2016 IEEE International Parallel and Distributed Processing Symposium

Dynamic Acceleration of Parallel Applications in Cloud Platforms by Adaptive Time-Slice Control

Song Wu*, Zhenjiang Xie*, Haibao Chen*, Sheng Di†, Xinyu Zhao* and Hai Jin*
* Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology
Wuhan, 430074, China
Email: {wusong, xiezhenjiang, chenhaibao, zhaoxinyu512, hjin}@hust.edu.cn
† Argonne National Laboratory, USA
Email: sdi1@anl.gov

***Abstract*—Tightly-coupled parallel applications in cloud systems may suffer from significant performance degradation because of the resource over-commitment issue. In this paper, we propose a dynamic approach based on the adaptive control over time-slice for virtual clusters, in order to mitigate the performance degradation for parallel applications in cloud and avoid the negative impact effectively on other non-parallel applications meanwhile. The key idea is to reduce the synchronization overhead inside and across *virtual machines* (VMs) in cloud systems, by dynamically adjusting the time-slices of VMs in terms of the spinlock latency at runtime. Such a design is motivated by our experimental finding that VM's time slice is a key factor determining the synchronization overhead as well as the parallel execution performance. We perform the evaluation on a real cluster environment deployed with XEN, using five well-known benchmarks with 10+ applications. Experiments show that our approach obtains 1.5-10× performance gain for running parallel applications, than other state-of-the-art solutions (including Credit Scheduling of Xen and the well-known methods like Co-Scheduling and Balance Scheduling), with nearly unaffected impact on the performance of non-parallel applications.**

I. Introduction

With the ever-increasing demand of *high performance computing* (HPC) power, virtual clusters in cloud platforms have been explored to help run tightly-coupled parallel applications [1], because of their high flexibility and cost-effectiveness. Virtual clusters are constructed using a number of *virtual machines* (VMs) in a cloud system based on user's demand on resources. The parallel applications running in virtual clusters, however, may suffer from huge performance degradation due to the inevitable synchronization overhead [2]. What is even worse is that the cloud providers often provide much more executable Virtual CPUs (VCPUs) than available Physical CPUs (PCPUs) on purpose [3], in order to maximize the profit. Such an over-commitment situation will aggravate the performance degradation issue when running tightly-coupled parallel applications in cloud.

There are already some existing strategies proposed for mitigating the synchronization overhead, including balance scheduling [4], dynamic switching-frequency scheduling [5], and co-scheduling [6, 7]. Such approaches, however, are mainly focused on multi-threaded applications under the *single symmetric multiprocessing* (SMP) virtual machine (VM) model. They are not suitable for a large-scale cloud platform, in which many parallel applications would be hosted by multiple virtual clusters across physical machines.

The performance issue induced by the synchronization operations inside a virtual cluster is very challenging. Firstly, there are many different parallel models designed for various scientific problems. It is hard to estimate beforehand how much communication and computation a parallel application would require at runtime. Hence, it is impossible to optimize the run-time parallel execution performance based on the static analysis of application's particular characteristics. Secondly, it is hard to characterize the mutual performance influence among the parallel applications running in virtualized clusters across different physical machines. Thirdly, it is non-trivial to avoid the performance impact on non-parallel applications when running parallel applications in a multi-tenant cloud, due to their substantially different running characteristics. Finally, implementation and improvement of virtual clusters in cloud requires the in-depth understanding of *virtual machine monitor* (VMM) as well as the related fundamental technologies.

In this paper, we carefully investigate the critical issue of how to mitigate/avoid the performance degradation when running parallel applications with many synchronization operations in virtual clusters. Based on our careful experiments with well-known benchmarks running over a real cluster, we observe that shortening time slices of VMs is able to improve the parallel execution performance significantly. The key reason is that shortening the time slice of VM can effectively mitigate the spinlock latency, which is a critical factor (as indicated by [4]) that determines the application performance. Accordingly, we propose an *Adaptive Time-slice Control* (*ATC*) model, under which the time slice of each VM is adjusted properly at run-time, such that the parallel execution performance can be improved a lot.

The main contributions are listed as follows:

- We carefully analyze the execution of parallel applications in virtual clusters, and discover that short time slices of VMs usually lead to short spinlock latencies for parallel applications in a cloud environment. This will further lead to relatively low synchronization overhead (both in VM and across VMs), and thus the performance of parallel applications will be improved.

- We devise an adaptive VMM time-slice control model that can adaptively adjust the time slice of VMs according to their spinlock latency at runtime. We characterize the overhead of short time slices and exploit a minimum time slice threshold to prevent over-shortened time slices.
- We implement a prototype based on XEN and Linux, and evaluate the solution by running 5 benchmarks with 10+ different types of applications over a real cluster. Experiments show that our solution can achieve 1.5-10× performance gain for the parallel applications, compared with XEN's Credit Scheduler and many other state-of-art approaches. Moreover, experiments show that our solution incurs little impact on the performance of non-parallel applications in cloud system.

II. Background and Motivation

A. Research Background

In what follows, we present the performance degradation of the existing approaches with increasing scale of parallel applications on virtual clusters and the impact of the parallel execution performance on the non-parallel applications.

1) Performance Degradation of Parallel Applications with Existing Scheduling Approaches: Although cloud platforms can effectively run tightly-coupled parallel applications, such applications still suffer from performance degradation because of the inevitable synchronization overhead [2], which is composed of two parts: the synchronization overhead inside VMs and the synchronization overhead across VMs.

The key reason the existing scheduling approaches (such as hybrid scheduling [6], dynamic Co-scheduling [7] and Balance Scheduling [4]) are not suitable for parallel applications in virtual clusters is that they are focused only on concurrent workload processing (i.e., multi-threaded applications) within an SMP VM. Their design objectives are to reduce the synchronization overhead caused by *lock holder preemption* (LHP) [8], instead of that of parallel applications across VMs in a virtual cluster. In fact, asynchronous schedule of VMs inside a virtual cluster will suffer huge synchronization overheads across VMs for tightly-coupled parallel applications.

Figure 1 illustrates the performance degradation caused by the synchronization overheads of Co-Scheduling (CS) method [7] (a well-known method of reducing synchronization overhead for parallel applications). There are 32 physical nodes adopted in the experiment (more detailed experimental settings like the size, number, placement of virtual clusters, can be found in Section IV-B). We compare the normalized execution time of the Co-Scheduling (CS) and XEN's Credit scheduler (CR) by using the following 6 parallel applications, *sp*, *bt*, *cg*, *is*, *mg*, and *lu*, which are all from *NASA Parallel Benchmark* (NPB) [9]. *Normalized execution time* refers to the ratio of the execution time to that of the approach CR. We just present the result of *lu* as a typical example to show the scalability issue in Figure 1 due to space limit. More experimental results can be found in Section IV-B.

