IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems

Self-boosted Co-scheduling for SMP Virtual Machines

Kun Wang, Yudi Wei, Cheng-Zhong Xu
Department of Electrical & Computer Engineering
Wayne State University, Detroit, Michigan
{kwang,ydwei,czxu}@wayne.edu

Jia Rao
Computer Science Department
University of Colorado, Colorado Springs
jrao@uccs.edu

***Abstract*—In this paper, we propose a self-boosted co-scheduling(SBCO) algorithm to reduce synchronization latency among consolidated virtual machines. Different from conventional co-scheduling which requires all runnable sibling vCPUs that are from the same VM to be scheduled at precisely the same time, SBCO reorders all these sibling vCPUs threads coarsely at the same level in their respective run queue, then schedules them at the same time window, and maintains global fairness between consolidated VMs. SBCO minimizes costly pCPU preemption and preserves the flexibility of the dynamic mapping between vCPUs and pCPUs. We have implemented SBCO in KVM and conducted comprehensive evaluations with various workloads. Results shows that SBCO is able to reduce the number of context switches significantly and achieve overall performance improve up to 10% compared with other competitors and improve up to 60% compared with the default scheduler.**

I. Introduction

SMP VMs are ubiquitous in today's scientific computing clusters, modern data centers and cloud computing infrastructures. Public infrastructure-as-as-a-service (IaaS) providers like Amazon's EC2 provides extra large instances each with as many as 16 virtual cores [1]. Though modern OSes can be seamlessly running inside a VM with techniques like para or full virtualization [25], [27] and endusers are able to run their applications as if they are running on native OSes. Due to resource contention, it is challenging to both improve resource utilization and achieve overall the best performance when consolidating many VMs together [21].

SMP VM blurs the distinction between a virtualized environment with multi-core vCPUs and a physical multiprocessor system, imposing a great challenge to vCPU scheduling. Commodity OSes often use spin-locks for exclusive access to shared code or data [23]. Such spin-locks require running processes to frequently acquire and release locks, while assuming only a short period of waiting time. They save the latency cost in circumstances such as interrupt service routines when yielding pCPU for context switch [22] is needed. However, in a virtualized environment, it is hard to keep the assumption when a vCPU is preempted while still holding a spin-lock and at the same time another sibling vCPU is still waiting for the spin-lock. Thus the sibling vCPU has to wait until the preempted vCPU to be rescheduled and releases the lock. Such switch between sibling vCPUs wastes large amounts of CPU cycles(usually in the order of a few milliseconds) and causes severe performance degradation, particularly when the waiting vCPU has been scheduled multiple times before the release of the lock. Such phenomenon is unique in multicore VM environments and often referred to as lock holder preemption(LHP) [23].

To solve the LHP issue, one solution is to detect the lock holder and avoid preemption. Lock holder could be detected by instrumenting guest OS's spin-lock primitives in paravirtualization [23] or by leveraging hardware techniques [28]. Once lock holder is detected, hypervisor's scheduler either avoids preempting lock holder or delays the lock waiter for the purpose of minimizing the synchronization latency [15]. This lock holder detection and avoidance technique is beneficial to the cases where spin-lock is infrequently involved. However, it still requires either the change of the guest OS itself or the support from low-level hardware. The lock holder detection itself also cause VM's response latency.

VM LHP issue could be addressed by co-scheduling [20], [26]. In SMP VM co-scheduling, the sibling vCPUs are co-scheduled on pCPUs simultaneously. This gives the guest OS an illusion of running on a dedicated server with the same number of processors. Co-scheduling improves performance by facilitating prompt communication and reducing synchronization delay between sibling vCPUs. For example, if one vCPU A is spinning on a lock waiting for another vCPU B to release the lock, co-scheduling A and B allows the spinning vCPU A to proceed as soon as B releases the lock without waiting for the preempted vCPU to get rescheduled. A few recent work applied co-scheduling to SMP VMs running concurrent tasks [9], [24]. Such proactive solutions are favorable to applications heavily relying on spin-lock and their performance gain outweighs the overhead of co-scheduling. However, co-scheduling often comes with side effects, such as CPU utilization fragmentation, execution delay and priority inversion [22]. These potential effects limit the massive use of SMP VM co-scheduling.

