocked process once an interrupt related to it occurs. Things are different in a virtualized system as illustrated in Fig. 1. Let us consider the arrival of an interrupt within the VM just before it is scheduled out. The interrupt will be handled only when a vCPU of that VM acquires a pCPU. This may occur tardily, thus increasing IO requests latency [14, 15]. It depends on both the number of vCPUs in the system and the quantum length.

Concurrent applications (noted $ConSpin$) and the lock holder preemption problem. Such applications are composed of several threads which compete for the same object (e.g. a data structure) and thus need to synchronize themselves. Two main mechanisms can be used for synchronization purpose: semaphores and spin-locks (the mostly used mechanism). The main difference between the two mechanisms is the way threads are waiting for the lock to be released. In the semaphore case, a blocked thread loses the processor when waiting for the lock to be released. This is not the case with spin-locks, where the waiting thread spins while waiting for the lock to be released, thus consuming processing time. This is why spin-locks are generally used for short duration locking. To improve spin-lock application performance, the OS ensures that a thread which holds a lock on an object is not be preempted until it releases the lock. This caution is ineffective in a virtualized system since vCPUs are in their turn scheduled atop pCPUs. This situation can lead blocked threads to consume their entire quantum to carry out an active standby. This situation is exacerbated with higher quantum lengths [6].

3.3 The vCPU type recognition system (vTRS)

3.3.1 The general functioning

Our scheduler implements an online vTRS. The latter relies on a monitoring system which periodically (every 30ms, called the monitoring period) collects the value of metrics needed to identify a vCPU type. It takes its decision after n monitoring periods. Notice that a small value of n (e.g. 1) allows taking quickly into account sporadic vCPU type variations. However, this may impact the performance of the application which uses the vCPU. Indeed, frequent type variations imply frequent vCPU migrations between pCPUs (because of clustering, see Section 3.5), which is known to be negative for the performance of applications. We have experimentally seen that setting n to 4 is acceptable.

For any metric-based workload recognition, it is crucial that the set of chosen metrics allows to uniquely identify all workload behaviour types. vTRS relies on the following metrics (Section 3.3.2 presents how they are collected) to identify a vCPU type: the number of IO requests processed by the vCPU (noted $IOInt\_level$), the number of spin-locks performed by its VM (noted $ConSpin\_level$), the LLC miss ratio (noted LLC_MR_level), and the LLC reference ratio

[3] CPU caches have a strong impact on performance when a wrong quantum length is used.

(noted LLC_RR_level). We have normalized all metrics in order to have a common unit: a percentage. The latter services as a cursor which indicates to what extent (a probability) the vCPU is close to a vCPU type. Each cursor (noted xx_cur) is computed as follows.

$IOInt$ **and** $ConSpin$ **cursors:**

$$\begin{aligned} &if\ (*\_level < *\_LIMIT) \\ &\qquad *\_cur = \frac{*\_level \times 100}{*\_LIMIT} \\ &else \\ &\qquad *\_cur = 100 \end{aligned} \tag{1}$$

where * is $IOInt$ or $ConSpin$. To explain equation 1, let us consider * be $IOInt$. $IOInt\_level$ is the number of IO requests processed during the previous monitoring period. $IOInt\_LIMIT$ is the threshold above which the vCPU is considered to be 100% $IOInt$.

$LoLCF$**,** $LLCF$**, and** $LLCO$ **cursors:**

Recall that these are sub-types of what we called CPU burn applications (see Section 3.2). The computation of their cursors relies on the same set of metrics and should respect the following equation:

$$LoLCF\_cur + LLCF\_cur + LLCO\_cur = 100 \tag{2}$$

$LoLCF$ cursor:

$$\begin{aligned} &if\ (LLC\_RR\_level < LLC\_RR\_LIMIT) \\ &LoLCF\_cur = \frac{(LLC\_RR\_LIMIT - LLC\_RR\_level) \times 100}{LLC\_RR\_LIMIT} \\ &else \\ &\qquad LoLCF\_cur = 0 \end{aligned} \tag{3}$$

where LLC_RR_LIMIT is the maximum LLC references a $LoLCF$ is allowed to generate. Indeed, a $LoLCF$ application makes very few LLC references (not to say nil). If the vCPU generates more than LLC_RR_LIMIT, it will be either $LLCF$ or $LLCO$.

$LLCF$ cursor:

$$\begin{aligned} &if\ (LLC\_MR\_level < LLC\_MR\_LIMIT) \\ &\quad LLCF\_cur = min(100 - LoLCF\_cur; \\ &\quad \frac{(LLC\_MR\_LIMIT - LLC\_MR\_level) \times 100}{LLC\_MR\_LIMIT}) \\ &\qquad else \\ &\qquad\qquad LLCF\_cur = 0 \end{aligned} \tag{4}$$

where LLC_MR_LIMIT is the maximum LLC misses a $LLCF$ is allowed to generate. In fact, since a $LLCF$ is cache friendly, the LLC miss number it could generate should be insignificant. Above LLC_MR_LIMIT, the vCPU is considered to be $LLCO$ (trashing).

$LLCO$ cursor:

$$LLCO\_cur = 100 - LoLCF\_cur - LLCF\_cur \tag{5}$$

A matrix of 5 lines (one per cursor type) and n (number of monitoring period before deciding on the type of a vCPU) entries is associated with each vCPU for recording all the metric values . At the end of each monitoring period, each cursor value is computed and stored in the last entry of the corresponding line (the modification is done in a sliding-window way). The average value of each line (noted xx_cur_avg) is then computed. The vCPU type corresponds to the cursor type with the highest xx_cur_avg. Note that it is difficult for a vCPU to have two or more cursor types with the same xx_cur_avg, and even more to have many xx_cur_avg equal to the highest value. In the evaluation section (Section 4.1), we show how these metrics are used.

3.3.2 Monitoring systems

This section presents the monitoring systems used for tracking $IOInt\_level$, $ConSpin\_level$, LLC_RR_level, and LLC_MR_level. Building these systems includes two key challenges:

- Intrusiveness. Given the diverse set of applications that might run in the cloud, vTRS cannot rely on any prior application knowledge, semantics, implementation details, or highly-specific logs. Furthermore, vTRS assumes that it has to work well without having any control over the guest VMs or the running applications. This is a desirable constraint given that we target hosting environments that provides only a "bare-bones" virtual server.
- Overhead. Since vTRS might be running all the time, it should drain as few CPU time as possible.