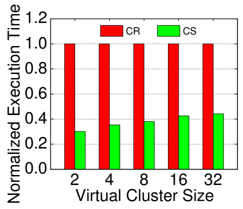


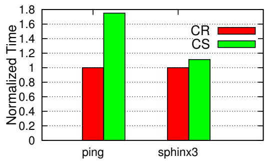
Fig. 1. Performance of CR & CS when running *lu* on 2, 4, 8, 16, 32 VMs

Through Figure 1, it is clearly observed that the execution time of parallel application *lu* under CS will increase, as the virtual cluster size (the number of VMs) increases. In particular, under CS, its normalized execution time in a virtual cluster is 0.3 when the virtual cluster consists of only two 8-VCPU VMs, while the normalized execution time increases up to 0.44 if we set the virtual cluster size to 32 8-VCPU VMs. Apparently, compared with CR, CS lacks scalability when running parallel applications on virtual clusters.

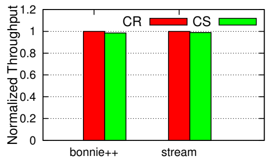
2) Performance Impact on Non-parallel Applications with Existing Scheduling Approaches: Most of the existing approaches reduce the synchronization overhead by simply promoting the scheduling priorities of the VMs involved, which may influence the execution of other non-parallel applications with latency-sensitive operations (such as request-response operations of web applications), as indicated by the experimental results shown in Figure 2. In this experiment, we run some parallel applications from the NPB benchmark and some non-parallel applications simultaneously in the same virtualized cloud system. Two physical nodes are used, and each is deployed with four 8-VCPU VMs, so there are totally eight 8-VCPU VMs. Three virtual clusters are constructed, and each consists of two 8-VCPU VMs from the two physical nodes respectively. The remaining two VMs are used to run non-parallel applications as shown below:

- *bonnie++*: bonnie++ is a benchmark that is aimed at the performance of hard drive and file systems.
- *sphinx3*: sphinx3 is a CPU-intensive application from SPEC CPU 2006.
- *stream*: stream is memory-intensive benchmark to measure the sustainable bandwidth of memory.
- *ping*: ping is a latency-sensitive application designed for measuring the round trip time.

Figure 2 clearly shows that CS does lead to the performance degradation for some types of non-parallel applications. The ping latency under CS, for instance, is 1.75 times longer than that of CR, the normalized execution time of *sphinx3* is 1.11 times longer than that of CR, and the throughput of stream under CS is slightly lower than that of CR. The reason for the experimental results is that co-scheduling promotes the scheduling priorities of the VMs running parallel applications, leading to the performance degradation of latency-sensitive applications (such as *ping*). Moreover, the additional context switches under the co-scheduling method would cause more cache flushes, introducing the negative impact on the applications like *sphinx3* and *stream*.



(a) *ping* and *sphinx3*



(b) *bonnie++* and *stream*

Fig. 2. Normalized performance of different applications under CR and CS

B. Design Motivation

Before exploring an effective solution to mitigate the parallel performance degradation, it is necessary to have an in-depth understanding of its root cause, i.e., synchronization overhead, and how it would impact the execution performance. In what follows, we analyze the synchronization overhead of parallel applications in virtual clusters, which is a fundamental basis of our design.

As mentioned previously, when a parallel application (e.g., an MPI program) runs in a virtual cluster composed of multiple VMs, the synchronization operations can be split into two types: (1) the synchronization among the processes inside a VM via shared memory; and (2) the synchronization among the processes across VMs by network communications. We will discuss them in details and explain why short time slices generally lead to low synchronization overhead.

1) Synchronization Overhead inside a VM: Prior research has identified that the synchronization overhead in SMP VM is mainly caused by LHP [8]. Figure 3 illustrates such overhead by using a simple scheduling example. In this figure, VCPU0 and VCPU1 belong to the same VM running parallel processes with mutual synchronization on different PCPUs. The time slices marked with 'X' are occupied by other VMs. When lock-holder VCPU0 (on PCPU0) is preempted by certain VCPU of another VM, the lock-waiter VCPU1 (on PCPU1) will keep spinning the lock due to the synchronization operation of the application and the preemption of lock-holder VCPU0. The lock-waiter VCPU1 cannot acquire the synchronization lock until VCPU0 is rescheduled with its lock released. The spinlock latency of VCPU1 here is $3L_{TS}$, where L_{TS} denotes the length of time slice. Intuitively, shortening the time slice of VMs (no matter what types of applications they host) may reduce the spinlock latency of VCPU1.

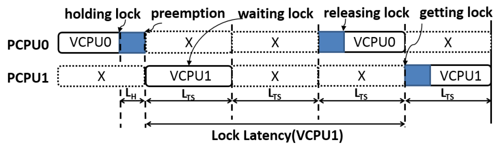


Fig. 3. The lock latency of VCPU1 when LHP occurs, where VCPU0 and VCPU1 belong to the same VM, and run parallel processes with synchronization operations

2) Synchronization Overhead across VMs: The synchronization across VMs is through network communications. In this section, we take Figure 4 as an example to illustrate the synchronization overhead across VMs, where VM1 and VM2 belong to a virtual cluster and run parallel processes with synchronization operations.

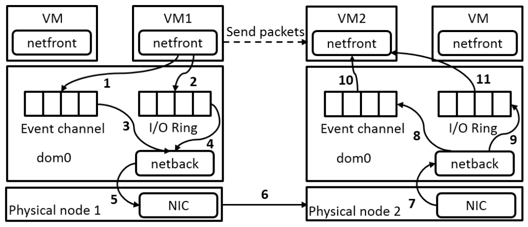


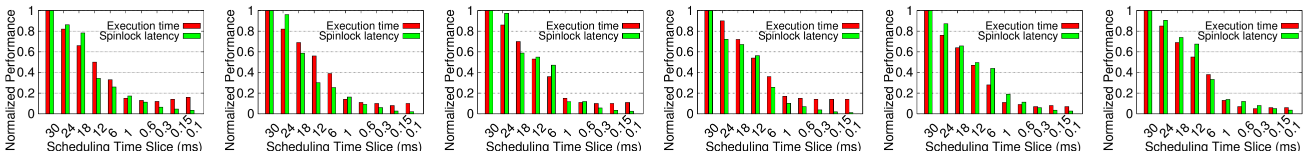
Fig. 4. An example of synchronization overhead between two VMs (VM1 and VM2) running parallel processes with synchronization operations

As shown in Figure 4, it takes 11 steps for VM1 to send network packets to VM2. The detailed process is presented as follows, with highlighted overhead sources.