In this paper, we propose a new approach called SBCO for performance optimization in virtualized SMP environments. SBCO inherits the advantages of traditional co-scheduling such as minimizing synchronization latency and accelerating communication between vCPUs without the side effects of scheduling fragmentation and priority inversion. SBCO does not force simultaneously co-scheduling all the sibling vCPUs.

1526-7539/15 $31.00 © 2015 IEEE
DOI 10.1109/MASCOTS.2015.34

Instead, it re-adjusts the sibling vCPUs positions in their respective run queues and facilitates sibling vCPUs to be scheduled at the same time window. Specifically, SBCO first dynamically adjusts the affinity between vCPUs and pCPUs to prevent sibling vCPUs from being assigned to the same run queue. By distributing sibling vCPUs evenly to different run queues, it significantly reduces the chance of stacking sibling vCPUs. Then through minimizing the scheduling distance of sibling vCPUs defined in Section II-C, SBCO reduces the maximal scheduling distance and further reduces synchronization latency. We have implemented SBCO in KVM, and performed extensive evaluations with both micro-benchmarks and real-world workloads. The experimental results show that SBCO can significantly reduce the number of vCPU context switches and achieves an overall performance improvement by more than 10%.

The rest of the paper is organized as follows. Section II-B discusses the background and challenges of co-scheduling in virtualized SMP environment and presents the motivations of our approach. Section III presents the design of the SBCO algorithm. The details of the implementations are introduced in Section IV. Section V shows the fairness experimental results, performance evaluation and scalability analysis of our SBCO approach. Section VI discusses other related work. Section VII concludes the paper with remarks on limitations and possible future work.

II. Background and motivation

In this section, we first discuss the common synchronization mechanisms of SMP VM scheduling and the Lock Holder Preemption phenomenon. Then, we elaborate the challenges of SMP VM scheduling caused by virtualization abstraction layer. Finally, we introduce a new concept called scheduling distance, analyze its effect on synchronization latency and present a few motivation examples for our new approach.

A. Background

The vCPUs of a SMP VM are usually attached to processes or threads on a physical host and they execute codes by direct code execution or instruction emulation [25]. vCPUs are scheduled as processes or threads on a host OS. In a parallel program, a lock primitive is widely used to provide synchronization and guarantee atomic and consistent state changes in a multiprocessor system. There are typically two types of lock primitives: *semaphore/mutex* and spin-lock [20]. The former lock primitive blocks the running process until the required resources or locks become available. The scheduler swaps out the running process(unless specifically stated, we use the term process, thread and task interchangeably) and immediately schedules the next runnable process so as to avoid wasting CPU cycles. It needs process context switches to wake up the sleeping process, thus degrading system performance. In contrast, spin-lock allows the thread waiting for the required resource to keep occupying the processor and repeatedly check the lock status. It works efficiently when synchronization only takes a small amount of time(usually dozens of or hundreds of microseconds). Because the lock efficiency directly affects system performance and capability, spin-lock is widely used in modern OSes.

Spin-lock poses challenges in SMP VM scheduling. It works effectively in the cases that the lock holder only holds the lock for a very short period of time and the target resources become available soon. This is satisfied in physical environments where OS itself has control over the resources and the way of scheduling via determining whether or not to preempt out a process. However, in a virtualized environment, it is the virtual machine monitor(VMM) that retains ultimate control of the resources and vCPUs scheduling usually based on time slices. Thus the current spin-lock design may not be effecient. For instance, if a vCPU is trying to acquire a spin-lock, it has to wait until the preempted vCPU is scheduled back and release the lock. Such phenomena, referred to as LHP issue, significantly increases the lock holding time and may even waste a vCPU's time slice, especially in CPU over-committed cases. The high vCPU contention from a preempted lock leads to significant waste of CPU cycles [23].