Using low-level metrics to infer the workload behaviour is interesting as it allows vTRS to uniquely identify different workloads without requiring knowledge about the deployed applications. The implementation of vTRS is based on low-level metrics.

The monitoring system for $IOInt\_level$**.** In the Xen system, the occurrence of an IO request can be observed at the hypervisor level. Following the split-driver model [45] (used by Xen as many other virtualization systems), the communication between IO device drivers and guest OSes requires the intervention of both the hypervisor (e.g. interrupt forwarding using event channel) and the device domain (typically the privileged domain). Therefore, we propose a monitoring system based on event channel analysis, implemented within the hypervisor. Each vCPU is associated with an IO request counter. Every time an event is related to an IO request, the IO request counter of the involved vCPU is incremented.

The monitoring system for $ConSpin\_level$**.** The monitoring system here is straightforward since it relies on the modern hardware ability to detect spinning situations. For instance, in Intel Xeon E5620 processor, such situations can be trapped with
EXIT_REASON_PAUSE_INSTRUCTION (the "Fancy" feature, Pause Loop Exiting). We implemented a hypervisor level tool for tracking such situations. To address architectures which do not include EXIT_REASON_PAUSE

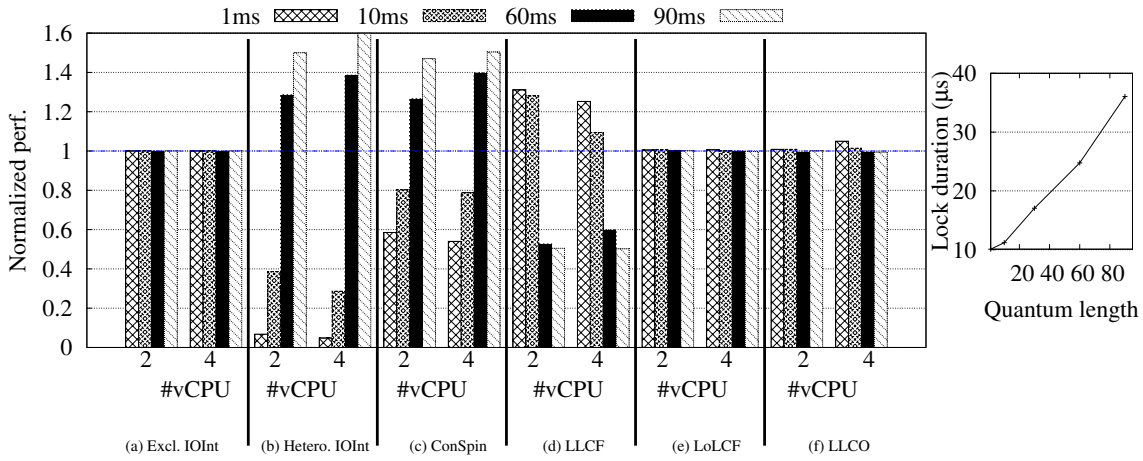

Figure 2. Calibration results: This figure presents the calibration results where values are normalized over the application type performance when it runs with the Xen default quantum length (30ms). The smaller the performance graph bar the better the performance.

_INSTRUCTION, we propose a second implementation which relies on a slight modification of the guest OS. The Xen hypercall framework is augmented with a new hypercall which wraps the spin-lock API. By this way, the hypervisor collects the number of spin-locks performed during each monitoring period.

The monitoring system for LLC_RR_level and LLC_MR_level. The monitoring system here relies on Performance Monitoring Units, provided by nearly all recent hardware. In the Xen system case, the hypervisor level framework perfctr-xen [29] can be used to collect LLC misses, LLC references and the number of executed instructions, necessary for the calculation of both LLC_RR_level and LLC_MR_level.

3.4 Quantum length calibration

One of the AQL_Sched scheduler key feature is its ability to know the best quantum length to use for scheduling a given vCPU type. Similarly to several research works on this topic [6, 15], identifying the best quantum length requires a calibration phase. We automated the latter by relying on both an autonomic deployment framework [22] and a self-benchmarking tool [23]. This section presents the set of experiments we performed for calibration.

3.4.1 The experimental setup

We relied on micro-benchmarks (presented in Table 1) either written for this article purpose or taken from previous works. We selected these benchmarks because they are representative of each application type. Experiments were performed on a HP machine with Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz processor. Its characteristics are presented in Table 2. The machine runs a Ubuntu Server 12.04 virtualized with Xen 4.2.0. All the application type calibrations follow the same scenario: a baseline VM (the VM hosting the application type being calibrated) colocated with several other VMs (hosting various workload types). For a specific type calibration, the difference between experiments comes from the quantum length and the number of vCPUs (from 2 vCPUs to 4 vCPUs[4]) sharing the same pCPU. We experimented with four quantum length values: 1ms, 10ms, 30ms, 60ms, and 90ms. Unless otherwise indicated, a VM is configured with a single vCPU. The next section presents the calibration results. Notice that all these results are platform dependent.

Benchmark	Description	Type
Wordpress [24]	simple web application	$IOInt$
Kernbench [25]	Linux compilation	$ConSpin$
[27]	parsing of a linked list	$LoLCF$, $LLCF$, and $LLCO$

Table 1. Benchmarks used for calibration. Each benchmark is representative of an application type.

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

Table 2. The characteristics of the experimental machine.

3.4.2 Results

All results presented in this section (reported in Fig. 2) are normalized over the application type performance when running with the Xen default quantum length (30ms). The smaller the performance graph bar the better the performance.

[4] [33] reports that a pCPU runs 4 vCPUs on average in a virtualized datacenter.