In the first place, VM1 waits for being scheduled by VMM (**overhead source 1**). As VM1 is scheduled to PCPUs by VMM, it will notify dom0 using an event channel (step 1) and put the network packets in I/O ring (step 2). After that, the dom0 of physical node 1 needs to wait for being scheduled by VMM (**overhead source 2**). As the dom0 of physical node 1 is scheduled to PCPUs, it will get an event notification regarding VM1 sent from event channel (step 3). Then, the dom0 of physical node 1 copies the network packets sent by VM1 to netback (step 4) and sends the network packets through *Network Interface Card* (NIC) on physical node 1 (step 5). The network packets will be transferred to VM2 hosted in physical node 2 via physical network (step 6). Before conducting step 7, the dom0 of physical node 2 waits for being scheduled by VMM (**overhead source 3**). When dom0 of physical node 2 is scheduled to PCPUs, it will copy the network packets from NIC card to netback (step 7). Then, dom0 will notify VM2 using the event channel (step 8) and copy the network packets to I/O ring (step 9). The process will not go to the step 10 until VM2 gets scheduled (**overhead source 4**). Once VM2 is scheduled onto some PCPUs, it will get a notification from the event channel (step 10) and copy network packets from I/O ring to netfront (step 11).

Through the above steps, we can see that there are mainly four synchronization overheads, which are all caused by VM scheduling of VMM. The overhead source 1, for instance, depends on the total length of time slice of VMs that wait ahead of VM1 in the PCPU run queue. Assume there are $N-1$ VMs ($VM_1, \cdots, VM_{N-1}$) waiting ahead of VM_N, then the synchronization overhead source 1 is $\sum_{i=1}^{N-1} TimeSlice_i$, where $TimeSlice_i$ is the length of time slice of VM_i. To minimize this synchronization overhead, the time slice of each VM that is ahead of VM_N is supposed to be set as short as possible. Therefore, it is viable to adjust the time slices of other VMs (no matter what types of applications they host) to achieve low synchronization overhead across the VMs running parallel applications.

In the well-known *Bulk Synchronous Parallel* (BSP) model [10], parallel application executes computation phases and synchronization phases alternatively. In synchronization phases, spinlock is commonly used for data sharing. The spinlock latency reflects how long it takes a VM to complete synchronization phases. Therefore, the spinlock latency has



(a) *lu* (b) *is* (c) *sp* (d) *bt* (e) *mg* (f) *cg*

Fig. 5. Performance when running *lu*, *is*, *sp*, *bt*, *mg*, or *cg* with different time slices

a significant influence on parallel application's performance, which is confirmed in [4].

In order to verify how time slice affects the spinlock latency and spinlock latency's influence on parallel application's performance, we conduct a set of experiments with two physical nodes, each of which consists of four 16-VCPU VMs. Four identical virtual clusters are constructed in the platform, and the four VMs on each physical node belong to them separately. We gradually shorten XEN's scheduling time slice from 30 ms to 6 ms with the decrement step of 6 ms, and also evaluate much shorter time slices (such as 1ms, 0.6ms, 0.3ms, 0.15ms, and 0.1ms). The experimental results are shown in Figure 5.

From Figure 5(a) through Figure 5(f), we can see that shortening the time slice of VMs is able to reduce spinlock latency effectively, and improve the performance of each parallel application significantly (even up to about $10\times$). Meanwhile, we can also observe that there exists a strong positive correlation (all pearson correlation coefficients are larger than 0.9) between the spinlock latency of applications and its performance under the settings of this experiment, which means that spinlock latency is probably a fairly effective indicator for identifying the parallel application performance.

Summary. Through the above analysis, we conclude that shortening time slice of VMs can reduce synchronization overhead of parallel applications running in virtual clusters effectively, improving the performance significantly.

III. Adaptive Time-slice Control

In this section, we propose a novel *adaptive time-slice control* (ATC) model that can dynamically adjust the time slices of VMs for improving the parallel execution performance in cloud systems. We will answer the following three questions:

- How to adaptively adjust the time slice of VMs that are running parallel applications in each scheduling period of VMM (such as 30ms of XEN's Credit scheduler), according to the monitoring information about average spinlock latency at runtime?
- When we adjust the time slice of parallel applications, how to prevent over-shortened time slice which may cause additional overhead?
- Since various types of applications coexist in real-world multi-tenant cloud platforms, how can we avoid the performance impact on other non-parallel applications?

A. Dynamically Adjusting Time Slice for VMs Running Parallel Applications

In this subsection, we describe how to compute the time slices of the VMs running parallel applications based on their average spinlock latency adaptively. In particular, if the average spinlock latency of a VM running parallel application exhibits an increasing trend, we will shorten its time slice in the coming schedules. If the average spinlock latency of the VM exhibits a decreasing trend, the time slice will be set based on its historical information and spinlock latency. The time slice computation process is performed at the beginning of each scheduling period of VMM based on the mean values of the spinlock latency and the time slice in the most recent three scheduling periods.

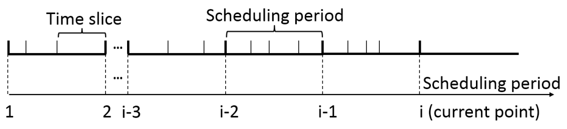


Fig. 6. Adjusting time slice length for VMs running parallel applications

As shown in Figure 6, at the end of a scheduling period (i.e., the $(i-1)^{th}$ scheduling period), the average spinlock latency of VM during the $(i-1)^{th}$ scheduling period will be used to determine the time slice of that VM in the next scheduling period, i.e., i^{th} scheduling period. Specifically, to determine the time slice of a VM in the i^{th} scheduling period, we use the historical information of last three scheduling periods to infer the changing trend of average spinlock latency, i.e., $(i-3)^{th}$, $(i-2)^{th}$, and $(i-1)^{th}$.

The pseudo-code of adjusting time slice for the VMs running parallel applications is shown in Algorithm 1. The key idea is to guarantee the parallel execution performance by adaptively controlling the time slices in terms of the spinlock latency information. The scheduling time slices of the VMs running parallel applications will be shortened when the spinlock latency is becoming longer. Specifically, if the average spinlock latency of a VM in the $(i-1)^{th}$ scheduling period is larger than that in the $(i-2)^{th}$ scheduling period (e.g., case 1 in Figure 7), the spinlock latency in the next period (i.e., the i^{th} scheduling period) will likely increase according to this trend. In order to reduce the spinlock latency in the i^{th} scheduling period, the time slice of this VM in the i^{th} scheduling period is supposed to be shorter than that in the $(i-1)^{th}$ scheduling period (lines 1-8 of Algorithm 1).

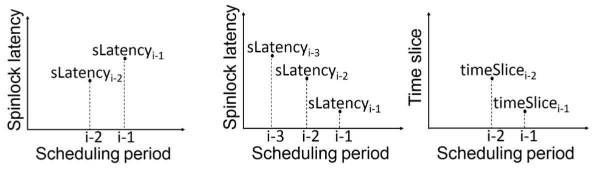
When the spinlock latency decreases, we will check whether it is caused by the decrease of time slice before adjusting the time-slice. If the average spinlock latency in the most recent three scheduling periods (i.e., $(i-3)^{th}$, $(i-2)^{th}$ and $(i-1)^{th}$ scheduling periods) keeps decreasing, and the time slice of this VM in the $(i-2)^{th}$ scheduling period is longer than that in the $(i-1)^{th}$ scheduling period (see the case 2 in Figure 7), we can infer that the smaller average spinlock latency is likely due to the shorter time slices. Accordingly, we will use a shorter time slice (e.g., decrease by 1ms) for this VM in the i^{th} scheduling period (lines 1-8 of Algorithm 1). Otherwise, the time slice of the VM will stay the same as the last scheduling period (lines 9-10 of Algorithm 1). If the average spinlock latency of a VM remains zero in the last three scheduling periods, indicating that the VM is performing less synchronization operations, then the time slice of this VM will be increased by a tiny increment (e.g., 1ms) (lines 12-20 of Algorithm 1) to reduce the possible overhead of short time slices. (see details in Section III-B).