B. Challenges

1) Dynamic vCPU affinity and vCPU stacking: A VM's vCPU affinity configuration is one of factors complicating SMP VM scheduling. VMM usually does not distinguish vCPUs from different VMs and the default scheduler often employs a global load balance policy by scheduling processes to less busy pCPUs [22]. Such policy keeps the balance of utilization between different pCPUs because of no limit of default affinity. The randomness of affinity is likely to have one or more sibling vCPUs scheduled in the same run queue. This is usually referred to as vCPU stacking issue [22]. Though stacking of sibling vCPUs is a probability type of issue, it greatly increases the lock synchronization latency in a virtualized environment. If stacked sibling vCPUs are competing for the same resource using spin-lock, the sequential vCPU execution would waste significant amounts of CPU cycles. The probability of stacking sibling vCPUs in CPU intensive workload case was studied in [22]. Their experimental results reveal that the chance of more than one sibling vCPU in the same run queue reaches as high as 45% when three CPU intensive VMs were consolidated on the same server [22]. We further conducted complementary experiments to examine the vCPU stacking issue with IO intensive and CPU-I/O mixed workloads such as SPECjbb and Kernbench. The details of these workloads are introduced in Section V-A. We implemented an independent kernel thread to periodically examine each pCPU run queue with an interval of one second. Then, we ran an kernel compile benchmark and SPECjbb in a number of VMs and counted the number of samples when more than one vCPU sibling exists in the same run queue. Table I shows the accumulated probability of stacking vCPUs can be higher than 20% for both workloads. In the case with three VMs, the probability can go beyond 42% with the Kernbench workload. Stacking sibling vCPUs can greatly increase the chance of having LHP issue. Such high stacking

TABLE I
PROBABILITY OF STACKING SIBLING VCPUS

Apps	2 VMs	3VMs
SPECjbb	20.25%	31.63%
Kernbench	33.19%	42.84%

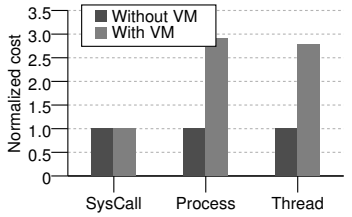


Fig. 1. Cost of vCPU context switch w/ and w/o virtualization.

ratio can even break an illusion of synchronous progress of vCPUs, which is expected from a guest OS [25]. Without this illusion, synchronization latency significantly degrades applications' performance.

2) Costly vCPU context switch: Due to the fact that each vCPU is associated with additional data structures to maintain information like the status of virtual registers, scheduling a vCPU thread causes uncertainties to the indirect cost [14], [19]. Even if a vCPU thread is scheduled back to the original pCPU, inside the guest VM, the vCPU may be serving a different task, which results in invalidation of pCPU cache. To evaluate the additional cost of context switching in a virtualized environment, we ran a context switch micro benchmark [6] in a KVM guest VM and examine the average cost of context switch and system call. As shown in Figure 1, on average, vCPU context switch costs 2.5 to 3 times more in a virtualized environment compared with the a physical environment. Therefore, effective vCPU context switch is one of the design goals of our SBCO approach.

C. The effect of scheduling distance

A commodity scheduler commonly splits up pCPU time between runnable processes in a fine-grained way in the order of nanosecond accurate time slices. Recall that sibling vCPUs could be stacked in the same run queue of a pCPU. Let P_{run} denote the current total number of processes in a run queue and T_w be a process's dynamic priority, also referred as the weight. T_{wi} is the weight of process `i` in a run queue. Let S_{min} be *sched_min_granularity*, the minimum time a task will be allowed to run on CPU before being forcibly preempted out. Let $S_{latency}$ be *sched_latency*, the default scheduling period in which all run queue processes are scheduled once. S_{min} and $S_{latency}$ are configurable parameters in the default scheduler. T_{slice} is the time slice of a process with the weight of T_w. Assuming all the processes in a pCPU run queue have the same weight, then each process also has the same time slice T_{slice}. The actual scheduling period T_p, which is the total time all run queue processes are scheduled once, is calculated in the following formula. T_p is also the maximum time one process has to wait until all other process to yield pCPU. For instance, if a VM has two vCPUs A and B stacked in the same pCPU run queue. Assuming A is at the front of the run queue and B is at the tail of the run queue, in the worst CPU intensive workload case, B has to wait for T_p after A gets the chance to be scheduled.