$IOInt$**.** From Fig. 2 (a), we can see that a vCPU which exclusively runs a network workload is quantum length agnostic. In fact, to accommodate low latency, a *BOOST* state was recently introduced in Xen [13] to prioritize the scheduling of a vCPU which was blocked waiting for an I/O event. Unfortunately, this mechanism is inefficient when the vCPU runs an heterogeneous workload (the web server also executes CGI scripts which consume a significant CPU time), as we can see in Fig. 2 (b). In fact, a vCPU is set to the BOOST state only if it has not entirely consumed its previous quantum. This is not the case with a heterogeneous workload. Fig. 2 (b) shows that lower quantum lengths are beneficial for such workloads. According to our quantum length discretization, the best length is 1ms. The latter will therefore be used as the quantum length of $IOInt$ vCPUs.

$ConSpin$**.** We configured kernbench to use 4 threads. Calibration results, presented in Fig. 2 (c), show that the best quantum length for this vCPU type is 1ms. In fact, higher quantum lengths increase the average duration of locks (see Fig. 2 rightmost, when the indicator VM uses 4 vCPUs).

$LLCF$**.** The micro-benchmark [27] was configured to use half of the LLC. Fig. 2 (d) shows that a higher quantum length is better for $LLCF$ applications. The best length we found is 90ms.

$LoLCF$ **and** $LLCO$**.** The micro-benchmark [27] was configured to use 90% of the L2 cache for the $LoLCF$ calibration while more than the LLC is used for $LLCO$. Fig. 2 (e) and (f) respectively show that $LoLCF$ and $LLCO$ are quantum length agnostic. Therefore, they will be used for balancing vCPUs clusters (see below).

3.5 Clustering

After each invocation of the vTRS, vCPUs are organized in clusters so that those which perform better with the same quantum length are scheduled atop the same pool of pCPUs. Clustering has to face two challenges: fairness (which is a property of cloud schedulers) and LLC contention (since vCPUs are grouped, clustering cannot ignore related work advices about vCPU colocation). We introduce a smart clustering solution which takes into account these challenges. To do so, we use a two-level clustering algorithm. The goal of the first level algorithm is to fairly distribute vCPUs on sockets (set of pCPUs) while avoiding as much as possible the colocation of disturbers (hereafter called "trashing") and sensitive (hereafter called "non-trashing") vCPUs. The second level algorithm works at the granularity of a socket (note that the LLC contention issue cannot be addressed here). Firstly, it organizes vCPUs per cluster according to their quantum length compatibility (see below). Secondly, it fairly associates a set of pCPUs with each cluster. The rest of the section describes the two algorithms.

At a first level (algorithm 1), vCPUs are organized into two groups ("trashing" and "non-trashing") according to their LLC pollution intensity (lines 4-10). vCPUs which are part of the trashing list are $LLCO$ (obviously), and $IOInt$ and $ConSpin$ whose $LLCO$ cursor is tremendous (let us say greater than 50%). In that case, they are noted $IOInt^{+}$ and $ConSpin^{+}$. Concerning the non-trashing list, we have $LLCF$ and $LoLCF$ (obviously), and $IOInt$ and $ConSpin$ (noted $IOInt^{-}$ and $ConSpin^{-}$) which are not part of the trashing list. Following that first step, trashing and non-trashing vCPUs are fairly distributed among sockets (lines 12-17). In order to minimize remote memory access which may result in performance degradation under NUMA architectures, our algorithm prevents as much as possible spreading vCPUs which belong to the same VM among different sockets. This is achieved by ordering vCPUs per VM before assigning them to sockets (line 3). For balancing purpose, the socket which hosts the last "trashing" vCPUs could also be assigned "non-trashing" vCPUs (line 15) when the number of "trashing" vCPUs is not a multiple of "n". By putting $LoLCF$ vCPUs at the beginning of the "non-trashing" list (line 11), we minimize the probability to colocate $LLCF$ vCPUs together with trashing vCPUs (the latter would disturb the former).

Algorithm 1 First level clustering.

Input:
totVCPUs: total number of vCPUs in the system
totSockets: total number of sockets in the system
Begin
1: trashing=∅
2: non-trashing=∅
3: order vCPUs so that those which belong to the same VM follow each other
4: **for each** vCPU v_i **do**
5: **if** $max(LLCF\_cur\_avg, LLCO\_cur\_avg, LoLCF\_cur\_avg) = LLCF\_cur\_avg$ **then**
6: trashing=trashing∪$\{v_i\}$
7: **else**
8: non-trashing=non-trashing∪$\{v_i\}$
9: **end if**
10: **end for**
11: order non-trashing vCPUs so that $LoLCF$ vCPUs appear at the beginning
12: n=$\frac{totVCPUs}{totSockets}$
13: **for each** $socket\ s_i$ **do**
14: auxSet=(trashing!=∅)?trashing:non-trashing
15: select the first n vCPUs from auxSet and assign them to socket s_i
16: apply Algorithm 2 to socket s_i
17: **end for**
End

The second level clustering (algorithm 2) works at the socket granularity. It organizes vCPUs according to quantum length affinity rather than vCPU types. In fact, from calibration results, we made two observations: (1) some distinct types reach their best performance with the same quantum length ($IOInt$ and $ConSin$ for example), and (2) $LoLCF$ and $LLCO$ vCPUs are quantum length agnostic. From these observations, we define the notion of quantum length compatibility (QLC for short) as follows: a vCPU set C, is q_QLC if all its vCPUs reach their best performance with the quantum length q. For instance, $\{IOInt, ConSpin\}$ is $1ms\_QLC$. Therefore, the clustering algorithm works as follows. First, all vCPUs (except $LoLCF$ and $LLCO$ ones) are organized into n clusters (lines 2-7), n being the number of calibrated quantum lengths. $LoLCF$ and $LLCO$ are used for balancing clusters (line 10). Subsequently, pCPU pools

Algorithm 2 Second level clustering (socket granularity).