Algorithm 1 Computing time slice for a VM running parallel applications

Input: 1) The historical information (e.g., average spinlock latency: $sLatency$ and time slice: $timeSlice$) of the VM in last three scheduling periods of VMM (i.e., $(i-3)^{th}$, $(i-2)^{th}$ and $(i-1)^{th}$); 2) The minimum time slice threshold $minThreshold$ of parallel applications (see details in Section III-B); 3) α and β, which are different granularities of time-slice adjustment, and the former is larger than the latter.

Output: The time slice of a VM will be used in the i^{th} scheduling period.

```
1: if {$sLatency_{(i-2)} < sLatency_{(i-1)}$} or {($sLatency_{(i-3)} > sLatency_{(i-2)} > sLatency_{(i-1)}$ and $timeSlice_{(i-2)} > timeSlice_{(i-1)}$)} then
2:     if $timeSlice_{(i-1)} > \alpha$ and $timeSlice_{(i-1)} - \alpha \geq minThreshold$ then
3:         $timeSlice_i = timeSlice_{(i-1)} - \alpha$
4:     else if $timeSlice_{(i-1)} > \beta$ and $timeSlice_{(i-1)} < \alpha$ and $timeSlice_{(i-1)} - \beta \geq minThreshold$ then
5:         $timeSlice_i = timeSlice_{(i-1)} - \beta$
6:     else
7:         $timeSlice_i = timeSlice_{(i-1)}$
8:     end if
9: else
10:     $timeSlice_i = timeSlice_{(i-1)}$
11: end if
12: if the spinlock latency remains zero in last three scheduling periods then
13:     if $timeSlice_{(i-1)} >$ DEFAULT $- \alpha$ then
14:         $timeSlice_i$ = DEFAULT
15:     else if $timeSlice_{(i-1)} > \alpha$ and $timeSlice_{(i-1)} <$ DEFAULT $- \alpha$ then
16:         $timeSlice_i = timeSlice_{(i-1)} + \alpha$
17:     else
18:         $timeSlice_i = timeSlice_{(i-1)} + \beta$
19:     end if
20: end if
21: return $timeSlice_i$
```



(a) *case1* (b) *case2*

Fig. 7. Two cases when adjusting the time slices of VMs, where $sLatency_i$ and $timeSlice_i$ denote the average spinlock latency and the time slice of a VM in i^{th} scheduling period, respectively

B. Optimizing Time Slice Threshold for Parallel Applications

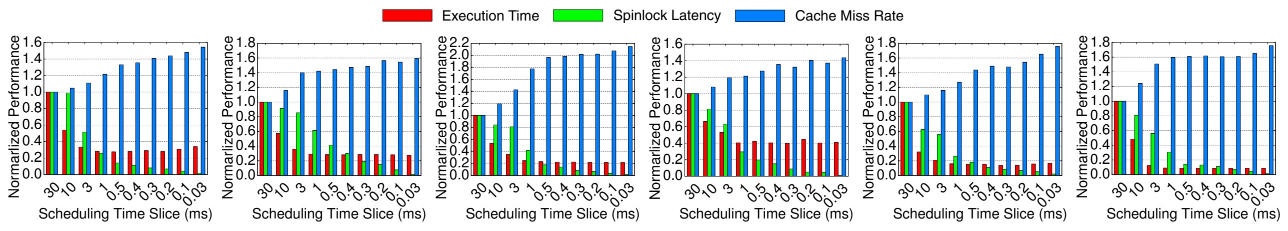
As mentioned previously, time slice is supposed to be set to a small value, however too short time slice will induce too frequent context switches with additional cache flushes [11], leading to low performance unexpectedly. Thus, time slice has to be optimized with both of the above two factors. In what follows, we discuss the overhead introduced by short time slices, and then we explore a bestfit threshold that can optimize the performance for all parallel applications.

In order to deeply understand the overhead introduced by short time slices, we conduct an experiment by gradually shortening time slice of parallel applications. We collect samples of *last-level cache* (LLC) misses by using *Xenoprof* [12], in order to measure cache flushes. The experimental settings are the same as we used in Section II-B1. In addition, we adopt four virtual clusters each of which consists of two VMs (from two physical nodes respectively). The parallel applications are all from NPB with class C, because they have long execution length, such that there are enough cache misses and spinlock latencies for us to study the overhead.

The experimental results are presented in Figure 8. From Figure 8(a) through Figure 8(f), we can observe that the execution time does not always decrease with decreasing time slice length (though the spinlock latency always keeps decreasing), because of the increasing cache miss rate. In Figure 8(a), for example, when the time slice is shorter than 0.2 ms, the performance of *lu.C* starts declining with decreasing time slice. This is because of too frequent context switches and many cache misses introduced, though the synchronization overhead can be mitigated more or less with short time slices. Thus, for a parallel application running in a virtual cluster, there must exist a performance inflection point (e.g., 0.2ms for the aforementioned *lu.C*), after which the performance improvement gained by spinlock latency will be totally canceled out by the cost of cache misses. Obviously, the best parallel execution performance can be represented by the performance inflection point, differing with applications.

Considering that the VMM is unaware of what exact parallel applications are running on the VMs, we exploit a uniform time slice threshold to achieve nearly optimal performance for all parallel applications, which is also for the sake of low computational complexity. In order to explore an optimized uniform time slice threshold, we adopt Euclidean metric [13] in Euclidean space to assess how close the performance of parallel applications under a given time slice is to the optimal performance each parallel application can achieve. Specifically, we use this metric in an n-dimensional Euclidean space. As shown in Equation (1), n is the number of applications, O_i represents the minimal normalized execution time of i^{th} application, while P_i stands for the normalized execution time of i^{th} application under a specified time slice. The closer $D(O,P)$ is to 0, the more optimized overall performance to be gained. The short time slices we used to calculate the Euclidean metric include 0.5ms, 0.4ms, 0.3ms, 0.2ms, 0.1ms and 0.03ms (shorter time slice intervals leads to tiny performance differences, so they are not presented in the figure). The Euclidean metrics under these short time slices are 0.034, 0.020, 0.018, 0.049, 0.039, 0.069, respectively. Obviously, the minimum Euclidean metric value is 0.018, thus the minimum time slice threshold is supposed to be set to 0.3ms.