$$P_{total} = S_{latency}/S_{min};$$

$$T_p = \begin{cases} S_{latency} & P_{run} \leq P_{total}; \\ P_{run} * S_{min} & Otherwise. \end{cases}$$

$$T_{slice} = T_p * (T_w / \sum_{1}^{n} T_{wi}), i \in [1, P_{run}].$$

The scheduling time can be viewed as an axis with the time to schedule a vCPU as the origin, and time slots when vCPUs will be scheduled as scheduling ordinates. We define a VM's schedule distance as the maximal difference of sibling vCPUs' scheduling ordinate. The latter is also the relative position in pCPU run queues. As the illustration example shown in Figure 2, VM1 and VM2 are running on a physical machine with two pCPUs. At T_0, both of VM1's vCPUs are in a pCPU's run queue and are ready to run. Moreover, VM1's $vCPU_0$ is the next candidate to be run on $pCPU_0$ and will be scheduled immediately when $pCPU_0$ becomes available. However, VM1's $vCPU_1$ is currently at the bottom of $pCPU_1$'s run queue and does not start to run until T_n. Let Pos(t) denote process t's position in its run queue and L(n) as the length of $pCPU_n$'s run queue. We define *delta*, also denoted as D(VM1)in Figure 2, as the maximum difference of scheduling distance disparity between sibling vCPUs as follows:

$$Pos(vCPU_0) = 1;$$

$$Pos(vCPU_1) = L(pCPU_1);$$

$$delta = |Pos(vCPU_0) - Pos(vCPU_1)|.$$

Though $vCPU_1$ is runnable in this case, if $vCPU_0$ is waiting for $vCPU_1$ to release the lock, then $vCPU_0$ has to wait for $pCPU_1$ to reschedule task $vCPU_1$. At the same time, $vCPU_1$ has to wait for processes before it to acquire pCPU, execute for T_{slice} and then yield pCPU for reasons like waiting for IO or using up its own time slice. The waiting time could be as long as T_p in the worst case, which is often the order of tens of milliseconds. Depending on the length of the run queue and how long a task typically run before getting switched out again, it can considerably degrade performance, especially in the dense consolidation of CPU intensive workloads, in which the average run queue size is usually large. A VM's scheduling distance can greatly increase synchronization latency even if the sibling vCPUs are dispatched to different run queues without stacking vCPU.

$$D_{threshold} = \alpha * Q_{size}, \alpha \in (0, 1).$$

We investigated VM's scheduling distance by running a few VMs with the average run queue size as six and ten respectively. We implemented an independent kernel thread to periodically check each pCPU's run queue and simply count the cases that a vCPU is ready to be scheduled but

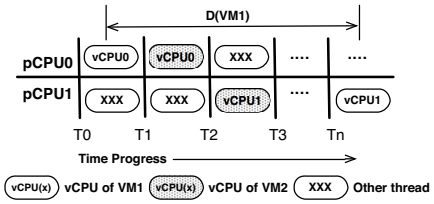


Fig. 2. Scheduling distance sibling vCPUs.

TABLE II
PROBABILITY OF SCHEDULING DISTANCE EXCEEDS THE THRESHOLD

RQ size	Probability of exceeding $D_{threshold}$	
	Kernbech	Parsec
6	42.36%	40.75%
10	48.21%	45.13%

with one or more sibling vCPUs having $D_{threshold}$ distance in the respective run queue. $D_{threshold}$ is equal to α times of a pCPU run queue length, denoted as Q_{size} in the above formula. Note that a big α value leads to large $D_{threshold}$, and less probability of exceeding the threshold. In contrast, a small α may cause frequent adjustment for balancing sibling vCPUs. We ran Kernbench and NPB benchmarks(refer to Section V-A for detailed benchmark introduction) and computed the probabilities for sibling vCPUs to exceed different $D_{threshold}$. As As shown in Table II, the probability of exceeding $D_{threshold}$ with α as 2/3 is between 40% and 50% for both workloads when the average run queue size is six and ten respectively. The α parameter is dynamically configurable and its default value is 2/3. To alleviate the synchronization latency problem from the LHP issue, we propose SBCO which leverages the scheduling distance information to make scheduling decisions.