Input:
totVCPUs: total number of vCPUs on the socket
totPCPUs: total number of pCPUs on the socket
Begin
1: i=0 //for indexing clusters (C_i^q)
2: **for each** quantum length q found by calibration **do**
3: i++
4: C_i^q =all vCPUs which are q_QLC
5: exclude $LoLCF$ and $LLCO$ vCPUs from C_i^q
6: pCPUsPool$_i$ = $\emptyset$
7: **end for**
8: n=i
9: pCPUsPool$_{n+1}$ = $\emptyset$
10: use $LoLCF$ and $LLCO$ for balancing clusters (C_*^*)
11: k=$\frac{totVCPUs}{totPCPUs}$ //the number of vCPUs per pCPU
12: i=1
13: **for each** pCPU p **do**
14: **if** k $\leq$ sizeOf(C_i^q) **then**
15: S={select k unlabelled vCPUs from C_i^q}
16: pCPUsPool$_i$=pCPUsPool$_i$$\cup${p}
17: **else**
18: S={select sizeOf(C_i^q) unlabelled vCPUs from C_i^q}
19: **if** i<n **then**
20: S=S$\cup${select (k-sizeOf(C_i^q)) unlabelled vCPUs from
21: $C_{i+1}^q, .., C_{i+j}^q$}
22: S's vCPUs are removed from their initial cluster and assigned to cluster C_{n+1}^{dq} (dq is the default quantum length (30ms)).
23: pCPUsPool$_{n+1}$=pCPUsPool$_{n+1}$$\cup${p}
24: i=i+j
25: **else**
26: pCPUsPool$_i$=pCPUsPool$_i$$\cup${p}
27: **end if**
28: **end if**
29: Label S's vCPUs //they have already be treated
30: **end for**
31: **for each** cluster C_i^q, including C_{n+1}^{dq} **do**
32: **for each** pCPU p in pCPUsPool$_i$ **do**
33: configure p's scheduler to use the cluster's quantum length
34: **end for**
35: **end for**
End

are built such that fairness is respected (lines 11-29). During that phase, some pCPUs (less than n) can be assigned vCPUs belonging to distinct clusters (line 20). Such vCPUs are assigned to a default cluster (the default quantum length will be used for scheduling). Finally, the algorithm reconfigures each pCPU scheduler so that the appropriate quantum length is used (lines 30-34). Note that scheduling within a cluster is ensured by the native scheduler, which is supposed to be fair.

Fig. 3 illustrates our clustering algorithms. We consider a four-socket machine, each socket having 4 pCPUs. One socket is dedicated to the dom0 (the privileged domain). The machine runs 12 $IOInt^+$, 7 $ConSpin^-$, 17 $LLCF$, and 12 $LLCO$ (a total of 48 vCPUs). Therefore, fairness is respected if each pCPU runs almost 4 vCPUs. As shown in Fig. 3, each socket is assigned exactly 16 vCPUs at the end of the first algorithm. With the second algorithm, 6 clusters are formed at the end of its execution. Let us explain this result by focusing on what happens in the first and the third sockets. All vCPUs in the first socket are $1ms\_QLC$ ($IOInt$ requires 1ms while $LLCO$ is quantum length agnostic), thus forming a unique cluster. Concerning the third socket, 2 clusters have initially been formed: C_4^{90} (for all 9 $LLCF$) and

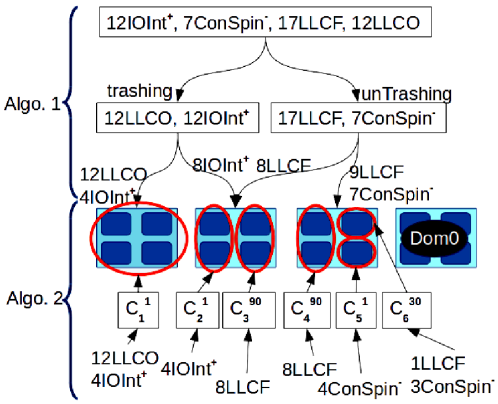

Figure 3. An illustration of our 2-level clustering solution: We consider a four-socket machine, each socket having 4 pCPUs. One socket is dedicated to the dom0 (the privileged domain).

C_5^1 (for all 7 $ConSpin^-$). Knowing that assigning pCPUs to clusters have to ensure fairness (4 vCPUs per pCPU), it was not possible to do that for C_4^{90} and C_5^1 unaltered. Therefore, respectively one vCPU and three vCPUs have been removed from C_4^{90} and C_5^1 in order to form the last cluster C_6^{30}. Since the latter contains vCPUs which are not QLC, it is configured to use the default quantum length. The evaluation of this scenario is presented in the next section.

4. AQL_Sched evaluation

This section presents the evaluations results (implemented within Xen) of our prototype. The evaluation covers the following aspects: the accuracy of vTRS, the effectiveness of the prototype, and finally the prototype overhead. By default, the experimental context is identical to the environment presented in Section 3.4. Note that a common practice [6, 14, 15] is to pin privileged domains' (dom0, driver domains) vCPUs to dedicated cores. Therefore, they are not considered by our scheduler. Otherwise specified, all the results are normalized over the performance with the default Xen Scheduler. A normalized performance value lower than 1 (respectively higher than 1) means that the application performs better (respectively worse) with Xms of quantum length than 30ms of quantum length.

4.1 Accuracy of vTRS

The first experiments evaluate the online vTRS, and at the same time they validate the robustness of the calibration results.

Benchmarks. These experiments were performed using reference benchmarks: SPECweb2009 [16], SPECmail2009 [17], and SPEC CPU2006 [18], which are implementations of an internet service, a corporate mail server, and a set of CPU intensive applications, respectively. The performance is evaluated with the network request average latency for the former, the average time needed for handling

$IOInt$	SPECweb2009, SPECmail2009
$ConSpin$	bodytrack, blackscholes, canneal, dedup, facesim, ferret, fluidanimate freqmine, raytrace, streamcluster
$LLCF$	astar, Xatanbmck, bzip2 gcc, omnetp
$LoLCF$	hmmer, gobmk, perlbench sjeng; h264ref
$LLCO$	mcf, libquantum

Table 3. Application types recognition: This table shows the type of each experimented application as detected by vTRS (an illustration of vTRS execution is shown in Figure 4).