$$D(O,P) = \sqrt{\sum_{i=1}^{n}(O_i - P_i)^2} \tag{1}$$

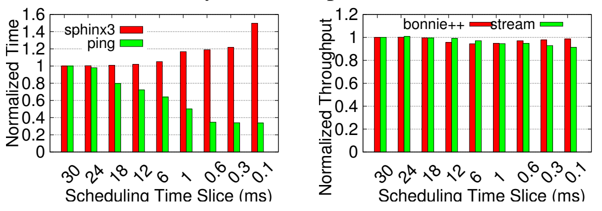


(a) *lu.C* (b) *is.C* (c) *sp.C* (d) *bt.C* (e) *mg.C* (f) *cg.C*

Fig. 8. The performance impact on parallel applications when adjusting the time slices of VMs

C. *Setting Time Slice for Non-Parallel Applications*

Various types of applications (e.g., parallel applications and non-parallel applications) from different users often coexist in the same virtualized cloud systems. In this section, we focus on how to avoid the performance impact of short time slices on non-parallel applications. We first conduct an experiment (using the same setting as in Section II-A2) to explore how time slice affects the non-parallel application's performance in the virtualized system, and then introduce our ATC algorithm, which can effectively solve the problem.



(a) *ping* and *sphinx3* (b) *bonnie++* and *stream*

Fig. 9. The performance impact on non-parallel applications when adjusting the time slices of VMs

As shown in Figure 9, the performance of *sphinx3* declines as the length of time slice decreases, because of the overhead of the additional context switches especially for CPU-intensive applications. For *ping*, the average round trip time decreases as time slice decreases, since more context switches offer more choices to deal with network packets. For *stream*, the memory bandwidth suffers from slight performance degradation as the time slice decreases, in that the additional context switches will result in more cache flushes.

In order to boost the performance of parallel applications, we devise the adaptive time slice control approach, which adjusts the lengths of time slices for the VMs running parallel applications and the VMs running non-parallel application separately. Specifically, we first calculate the time slice of each VM that is running parallel application by Algorithm 1. For the purpose of not introducing additional computational complexity and better ensuring fairness, we choose the minimum one among the calculated time slices for all VMs running parallel applications. In order to avoid the negative impact on the performance of non-parallel applications, we set the time slice of the VMs running non-parallel applications to the default value of XEN VMM. Besides, for the sake of higher flexibility, we design an interface that allows system administrators to specify the time slice of the VMs running non-parallel applications on demand.

Algorithm 2 Adjusting time slice for VM separately

```
Input: 1) the list of VMs in a physical node: $vmList$; 2) the type of vm
Output: the time slice for each VM
 1: $tempvmList = []$
 2: for each $vm$ in $vmList$ do
 3:     if the type of $vm$ is set to PARALLEL then
 4:         $vm.timeSlice$ = compute_timeSlice($vm$)/* call Algorithm 1 */
 5:         Add $vm$ into $tempvmList$
 6:     end if
 7: end for
 8: if $tempvmList$ is empty then /* There are no VMs that are running
    parallel application */
 9:     for each $vm$ in $vmList$ do
10:         $vm.timeSlice$ = DEFAULT
11:     end for
12: else
13:     $tempTimeSlice$ = min_timeSlice($vmList$)
14:     for each $vm$ in $vmList$ do
15:         if the $vm$ is running parallel applications then
16:             $vm.timeSlice = tempTimeSlice$
17:         else if system administrator specifies the time slice for VMs
    running non-parallel application then
18:             $vm.timeSlice$ = SPECIFIED
19:         else
20:             $vm.timeSlice$ = DEFAULT
21:         end if
22:     end for
23: end if
```

The pseudo-code is shown as Algorithm 2, and its input information is the list of VMs queried on a physical machine and the types of VMs. We use *tempvmList* to denote the list of VMs running parallel applications on the physical node (line 1). Then, the algorithm traverses all of the VMs. If a VM is running parallel application, `compute_timeSlice` is called to compute its time slice and this VM is added into *tempvmList* (lines 2-7). If there is no VM running parallel applications in the system, the time slice of each VM is set to the default value of VMM (lines 9-11). If there exists at least one VM running parallel application, the algorithm will perform the following two steps: (1) call function `min_timeSlice` to get the minimum time slice (denoted as *tempTimeSlice*) of all VMs running parallel applications (line 13); and (2) find all of the VMs that are running parallel applications and set their times slices to the minimum time slice *tempTimeSlice* (lines 15 and 16). If the VM is running non-parallel application and system administrator already assigned its time slice a value, the time slice of this VM is set as the specified value (lines 17 and 18). Otherwise, its time slice will be set to the default value (lines 19 and 20).

As depicted in Algorithm 2, the time slice of all VMs running parallel applications will be set to the minimum value from all of the time slices calculated by Algorithm 1 at the beginning of each scheduling period of VMM. For the VMs running non-parallel applications, the time slices will be determined on demand by the system administrator to set to the default values otherwise. Suppose there are N VMs, then the time complexity of our algorithm is O(N) because that of `compute_timeSlice` for each VM is O(1).

IV. Performance Evaluation

We describe the experimental setting in Section IV-A, and then present the experimental results thereafter.

A. Experimental Setting

(1) Experimental Platform. We adopt 32 nodes with totally 256 cores connected by a 1Gbps Ethernet. Each physical node is equipped with two Intel Xeon E5620 Quad-Core CPU and 24 GB RAM, and deployed with Xen-4.2.1. Each VM booted up on top of the VMM is running the Linux-3.9.3 kernel.

(2) Scheduling Approaches. We compare our ATC approach to well-known scheduling approaches, as listed below:

- **Credit** (CR). CR is the default scheduler implemented in XEN [14] VMM.
- **Co-scheduling** (CS). CS [7] dynamically sets the type of VM according to spinlock wait time, and co-schedules the VM as the spinlock wait time exceeds the minimum spinlock wait time threshold.
- **Balance Scheduling** (BS). BS [4] ensures at most one VCPU in a VM runs in the same PCPU run queue.
- **Dynamic Switching-frequency Scaling** (DSS). DSS [5] sets the length of time slice for each VM running concurrent workload according to their I/O behaviours.
- **vSlicer** (VS). VS [15] is a differentiated-frequency CPU microslicing approach to accelerate latency-sensitive applications, especially when latency-sensitive applications and latency-insensitive applications coexist.

(3) Classification of Experiments. Our experiment is divided into two categories. The first category is conducted with only parallel applications running in the system. There are two evaluation types in the first category. The evaluation type A runs the same parallel application in four virtual clusters with different scales (different numbers of physical nodes). For the evaluation type B, considering it is very common for cloud systems (e.g., Amazon EC2) to host various types of applications with different sizes, we synthesize the size and number of virtual clusters launched in cloud environment based on the job traces of a Linux cluster (Atlas) at Lawrence Livermore National Laboratory (LLNL) [16]. For the second category, we perform an experiment with a mixture of parallel and non-parallel applications, running parallel applications and other types of applications simultaneously.