III. SELF-BOOSTED CO-SCHEDULING

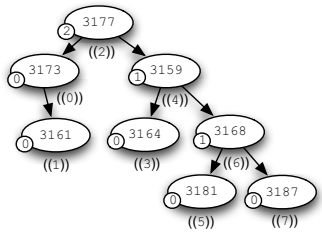
SBCO is designed to reduce costly vCPU context switching and shorten the synchronization latency caused by `spin-lock` holder. It maintains a balance between a fast vCPU with a large scheduling ordinate and a slow sibling vCPU with a small scheduling ordinate. We try to answer the following questions when designing SBCO. 1) How to avoid stacking sibling vCPUs? 2) How to flexibly balance a fast vCPU and a slow sibling vCPU? 3) How to avoid forcibly preemption and reduce vCPU context switches? 4) How to control SBCO's overhead while keeping its efficiency? In this section, we elaborate the design details, and discuss a few optimization techniques of the design.

A. Extending red black tree

In existing Completely Fair Scheduler(CFS) scheduler, each pCPU has an independent run queue. All the processes in a pCPU run queue are managed with a self-balanced binary search tree called read black tree(RB tree) [2]. The process with the smallest `vruntime` (virtual runtime in nanoseconds), which corresponds to the left most node in the RB tree, is chosen by the scheduler as the next candidate to run.

```
RBINDEX  PID  VRUMTIME          PARENT_VRUN    COMMAND  ORDER
  <0>   3173  [5556.399127] --> [5556.792013]  qemu-kvm  ((0))
  <0>   3161  [5556.620530] --> [5556.399127]  cpuhog    ((1))
  <2>   3177  [5556.792013] --> [0]            cpuhog    ((2))
  <0>   3164  [5557.007206] --> [5557.192115]  qemu-kvm  ((3))
  <1>   3159  [5557.192115] --> [5556.792013]  cpuhog    ((4))
  <0>   3181  [5557.198289] --> [5559.010627]  qemu-kvm  ((5))
  <1>   3168  [5559.010627] --> [5557.192115]  cpuhog    ((6))
  <0>   3187  [5560.140783] --> [5559.010627]  qemu-kvm  ((7))
```

(a) cfs run queue snapshot



(b) extended red black tree with index

Fig. 3. Scheduling distance in a cfs run queue.

The default RB tree is constructed starting from the arrival of the first process in a run queue. New processes with smaller `vruntime` will be placed before the left most child and the tree will rotate itself to keep balanced. When a process finishes running, its `vruntime` with the total execution time weighted by its priority is updated. Once a process leaves its run queue, the associated node will be removed from the RB tree of that run queue and the RB tree will also rotate to keep balanced.

The default RB tree does not maintain processes' relative positions in a run queue. Instead, it simply sorts processes according to their `vruntime`. Therefore, it involves RB tree traversal in order to calculate how many processes in a run queue are ahead of a process, which is contradicting to the simple but efficient design philosophy of scheduler design. We solve this dilemma by extending the default RB tree data structure by adding `RB index` when constructing a RB tree. The `RB index` of a RB tree node is defined as the total number of nodes in the left child sub-tree of this node. We summarize all the terms we mentioned as following:

RB index: The total number of nodes in the left child sub-tree of a node in a RB tree. It is updated with RB tree balance rotation when there is a node added into or removed from a run queue.

Scheduling ordinate: A vCPU process's position in its run queue. It reflects the maximum number of processes ahead before a process gets scheduled.

Scheduling distance: The difference between of the fastest vCPU's `scheduling ordinate` and the slowest vCPU's `scheduling ordinate` in a VM. Figure 3(a) gives a snapshot of a pCPU run queue. To demonstrate the RB tree of a run queue, we run a four vCPUs KVM VM with CPU intensive workloads to keep all vCPUs busy. Meanwhile, we ran a four threads application(`cpuhog`) on the host machine to represent non-vCPU threads in the pCPU run queue. In reality, these non-vCPU threads could be kernel threads or