Scenarios	Clusters	Applications	#pCPUs
S_1	C_1^1	$5ConSpin, 3LoLCF$	2
	C_2^{90}	$5LLCF, 3LoLCF$	2
S_2	C_1^1	$5IOInt, 3LLCO$	2
	C_2^{90}	$5LLCF, 3LLCO$	2
S_3	C_1^{90}	all	all
S_4	C_1^1	$4IOInt, 4ConSpin$	2
	C_2^{90}	$4LLCF, 4LLCO$	2
S_5	C_1^1	$4IOInt, 4ConSpin$	2
	C_2^{90}	$4LLCF, 2LLCO, 2LoLCF$	2

Table 5. Clustering applied to each scenario presented in Table 4.

a mail operation for the second, and each program execution time for the latter. We also used PARSEC [20], a set of multithreaded programs, to evaluate applications using spin-locks for synchronization. PARSEC benchmark's performance is measured with execution time.

Results. Fig. 4 shows for 5 representative applications, 50 collected decision metric values $*\_cur\_avg$ used by vTRS for inferring the vCPU type. We define an application type as the type having its curve higher than the others most of the time (this is represented in the figure by the red lines). We can see that vTRS effectively identifies each benchmark type. For example, SPECweb2009 is identified as $IOInt$, which is its known behavior. Table 3 summarizes each application type according to vTRS results. Notice that for $LLCF$, $LoLCF$ and $LLCO$, these results depend on our experimental environment. Indeed, an application which has been identified as $LLCO$ in our environment may be identified as $LLCF$ on a machine with a larger LLC.

Concerning the calibration result robustness, Fig. 5 shows the normalized performance of each application when running with different quantum lengths. The experimental environment and procedure in this experiment are the same as in Section 3.4.1, but we limited the evaluation to 4 vCPUs sharing a pCPU (This is the most common case observed in cloud platforms [6]). We can see that each application obtains its best performance when the quantum length corresponding to the type identified by vTRS is used. For example, the best performance of SPECweb2009 (which is typed as $IOInt$) is obtained when the quantum length is 1ms. Remember that 1ms corresponds to the calibrated quantum length for $IOInt$ applications.

4.2 AQL_Sched effectiveness and comparison with existing approaches

Evaluation on a single-socket machine.
We firstly evaluate the effectiveness of the prototype using simple use cases which correspond to different colocated application scenarios (presented in Table 4). Each scenario runs 16 vCPUs on 4 pCPUs, resulting in 4 vCPUs per pCPU for fairness. Table 5 shows for each scenario how the clustering system has organised vCPUs most of the time during the experiment. Fig. 6 left presents the performance of each application for each scenario. We can see that, except $LoLCF$ and $LLCO$ applications (which are quantum length agnostic), our prototype outperforms the default Xen scheduler (up to 20% of improvement).
Evaluation on a 4-socket machine.
We also evaluated our prototype by experimenting the complex use case presented in Section 3.5. The experimental machine for this experiment is an Intel Xeon Processor E5-4603 (composed of 4 sockets). For this specific experiment, we relied on micro benchmarks in order to exactly simulate the application behaviour presented in Section 3.5 (e.g. $IOInt^+$). Evaluation results are presented in Fig. 6 right. Remember that the clusters generated in this scenario are presented in Fig 3. As noted above, the worse performance obtained with our prototype is the same as with native Xen. Let us focus on $LLCF$ performance, which is not the same in clusters C_3^{90}, C_4^{90} and C_6^{30}. Since C_6^{30} uses the default quantum length, $LLCF$ performance in that cluster is the lowest. Concerning $LLCF$ in C_3^{90}, they share the LLC of their socket with $IOInt^+$ vCPUs, which are disturbers. This explains the lower performance of $LLCF$ in C_3^{90} in comparison with C_4^{90} (which does not host any disturber). Furthermore, this also shows the benefits of the clustering system.
Quantum length customization benefit.
The previous section showed the benefits of the whole system. The latter relies on two main phases: clustering and quantum length customization. We underlined the benefits of the clustering step in the previous section (see the comments about $LLCF$'s performance in clusters C_3^{90} and C_4^{90}). Let us focus now on the benefits of the quantum length customization step. To do so, we replayed the previous experiment, but the quantum length customization step was discarded. We experimented with three quantum lengths: small (1ms), medium (30ms) and large (90ms). Fig 7 presents the obtained results. Results are normalized over the performance when the customization and clustering steps are both activated: a performance graph bar above the normal value means that the activation of the quantum length customization step has improved applications' performance (higher is better). We can see that this is true for almost all application types. Obviously, some applications which run in the default cluster (C_6^{30}, the default Xen quantum length is used) do not have their best performance. This is the case for $ConSpin^-$ and $LLCF$ which perform better with a small and a large

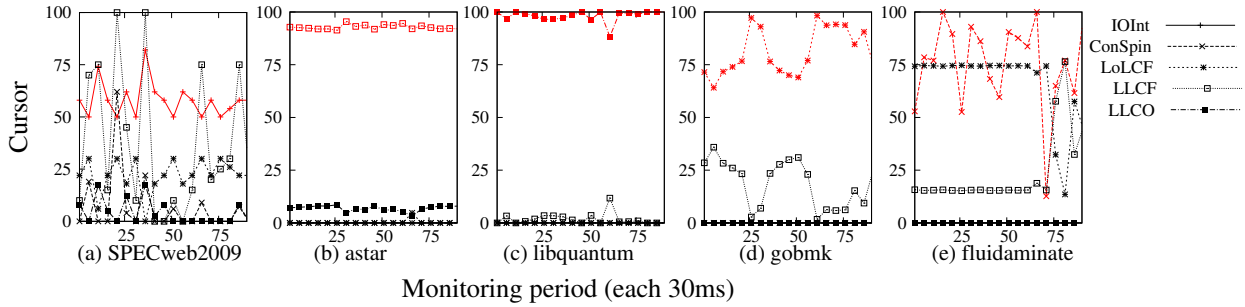

Figure 4. Online vTRS in action: This figure shows for 5 representative applications, 50 collected decision metric values $*\_cur\_avg$ used by vTRS for inferring the vCPU type. We define an application type as the type having its curve higher than the others most of the time (this is represented in the figure by the red lines).