(4) Benchmarks. The benchmarks used here are *NPB* [9], *Httperf* [17], *bonnie++* [18], *SPEC CPU 2006* [19], and *stream*. *NPB*, *bonnie++* and *stream* have been described in Section II. *Httperf* is for measuring the performance of web servers. *SPEC CPU 2006* is a CPU-intensive benchmark suite.

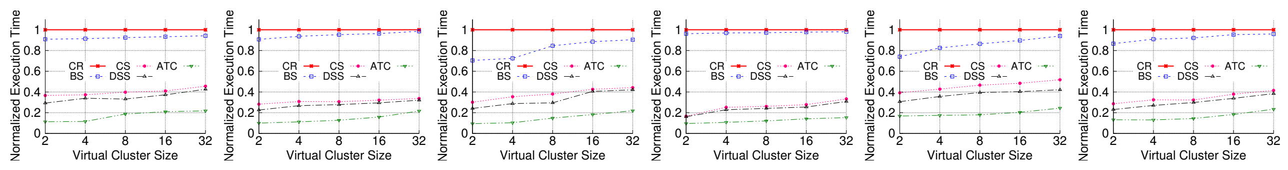
B. Scenarios with Parallel Applications

In this section, we adopt parallel applications (i.e., *lu*, *is*, *sp*, *bt*, *mg*, *cg*) from *NPB* benchmark. For the evaluation type A, we run the same parallel applications in all virtual clusters. Evaluation type B runs various parallel applications based on the job traces of a Linux cluster (Atlas) at LLNL [16].

1) Running the Same Parallel Applications: In this evaluation, we scale the number of physical nodes from 2 to 32 (2, 4, 8, 16, and 32), and four 8-VCPU VMs are booted up on each physical node. Four identical virtual clusters are constructed using all VMs in the platform, and the four VMs on each physical node belong to them separately. We run *lu* on these four virtual clusters simultaneously for ten times, and record the execution time of *lu* on each virtual cluster. The same test procedures also go to *sp*, *bt*, *cg*, *is* and *mg*, respectively.

Figure 10 shows the average execution times of *sp*, *bt*, *lu*, *cg*, *is* and *mg* running on virtual clusters with different solutions: BS, CS, DSS and ATC (we do not test VS in this experiment because it is designed for the scenario where latency-sensitive applications and latency-insensitive applications coexist, we will compare our ATC to VS in Section IV-C with various types of applications). The execution times are all normalized by comparing to that of traditional approach CR. We can see that our ATC achieves the best performance and scalability among these approaches. For example, the normalized execution time of *lu* under BS and CS approaches are 0.85/0.15= 566.7% and 0.38/0.15=253.3% times long of which runs under our ATC approach when the number of physical nodes is 8. The reasons for this result are discussed as follows:

- For parallel applications, our ATC approach exhibits better performance and scales better than other approaches, because it can automatically adjust the time slice of VMs according to the information of their spinlock latency.
- BS is a probabilistic co-scheduling approach [4], and the probability of co-scheduling VCPUs of virtual cluster will become lower and lower with increasing number of physical nodes (VMs of virtual cluster). Thus, BS has a slight performance gain over CR when the number of physical nodes is small (e.g., 2), while the performance gain is not clear with large number of nodes (e.g., 32).
- Although CS schedules the VCPUs of a SMP VM simultaneously, all VMs belonging to the same virtual cluster are scheduled asynchronously (i.e., not co-scheduled) from the perspective of virtual cluster, which degrades the performance of tightly-coupled parallel applications. Therefore, the performance and scalability of CS are between BS and ATC.
- Since DSS adjusts the length of time slice for each VM separately according to its I/O behaviour, time slices of all VMs under DSS are probably different from each other. The VMs with long time slices will result in long spinlock latency of other VMs. By contrast, ATC sets a minimum time slice obtained by comparison for all VMs uniformly. Therefore, ATC outperforms DSS.



(a) *sp* (b) *bt* (c) *lu* (d) *cg* (e) *is* (f) *mg*

Fig. 10. Performance comparison of approaches (CR, BS, CS, DSS and ATC) when running *sp*, *bt*, *lu*, *cg*, *is* or *mg* on 2, 4, 8, 16, and 32 VMs

2) *Running Mixed Parallel Applications:* In real-world cloud systems (eg., Amazon EC2), it is very common to host different types of applications with different sizes. So in order to simulate an environment close to the real-world cloud systems, we synthesize the size and the number of virtual clusters based on the job traces of a Linux cluster (Atlas) at Lawrence Livermore National Laboratory (LLNL) [16]. In this experiment, there are 128 8-VCPU VMs hosted on 32 physical nodes (each node hosts four 8-VCPU VMs).

(1) The configuration of virtual clusters. According to the total number of VMs in our platform, we set the size of each virtual cluster in the platform to be in the range between 16 and 256 VCPUs. Specifically, we try to make the distribution of virtual clusters be consistent with the trace in Table I.

TABLE I
THE PERCENTAGE (P) OF THE NUMBER OF JOBS WITH DIFFERENT SIZES (S) (I.E., PROCESSOR COUNTS) BASED ON THE TRACES OF LLNL ATLAS

S	*8*	*16*	*32*	*64*	*128*	*256*	*others*
P	31.4%	12.6%	4.5%	12.6%	6.1%	4.5%	28.3%

For example, the number of virtual clusters with 16 VCPUs (12.6%) is about 3 times as large as that of the ones with 256 VCPUs (4.5%). Specifically, among the 128 VMs, ninety are used to build the 10 virtual clusters with different sizes, and the remaining 30 VMs act as independent VMs. Thus, the 10 virtual clusters are organized as below.

- One 256-VCPU virtual cluster (denoted as *VC1*)
- Two 128-VCPU virtual clusters (*VC2* and *VC3*)
- Three 64-VCPU virtual clusters (*VC4* $\sim$ *VC6*)
- One 32-VCPU virtual clusters (*VC7*)
- Three 16-VCPU virtual clusters (*VC8* $\sim$ *VC10*)
- Thirty 8-VCPU independent VMs

(2) Benchmarks. Each virtual cluster randomly runs a parallel application selected from *lu.B*, *bt.B*, *cg.B*, *is.B*, *mg.B*, and *sp.B*. Each independent VM runs an application randomly selected from *lu.B* and *is.B*. The execution times of the applications may differ from each other, so multiple applications may not finish at the same time, though they start the execution simultaneously on corresponding virtual clusters and independent VMs. Thereby, we run each application repeatedly with a batch program. The number of repetitions is set large enough to ensure that other applications are still running when each application finishes its 10^{th} round.