Algorithm 1 SBCO Main Algorithm

```
1: procedure SBCO(VOID)
2:     a ← leftmost task in RB tree
3:     n ← a
4:     if Task a is a vCPU task then
5:         orda ← SCHED_ORDINATE(a)
6:         for all Task t's sibling task b do
7:             if Task b's dirty flag is set then
8:                 continue
9:             end if
10:            ordb ← SCHED_ORDINATE(b)
11:            delta ← abs(orda - ordb)
12:            if delta ≥ D_threshold then
13:                n ← Task a's successor in the run queue
14:                SCHED_BLANCE(t, b)
15:                break
16:            end if
17:        end for
18:    end if
19:    return n
20: end procedure
```

any applications running together with a VM hypervisor. All the processes's information, including process name(command column), process id(pid column), virtual runtime(vruntime column) and parent process's virtual runtime, are listed in the figure. All the vCPUs in a KVM VM have the same process name(`qemu-kvm`) . Based on the virtual runtime relationship between child process and parent process in the figure, a RB tree is constructed in Figure 3(b). The `RB index` column in Figure 3(a) and each number in the small circles in Figure 3(b) represents the total number of nodes in the left child sub-tree of a process. Based on `RB index`, `scheduling ordinate` of a task in its run queue is calculated by Algorithm III-B and printed out in the right most column in Figure 3(a) and in the brackets under each tree node in Figure 3(b). The details of complexity analysis and performance considerations are provided in the following subsections.

B. SBCO Algorithm

Algorithm 1 shows the pseudo code of SBCO. For each scheduling period, SBCO first chooses a task with the smallest `vruntime` in a run queue as the default candidate to run (line 2). If the current candidate is a vCPU process, which is implied by the process's name, SBCO then calculates this vCPU's scheduling ordinate (line 5), iterates other sibling vCPUs to identify if there is any runnable sibling vCPU with large scheduling distance and decide if it is necessary to balance the fast and slow sibling vCPUs. If the scheduling distance between two sibling vCPUs, calculated in line 11, exceeds the $D_{threshold}$, which indicates the candidate vCPU runs too fast, then there is a need to enforce adjustment to delay the fast vCPU and speed up the slow one(line 13). As a result, the previously selected candidate vCPU, the default left most node, is no longer the next task to run. Instead, the scheduler chooses the candidate's next successor process in the RB tree to run.

Note that each vCPU process is guaranteed one time to be scheduled in a scheduling period T_p, defined in Section II-C, we design two approaches to eliminate repetitive adjustment on one vCPU and ensure each vCPU being scheduled once in T_p respectively. First, we mark those sibling vCPUs that have been already adjusted as dirty. This dirty tag aims to prevent a vCPU thread from repeatedly yielding its pCPU. The adjustment is realized as follows: the vCPU with the smallest scheduling ordinate lends certain amount of `vruntime` to the sibling vCPU with largest scheduling ordinate, causing both move towards the center of their respective run queues. When the scheduler decides a task to run, it first checks a vCPU's dirty tag and it will not re-balance with the sibling vCPU marked as dirty. Second, as shown in function `SCHED_DISPATCH`, each VM's sibling vCPUs are dispatched to different pCPU run queues, preventing them from the stacking issue. But we let the default load balancing to take over the control of the mapping between a pCPUs and a vCPUs. This still maintains the physical resources utilization efficiency.

C. Performance Considerations

We have following design considerations to minimize the overhead of SBCO: 1) maintain the RB index for each vCPU. 2) set dirty tag for balanced vCPUs. 3) maintain a debt list for the adjustment between sibling vCPUs. In the following section, we analyze these design considerations and their tradeoff.

RB index. Instead of directly keep each vCPU process's scheduling ordination in pCPU run queues, SBCO seamlessly inserts RB index information into the existing tree structure. Due to the fact that each vCPU process is swapped in to or be swapped out from a run queue frequently during the execution, the scheduling ordinate of a vCPU is constantly updated with the change of its position in its resident run queue. There are two advantages to introduce RB index. First, RB index can be used to efficiently calculate the scheduling ordinate. As show in function *SCHED_ORDINATE* in Algorithm 2, the calculation of a vCPU's scheduling ordinate only involves RB tree traversal from root to vCPU's corresponding node and the complexity is bounded to O(log(n)). Second, a vCPU's RB index is updated dynamically with the RB tree rotation. This update only involves the change of the nodes on the path from root to the node. The additional cost on operating RB index is only limited to assigning value to the *rb_index* in the data structure without any extra lock.