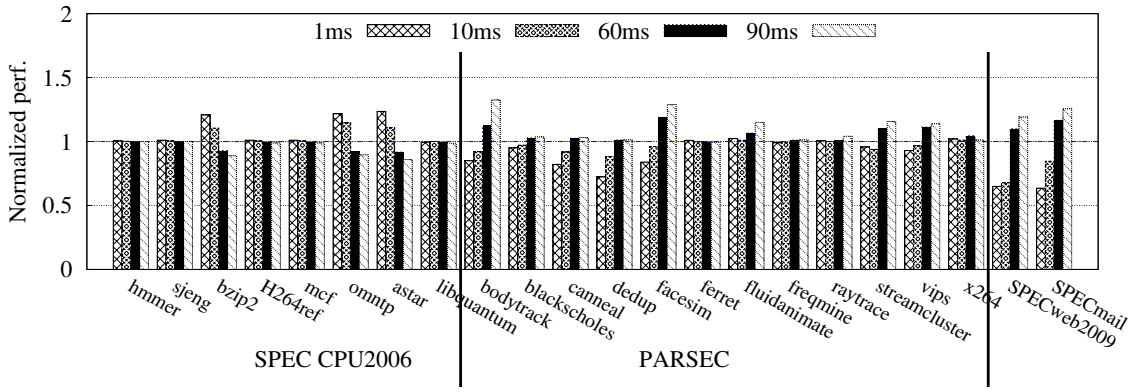

Figure 5. Validation of vTRS effectiveness and calibration results robustness. These results are normalized over the performance when running with the default Xen scheduler. We can see that each application obtains its best performance when the quantum length corresponding to the type identified by vTRS is used.

Scenarios	Colocated types	Applications
S_1	$5ConSpin, 5LLCF$	fluidanimate, $6LoLCF$ bzip2, hmmer
S_2	$5IOInt, 5LLCF, 6LLCO$	SpecWeb2009, bzip2, libquantum
S_3	$5LLCF, 5LLCO, 6LoLCF$	bzip2, libquantum, hmmer
S_4	$4IOInt, 4ConSpin, 4LLCF, 4LLCO$	SPECweb2009, facesim, bzip2, hmmer
S_5	$4IOInt, 4ConSpin, 4LLCF, 2LLCO, 2LoLCF$	SPECweb2009, facesim, bzip2, hmmer, libquantum

Table 4. Colocation scenarios.

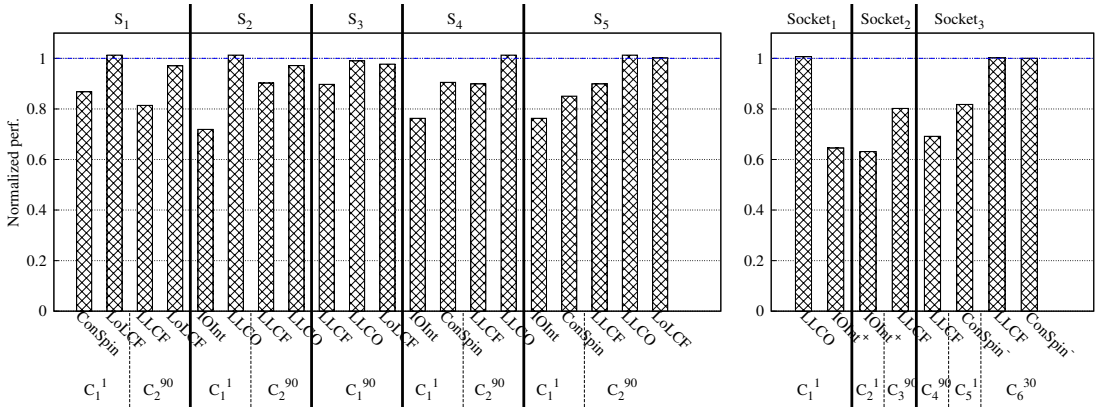

Figure 6. The effectiveness of the prototype: We experimented both a single-socket (left) and a multi-socket (right) machine. The results presented in this figure are normalized over the performance of each application when it runs with the default Xen scheduler. We can see that except $LoLCF$ and $LLCO$ applications (which are quantum length agnostic), our prototype outperforms the default Xen scheduler (up to 20% of improvement).

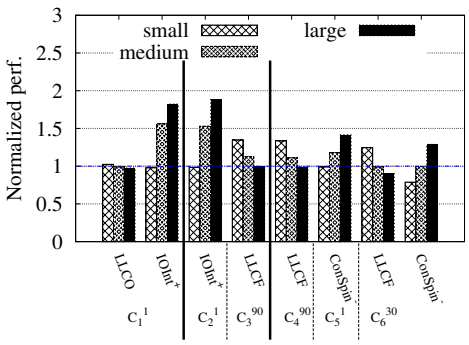

Figure 7. The benefit of quantum length customization: This figure shows that even if the clustering step improves applications' performance, performance can be further improved by quantum length customization. The results are normalized over the performance when both clustering and quantum length customization steps are activated.

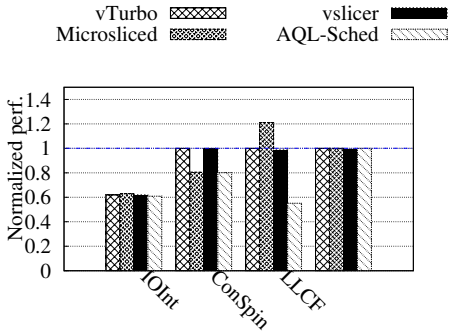

Figure 8. Comparison with other systems: This figure presents the comparison results of our prototype (AQL_Sched) with vTurbo, vSlicer, and Microsliced. These results are normalized over the performance obtained with the default Xen scheduler.

quantum length respectively. We can also notice that the small quantum length has good results as AQL_Sched for most application types, except $LLCF$ applications.

Comparison with other approaches.
We compare our solution with 3 other solutions:

- vTurbo [14]: it dedicates a pool of pCPUs for scheduling $IOInt$ vCPUs, using a lower quantum length (see Section 1).
- vSlicer [15]: it uses a lower quantum length for scheduling $IOInt$ vCPUs. In comparison with vTurbo, vSlicer does not dedicate a pCPU pool for the exclusive scheduling of $IOInt$ vCPUs.
- Microsliced [6]: it uses a lower quantum length for scheduling all vCPU types.