(3) Experimental results. Figure 11 shows the normalized execution time of applications running in ten virtual clusters and two randomly selected independent VMs. From Figure 11, we can see that ATC achieves the best performance among all approaches. For example, the normalized execution time of *sp* running in VC1 with ATC, DSS, CS, BS, and CR are 0.25, 0.45, 0.49, 0.9, and 1, respectively. The performance

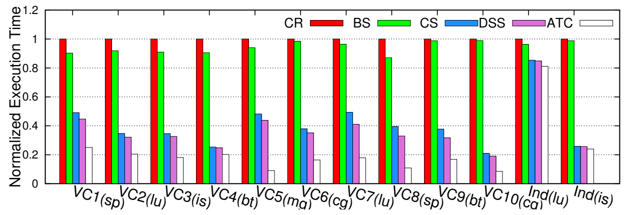


Fig. 11. Normalized execution time of *lu*, *bt*, *cg*, *is*, *mg*, and *sp* running on ten virtual clusters with different sizes and independent VMs

trends of all approaches are similar to those in Figure 10 due to the same reasons for Figure 10.

C. Scenarios with Parallel and Non-parallel Applications

In this section, we make experiments in the scenario where parallel applications and non-parallel applications coexist. For our ATC approach, we first set the time slice for VMs running non-parallel applications to be the default time slice in VMM (denoted as ATC(30ms)). Besides, in order to test the impact of adjusting time slice of VMs running non-parallel applications, we set a non default time slice for VMs running non-parallel applications, specifically, we take 6ms as an example (denoted as ATC(6ms)) to carry out our experiment.

The experimental settings about virtual clusters and the parallel applications are the same as that used in Section IV-B2. Differently, the application running on each independent VM is randomly selected from *lu*, *is* of NPB benchmark and non-parallel applications (*Apache server*, *bonnie++*, *SPEC CPU 2006*, and *stream*). We use *httperf* to measure average response time for web servers, and evaluate I/O throughput and memory bandwidth for *bonnie++* and *stream* respectively.

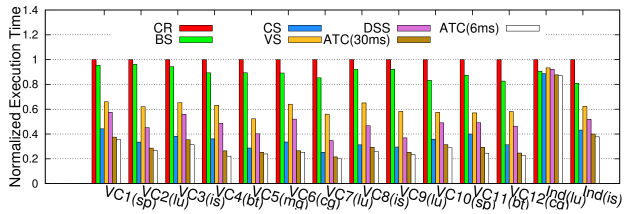


Fig. 12. Normalized execution time of *sp*, *bt*, *lu*, *is*, *cg*, and *mg* running on ten virtual clusters with different sizes, and applications selected from parallel ones and non-parallel ones running on independent VMs

From Figure 12, we can observe that ATC(30ms) achieves better performance than DSS because of the same reasons analyzed for Figure 10. Different from Figure 11 presented in Section IV-B2 where DSS exhibits better performance than CS, Figure 12 shows that DSS is inferior to CS. The performance of DSS depends on the time slice adjustment for latency-sensitive applications. When there exist latency-insensitive applications in the system, the spinlock latency

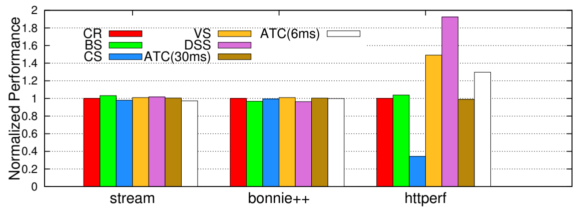


Fig. 13. Performance comparison of *stream*, *bonnie++* and web application under CR, BS, CS, VS, DSS and ATC(30ms), ATC(6ms)

of parallel application under DSS will become longer, which results in worse performance. By contrast, the performance of parallel applications under CS is not affected by the latency-insensitive applications in the system. DSS exhibits better performance than VS, because the time slice of VMs running parallel application under DSS is shorter than that in VS.

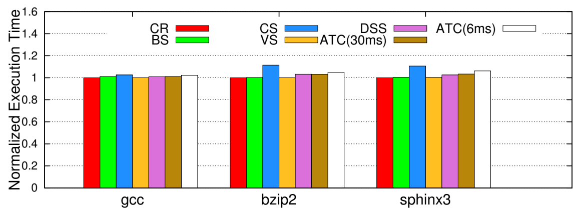


Fig. 14. Performance comparison of *gcc*, *bzip2*, and *sphinx3* under CR, BS, CS, VS, DSS, ATC(30ms), and ATC(6ms)

As shown in Figure 13, we can see that the performance of *bonnie++* with BS, CS, DSS, ATC(30ms), and ATC(6ms) approximates to those with CR. The performance of *stream* with CS and ATC(6ms) is slightly worse than that with CR, because the additional context switches lead to more cache flushes. The performance of web server with CS is about 35% of that with CR. The average response time of web server with CS approach becomes longer due to the additional VM preemption, which results in performance degradation of latency-sensitive application. The performance of web server under VS, DSS, and ATC(6ms) is better than that of CR, because these approach increase the scheduling frequency of web server, which leads to shorter average response time.

As depicted in Figure 14, the performance of CPU-intensive applications (such as *gcc*, *bzip2*, and *sphinx3*) with CS and ATC(6ms) is worse than that with CR. Meanwhile, the performance with BS, VS, DSS and ATC(30ms) approximates to that with CR. The reasons are as follows:

- Because of the VM preemption, the performance of CPU-intensive applications with CS is worse than that with CR. For ATC(6ms), the performance degradation in CPU-intensive applications is due to context switches.
- For BS, VS, DSS and ATC(30ms), the number of context switches of CPU-intensive VM is almost the same as CR. Therefore, the performance of these four approaches approximates to that of CR.

V. Related Work

In traditional systems, the principle of strict Co-scheduling [20] is to schedule and de-schedule threads belonging to the same parallel application synchronously. Although this approach can maximize synchronization efficiency, it suffers from CPU fragmentation (which may lead to reduced CPU utilization and delayed VCPU execution) and priority inversion issues [21]. Demand-based scheduling [22] involves the scheduling of threads that communicate with each other, which is a relaxed type of Co-scheduling. However, it still suffers from the issues as strict Co-scheduling does.

With the development of virtualization technology, many studies [23, 24] have explored the feasibility of running parallel applications in a virtualized environment, where the synchronization overhead of a parallel workload running in SMP VM is significant due to LHP. In such an environment, Co-scheduling in traditional environment does not perform very effectively due to the LHP issue. A hybrid scheduling framework [6] were introduced to mitigate the synchronization overhead, but it inevitably affects the performance of non-parallel applications, which results from promoting the scheduling priorities of VMs running parallel applications. Besides, it mainly focused on performance degradation on SMP VM instead of that across VMs in a virtual cluster.

A dynamic adaptive Co-scheduling [7] were proposed to deal with the performance degradation of parallel workload on an SMP VM. With this approach, the performance of non-parallel application is also degraded due to VCPU preemption. Sukwong et al. [4] presented Balance Scheduling, a probabilistic type of Co-scheduling, the principle of which is to balance the VCPU siblings in the same VM into different PCPU run queues. Since Balance Scheduling is a probabilistic co-scheduling, it has limited application performance improvement. Rao et al. [2] enforced fairness at VM-level and improved the efficiency of hosted parallel workloads in SMP VMs. A demand-based coordinated scheduling scheme for multi-threaded workloads were proposed in [25]. It is only applicable to *inter-processor interrupt* (IPI) based synchronization and cannot detect spin-based synchronization. Our previous work [26] proposed a communication-driven scheduling approach for virtual clusters in VMM. It does not optimize the performance by adaptive slice control and also has negative performance impact on non-parallel applications.