Debt list. It is very costly for the scheduler to hold the locks of two run queues while changing one of them, such as migrating processes. In order to avoid locking two run queues at the same time when conducting the adjustment, we maintain an independent *debt* list for each pCPU run queue. Therefore, changing a *debt* list does not require to acquire that run queue's lock. As shown in function `SCHED_BLANCE` in Algorithm 2, when balancing task T_a and T_b, T_a's *vruntime* is adjusted and its location in its tree is updated immediately. However, T_b's *vruntime* and location are recorded in the associated pCPU *debt* list temporarily. The change is delayed to the time when the scheduler needs to choose a process from T_b's resident

Algorithm 2 SBCO balance and RB ordinate algorithm

```
1: procedure SCHED_DISPATCH(T_a)
2:     cpus ← all pCPUs
3:     if T_a is a vCPU then
4:         for all Task t's sibling task b do
5:             cpu_occupied ← b's pCPU
6:             cpus ← cpus − cpu_occupied
7:         end for
8:     end if
9:     return cpus
10: end procedure

11: procedure SCHED_BLANCE(T_a, T_b)
12:     if T_a is clean then
13:         Adjust T'_a s vruntime
14:         Reposition T_a in its RB tree
15:         Update the debit list of T'_b s run queue
16:     end if
17: end procedure

18: procedure SCHED_ORDINATE(T_a)
19:     parent ← T'_a s parent task in rb tree
20:     n ← RBindex_{T_a}
21:     while parent ≠ null do
22:         if T_a is parent's right child then
23:             n ← n + RBindex_{parent} + 1
24:         end if
25:         T_a ← parent
26:         parent ← T'_a s parent
27:     end while
28:     return n
29: end procedure
```

pCPU run queue. As a result, the actual balancing is conducted in two different times, which avoids locking two run queue simultaneously. In addition, since the scheduler has to check its *debt* list and apply the changes to T_b before it chooses a process candidate to run, *SBCO* incurs the additional marginal cost, mainly on updating data structure. The default scheduler does not distinguish a vCPU process from other normal tasks, *SBCO* always checks a task's name when making scheduling decisions, and only balances *qemu-kvm* processes, which are KVM vCPUs. Other non-vCPU processes are ignored for balancing.

Dirty tag. To prevent repeatedly adjusting the same vCPU in the same balance round, which may cause starvation, we mark the changes when a vCPU is adjusted but the changes has not been applied yet, either in the case of being given *vruntime* by or lending *vruntime* to other siblings. For instance, as shown in Algorithm 1 line 7, if a sibling vCPU has been adjusted before, SBCO passes that sibling vCPU and continues to check if there is any other available sibling for balancing. After the change recorded in a *debt* list is applied to a vCPU, this vCPU's dirty flag is cleared and the vCPU becomes available again for future balancing. The detailed cost analysis is provided in the evaluation section.

IV. Implementation

We implemented the prototype of SBCO algorithm in KVM with Linux kernel 2.6.34.4. KVM is a user friendly virtualization solution seamlessly integrated into Linux kernel. In KVM, there are two kinds of important threads which are QEMU threads and vCPU threads. The QEMU threads share the responsibility for the actual disk I/O by emulating the hardware devices. The vCPU threads execute the real code. KVM relies on existing Linux scheduler for the scheduling of vCPUs and each vCPU is treated as a normal task in host OS.

Our SBCO algorithm is implemented based on CFS scheduler. We extended the default RB tree to carry RB index and implemented associated APIs to calculate scheduling ordinates. We added new *rb_index* and *rb_dirty* to each node of the RB tree. The *rb_index* keeps the number of nodes on the left side of a node in the RB tree. It is updated during self rotation of a RB tree when a new task is enqueued or an existing task is dequeued. In addition, the *rb_dirty* records if a process is needed to be adjusted. To avoid repeatedly yielding the same process in one balance round, when *rb_dirty* is set, the process is ignored for balancing with its sibling.

Note that a vCPU's run queue is updated in three cases: 1) a vCPU process is created and then inserted to a run queue. 2) a vCPU process wakes up from sleep and needs to enter a run queue. 3) a vCPU is migrated between two pCPUs. We instrumented the scheduler to avoid stacking sibling vCPUs in all these cases. We first modified CFS scheduler to dynamically set a task's *cpus_allowed* field which is a set of pCPUs that a task can run on. Then, we changed the scheduler's default load balance function *can_migration* by limiting the options of migration destination pCPUs.