These solutions do not implement any online vTRS. Therefore, we manually configured each solution in order to obtain its best performance. We decided to use a 1 ms quantum length for both vTurbo and Microsliced solutions. Evaluations are based on scenario S_5 described in Table 3. Fig. 8 presents evaluation results. The latter are normalized over the performance obtained with the default Xen scheduler. We can see that our prototype provides in the worse case the same performance as other solutions. In summary, none of the other solutions provides the best performance for all application types. AQL_Sched is the first algorithm which adapts the quantum length to the behavior of the application, thus outperforming existing solutions.

4.3 Measuring AQL_Sched's overhead

Monitoring systems. IO request monitoring is accomplished by analyzing event channels. This task does not incur any overhead since the required mechanisms already exists in the hypervisor. Regarding the monitoring systems which rely on hardware counters, we did not observe any overhead, as also reported by [26].

Recognition and clustering systems. The complexity of both systems is $\mathcal{O}(max(m,n))$ where m and n are respectively the number of processors and the number of vCPUs. Knowing that both values are in the range of hundreds (we are in the context of virtualized systems, not native systems with thousands of tasks), $\mathcal{O}(max(m,n))$ is negligible. Regarding the overhead that can be induced by vCPU migration across pools, we avoid it with some implementation tricks. In Xen, a CPU pool is represented by a single data structure shared among schedulers of the same pool. Therefore, a vCPU migration between different CPU pools requires a data structure migration. This data structure migration can generate an overhead. This implementation is justified in Xen because Xen allows the use of different schedulers (Credit, SEDF) at the same time. In our case, only one scheduler is used (Credit). Therefore, we use the same data structure for all CPU pools. By doing so, no data structure copy is required when a vCPU migration is performed, thus avoiding any overhead due to migration.

The entire prototype. Putting all components together, the overhead of the entire prototype is negligible. We did not observe any degradation above 1%.

5. Related work

Many research studies have investigated scheduling in virtualized systems for improving application performance. Most of them focused on a specific aspect: interrupt handling (for IO intensive applications), lock holder preemption (for concurrent applications), prioritization (for real time applications) and LLC cache contention.

Interrupt handling. There were many prior efforts [6, 13–15, 33, 36] to improve latency sensitive applications in

Solutions/Features	Dynamic application type recognition	Handled application types	Overhead	Hardware modification required
vTurbo	Not supported	IO	No overhead	no
vslicer	Not supported	IO	No overhead	no
Microsliced	Not supported	IO, spin-lock	Overhead for CPU burn applications	yes
Xen BOOST	supported	IO	No overhead	no
AQL_sched	supported	IO, spin-lock, CPU burn	No overhead	no

Table 6. AQL_Sched compared with exiting solutions.

virtualized environments. Xen introduced a boosting mechanism which preempts the current running vCPU to quickly handle IO requests. However, this solution is not efficient for heterogeneous workloads (see Section 3.4). [15] presents vSlicer, a scheduler which uses a different quantum length (a lower value) for scheduling vCPUs which perform IO requests. In the same vein, [14] presents vTurbo, a solution which dedicates one or several pCPUs for scheduling IO intensive vCPUs using a lower quantum length. [6] proposed to shorten the quantum length of all vCPUs, thus improving both IO intensive and concurrent application performance. In order to reduce the impact of this solution on LLC sensitive applications, [6] introduced a new hardware design for minimizing LLC contention.

Lock holder preemption. This is a well known issue in virtualized environment [36–39]. The commonly used approach to address it, is coscheduling: vCPUs of the same VM are always scheduled in or out at the same time. This solution is limited because vCPUs do not always need the processor at the same time. [36] proposed an adaptive dynamic coscheduling solution where the scheduler dynamically detects VMs having long waiting time spin-locks. Only these VM vCPUs are coscheduled. [39] introduced Preemptable Ticket Spin-lock as a new locking primitive for virtualized environments in order to address the problem of lock waiter preemption. The latter improves the performance of traditional ticket spin-locks by allowing the preemption of unresponsive thread waiter. [6] demonstrates that using a shorter time slice is the simplest solution for addressing the lock holder preemption problem. Our work confirms that conclusion.

Prioritization. Several schedulers [40–44] were designed for real time applications in virtualized environments. For instance, [42] adds a new priority called *RealT* in Xen Credit scheduler for considering real-time guests. They are inserted at the first position in the run queue. [43, 44] proposed a similar solution. Our work does not consider real-time applications because our research context is cloud environments where fairness has to be ensured.

LLC cache contention. Several previous works proposed cache aware scheduling algorithms to address the LLC contention issue. In the context of non-virtualized environments, [31, 50, 51] presented some methods for evaluating the sensitivity and aggressiveness of an application. [30] proposed ATOM (Adaptive Thread-to-Core Mapper), a heuristic for finding the optimal mapping between a set of processes and cores such that the effect of cache contention is minimized. [52] is situated in the same vein. It proposed two scheduling algorithms for distributing processes across different cores such that the miss rate is fairly distributed. [53] presented a cache aware scheduling algorithm which awards more processing time to a process when it suffers from cache contention. Several researches [54, 55] addressed the problem of LLC contention in virtualized environments. [54] studied the effects of collocating different VM types under various VM to processor placement schemes for discovering the best placement. [55] proposed a cache aware VM consolidation algorithm which computes a consolidation plan so that the overall LLC misses are minimized in the IaaS.

Positioning of our work. Table 6 summarizes the comparison of existing solutions with our solution (AQL_sched). We can see that existing solutions have the following limitations: (1) they only address a specific issue; (2) vCPUs need to be manually typed (which is not realistic as a vCPU type may change); (3) some of them require the modification of the hardware (making them not yet usable). Our solution smartly addresses all these issues while being applicable in both today's virtualized systems and hardware.

6. Conclusion

This article presented AQL_Sched, the first VM scheduler which dynamically adapts the quantum length according to the application behavior in a multi-core platform. To this end, AQL_Sched dynamically associates an application type with each vCPU and uses the best quantum length to schedule vCPUs of the same type. We identified 5 main application types and experimentally found their best quantum length. By using a two-level clustering algorithm, our solution takes into account the LLC contention issue. We implemented our solution in Xen and we showed its effectiveness by experimenting with several reference benchmarks (SPECweb2009, SPECmail2009, SPEC CPU2006, and PARSEC). We compared our solution with the default Xen Credit scheduler, vSlicer, vTurbo and Microsliced and we obtained an improvement of up to 25%.