An approach in [27] was presented to handle mixed batch and interactive VMs on the same physical hardware, which can satisfy constraints on responsiveness and compute rates for each workload. However, it requires to know each application's compute/communication balance to set the constraints. Lin et al. [28] proposed a self-adaptive approach to time-sharing such machines that provides isolation and allows the execution rate of an application to be tightly controlled by the administrator, which overlooked the performance degradation issue in virtualized systems. Chen et al. [5] proposed a *dynamic switching-frequency scheduling* (DSS) approach to improve the performance of concurrent applications in SMP VMs rather than virtual clusters. In order to accelerate latency-sensitive applications, Xu et al. [15] presented a differentiated-frequency CPU microslicing approach (vSlicer). They also proposed vTurbo [29] to improve I/O performance by offloading I/O processing to a designated core, yet it overlooked parallel synchronization requirement.

VI. Conclusion and Future Work

In this paper, we investigate the performance degradation issue of parallel applications running in virtual cluster. We find that by shortening the VM's time slice, the spinlock latency can be significantly reduced, therefore synchronization overhead is mitigated, and the performance can be considerably improved. We propose a novel *Adaptive Time-slice Control* (ATC) approach for virtual clusters. We rigorously implement the ATC scheduler based on XEN and compare it to state-of-the-art methods in a series of experiments. Experimental results show that our approach can achieve 1.5-10$\times$ performance gains for tightly-coupled parallel applications compared to traditional time-slice control schemes like XEN's Credit scheduler and the scheduling approaches like Co-scheduling and Balance Scheduling, with unaffected impact on the performance of co-running non-parallel applications. In the future, we will exploit a more flexible approach to adjust the time slice of the VMs running non-parallel applications, such that the scheduler can be more aggressive and better meet the demand of the non-parallel applications for synchronization and interrupt processing. Moreover, similar to other existing approaches, we used an intrusive monitoring method in the OS kernel, which may degrade the generality. A non-intrusive monitoring method is under our plan in the future work.

VII. Acknowledgements

This work was supported by National Science Foundation of China under grant No. 61232008, National 863 Hi-Tech Research and Development Program under grant Nos. 2015AA01A203 and 2014AA01A302, U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under Contract DE-AC02-06CH11357.

References

[1] T. J. Hacker and K. Mahadik, "Flexible resource allocation for reliable virtual cluster computing systems," in *Proceedings of Supercomputing (SC)*. ACM, 2011, p. 48.

[2] J. Rao and X. Zhou, "Towards fair and efficient SMP virtual machine scheduling," in *Proceedings of PPoPP*. ACM, 2014, pp. 273–286.

[3] V. Soundararajan and J. Anderson, "The impact of management operations on the virtualized datacenter," in *Proceedings of ISCA*. ACM, 2010, pp. 326–337.

[4] O. Sukwong and H. Kim, "Is co-scheduling too expensive for SMP VMs?" in *Proceedings of EuroSys*. ACM, 2011, pp. 257–272.

[5] H. Chen, H. Jin, K. Hu, and J. Huang, "Dynamic switching-frequency scaling: scheduling overcommitted domains in Xen VMM," in *Proceedings of ICPP*. IEEE, 2010, pp. 287–296.

[6] C. Weng, Z. Wang, M. Li, and X. Lu, "The hybrid scheduling framework for virtual machine systems," in *Proceedings of VEE*. ACM, 2009, pp. 111–120.

[7] C. Weng, Q. Liu, L. Yu, and M. Li, "Dynamic adaptive scheduling for virtual machines," in *Proceedings of HPDC*. ACM, 2011, pp. 239–250.

[8] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines." in *Proceedings of Virtual Machine Research and Technology Symposium*. USENIX Association, 2004, pp. 43–56.

[9] NASA Parallel Benchmark, http://www.nas.nasa.gov/publications/npb.html.

[10] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[11] J. Ahn, C. H. Park, and J. Huh, "Micro-sliced virtual processors to hide the effect of discontinuous CPU availability for consolidated systems," in *Proceedings of MICRO*. IEEE, 2014, pp. 394–405.

[12] A. Menon, J. R. Santos, and Y. Turner, "Diagnosing performance overheads in the Xen virtual machine environment," in *Proceedings of VEE*. ACM, 2005, pp. 13–23.

[13] Euclidean metric, http://en.wikipedia.org/wiki/Euclidean_distance.

[14] P. Barham, B. Dragovic, F. K., S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of SOSP*, 2003, pp. 164–177.

[15] C. Xu, S. Gamage, P. Rao, A. Kangarlou, R. Kompella, and D. Xu, "vSlicer: latency-aware virtual machine scheduling via differentiated-frequency CPU slicing," in *Proceedings of HPDC*. ACM, 2012, pp. 3–14.

[16] Parallel Workload Trace, http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

[17] Httperf, http://www.hpl.hp.com/research/linux/httperf/.

[18] Bonnie++, http://www.coker.com.au/bonnie++/.

[19] SPEC CPU 2006, http://www.spec.org/cpu2006/.

[20] J. Ousterhout, "Scheduling techniques for concurrent systems," in *Proceedings of ICDCS*, 1982, pp. 22–30.

[21] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph, "Implications of I/O for gang scheduled workloads," in *Proceedings of IPPS*, 1997, pp. 215–237.

[22] A. Dusseau and A. Carol, "Implicit coscheduling: coordinated scheduling with implicit information in distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 19, no. 3, pp. 283–331, 2001.

[23] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling, "Data sharing options for scientific workflows on Amazon EC2," in *Proceedings of Supercomputing (SC)*. ACM, 2010, p. 9.

[24] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, "Cloud versus in-house cluster: evaluating Amazon cluster compute instances for running MPI applications," in *Proceedings of Supercomputing (SC)*. ACM, 2011, p. 11.

[25] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based coordinated scheduling for SMP VMs." in *Proceedings of ASPLOS*, 2013, pp. 369–380.

[26] S. Wu, H. Chen, S. Di, B. Zhou, Z. Xie, H. Jin, and X. Shi, "Synchronization-aware scheduling for virtual clusters in cloud," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 26, no. 10, pp. 2890–2902, 2015.

[27] B. Lin and P. Dinda, "Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling," in *Proceedings of Supercomputing (SC)*. ACM, 2005, p. 8.

[28] B. Lin, A. Sundararaj, and P. Dinda, "Time-sharing parallel applications with performance isolation and control," in *Proceedings of ICAC*. IEEE, 2007, pp. 28–28.

[29] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu, "vTurbo: Accelerating virtual machine I/O processing using designated turbo-sliced core," in *Proceedings of ATC*, 2013, pp. 243–254.