Acknowledgements

We sincerely thank the anonymous reviewers for their feedback. This work benefited from the support of the French "Fonds national pour la Société Numérique" (FSN) through the OpenCloudware project.

References

[1] Customer Success. Powered by the AWS Cloud. https://aws.amazon.com/solutions/case-studies/, consulted on September 2015

[2] reddit. https://www.reddit.com/, consulted on September 2015

[3] illumina. https://www.illumina.com/, consulted on September 2015

[4] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. Antonio Nicolo, "The impact of operating system scheduling policies and synchronization methods of performance of parallel applications", SIGMETRICS'91.

[5] Francesc Gin, Francesc Solsona, Porfidio Hernndez, and Emilio Luque, "Adjusting the Lengths of Time Slices when Scheduling PVM Jobs with High Memory Requirements", European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2002.

[6] Jeongseob Ahn, Chang Hyun Park, and Jaehyuk Huh, "Micro-sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems", MICRO, 2014.

[7] Luiz Andr Barroso and Urs Hlzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines", 2nd ed. Morgan & Claypool Publishers, 2013.

[8] Jacob Leverich and Christos Kozyrakis, "Reconciling High Server Utilization and Sub-millisecond Quality-of-Service", EuroSys, 2014.

[9] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis, "Heracles: Improving Resource Efficiency at Scale", ISCA, 2015.

[10] Credit Scheduler. http://wiki.xen.org/wiki/Credit_Scheduler, consulted on September 2015

[11] Xen. http://www.xenproject.org/, consulted on September 2015

[12] The CPU Scheduler in VMware vSphere 5.1. https://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf.

[13] Diego Ongaro, Alan L. Cox, and Scott Rixner, "Scheduling I/O in Virtual Machine Monitors", VEE, 2008.

[14] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu, "vTurbo: accelerating virtual machine I/O processing using designated turbo-sliced core", USENIX ATC, 2013.

[15] Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu, "vSlicer: latency-aware virtual machine scheduling via differentiated-frequency CPU slicing", HPDC, 2012.

[16] SPECweb2009. https://www.spec.org/web2009/, consulted on September 2015

[17] SPECmail2009. https://www.spec.org/mail2009/press/release.html, consulted on September 2015

[18] SPEC CPU2006. https://www.spec.org/cpu2006/, consulted on September 2015

[19] SPEC. https://www.spec.org/benchmarks.html, consulted on September 2015

[20] PARSEC. http://parsec.cs.princeton.edu/, consulted on September 2015

[21] EOLAS. http://www.businessdecision-eolas.com/, consulted on September 2015

[22] Roboconf. http://roboconf.net/en/index.html, consulted on September 2015

[23] clif. http://clif.ow2.org/, consulted on September 2015

[24] Wordpress. https://wordpress.org/plugins/benchmark/, consulted on September 2015

[25] Kernbench. http://ck.kolivas.org/apps/kernbench/kernbench-0.50/, consulted on September 2015

[26] CERN openlab publishes a report on the overhead of profiling using PMU hardware counters. http://openlab.web.cern.ch/news/cern-openlab-publishes-report-overhead-profiling-using-pmu-hardware-counters. July 2014.

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

[28] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat, 'Comparison of the Three CPU Schedulers in Xen', SIGMETRICS, 2007.

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

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

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

[32] Jinho Hwang and Timothy Wood. Adaptive dynamic priority scheduling for virtual desktop infrastructures. In *Proceedings of the 2012 IEEE 20th International Workshop on Quality of Service*, IWQoS '12, pages 16:1–16:9, Piscataway, NJ, USA, 2012. IEEE Press.

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

[36] XiaoBo Ding, Harbin, Zhong Ma, XingFa Da, 'Dynamic time slice of credit scheduler', ICIA 2014

[35] Nagakishore Jammula, Moinuddin Qureshi, Ada Gavrilovska, Jongman Kim, 'Balancing Context Switch Penalty and Response Time with Elastic Time Slice', HiPC 2014

[36] Chuliang Weng, Qian Liu, Lei Yu, Minglu Li, 'Dynamic adaptive scheduling for virtual machines' HPDC 2011

[37] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, Uwe Dannowski, 'Towards scalable multiprocessor virtual machines' VM 2004

[38] Wells, P.M, Chakraborty, K.Sohi, G,S, 'Hardware Support for Spin Management in Overcommitted Virtual Machines' PACT 2006

[39] Jiannan Ouyang, John R. Lange, 'Preemptable Ticket Spinlock: Improving Consolidated Performance in the Cloud', VEE 2013

[40] Sisu Xi, Wilson, J., Chenyang Lu, Gill C., 'RT-Xen: Towards real-time hypervisor scheduling in Xen', EMSOFT 2011

[41] Sisu Xi, Meng Xu, Chenyang Lu, Linh T. X. Phan, Christopher Gill, Oleg Sokolsky, Insup Lee, 'Real-Time Multi-Core Virtual Machine Scheduling in Xen' EMSOFT 2014

[42] Min Lee Georgia, A. S. Krishnakumar, P. Krishnan, Navjot Singh, Shalini Yajnik 'Supporting Soft Real-Time Tasks in the Xen Hypervisor' VEE 2010

[43] Peijie Yu, Mingyuan Xia, Qian Lin, Min Zhu, Shang Gao, 'Real-time Enhancement for Xen hypervisor', EUC 2010

[44] Seehwan Yoo, Kuen-Hwan Kwak, Jae-Hyun Jo, Chuck Yoo, 'Toward under-millisecond I/O latency in xen-arm' APsys 2011

[45] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Mark Williamson, 'Safe hardware access with the Xen virtual machine monitor', OASIS 2004

[46] KVM. http://www.linux-kvm.org, consulted on September 2015

[47] OpenVZ. https://openvz.org, consulted on September 2015

[48] VMware. www.vmware.com, consulted on September 2015

[49] Hyper-V. www.microsoft.com/Virtualisation, consulted on September 2015

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

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

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

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

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

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