For comparison, we also implemented the idea of balanced scheduling and two conventional co-scheduling approaches [22]. The balanced scheduling simply puts sibling vCPUs to different pCPU run queues by adjusting `cpus_allowed` field of their process structure. We also developed two more co-scheduling approaches. First, when $vCPU_0$ of a VM is scheduled, the rest of sibling vCPUs are forcibly scheduled on other pCPUs concurrently. Second, let $pCPU_0$ decide to co-schedule all the vCPUs of a VM depending on which vCPU the first pCPU will run. We refer these two co-scheduling approaches as PROCCO and CPUCO respectively in our evaluation. In both cases, an inter-processor interrupt (IPI) request is sent to the related pCPU to force context switch and pick a sibling vCPU instead of the default lowest `vruntime` task to run. In our prototype, we define the scheduling distance threshold to be 2/3 times of the size of each run queue, as suggested in Section II-C.

V. Evaluation

In this section, we first introduce our experiment environment and benchmarks selected. Then, we evaluate SBCO and compare its performance with the default scheduler and other representative solutions.

A. Experiment Design

We ran all experiments on Dell PowerEdge1950 physical machines with two quad-core Intel Xeon CPU and 8GB memory, running Linux kernel 2.6.34.4. The guest VMs run CentOS 5.4 without any modification. Since it is the vertical

length of a pCPU run queue, not the horizontal total number of run queues, that affects a VM's scheduling distance, we selected following workloads to saturate vCPUs. Meanwhile, on a physical server, we consolidate a few VMs and ran CPU intensive workloads to further saturate pCPUs run queues. The detailed specifications of the benchmarks we used to measured the SBCO's performance and overhead are listed as follows:

Parsec Parsec is a benchmark suite for Chip-Multiprocessors (CMPs) that focuses on emerging applications. It includes a diverse set of workloads from different domains such as interactive animation or systems applications that mimic large-scale commercial workloads [11]. We used the pthread implementation of the benchmarks which uses `spin-lock` for synchronization.

SuperPI SuperPI [5] is a CPU-bound workload to calculate the digits of PI. We run SuperPI in a few VM as CPU intensive workload and also use it as disturbance workload.

NPB NAS parallel benchmarks [10] contain 9 parallel programs derived from computational fluid dynamics applications. We activate the environment variable OMP_WAIT_POLICY to allow benchmarks using busy-waiting synchronization.

Kernbench We use the parallel make benchmark, Kernbench [3], to compile Linux 2.6.34.4 kernel source with 16 threads (make -j 16) and use the kernel compile completion time as the performance metric. The VMs running Kernbench are configured with enough memory to avoid swap storms.

SPECjbb We use SPECjbb2005 [4] v1.07 and BEA JRockit 6.0 JVM. It emulates a three tier client/server system by spawning multiple java threads to simulate users transaction requests in multiple warehouses. Synchronization is required when user requests and server side management operations need to access the same database table. We start with one warehouse(thread) and stop at 16, and report the average business operations per second(bops) from 8 to 16 warehouses.

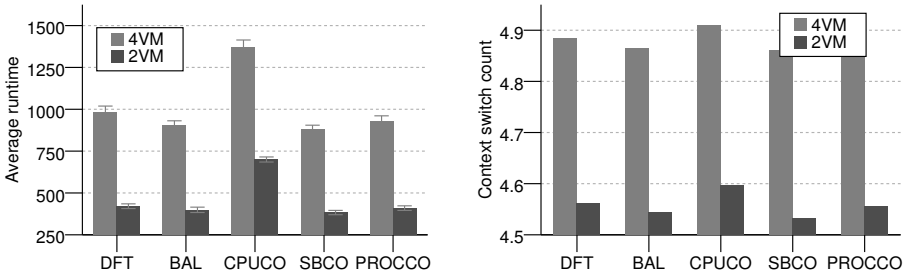
B. Experimental Results

1) Performance: We ran Kernbench in one 4-vCPU VM with other one or three VMs running CPU-intensive SuperPI workload and measure the completion time of KernBench. To avoid swap storms and eliminate uncertainties, we assigned about 2G memory to each VM to allow KernBench caches all the Linux kernel source in the RAM. KernBench frequently reads files or links through Linux VFS layer, thus incurring file system's inode lock contentions, which is protected by spin-lock in kernel space. We compare the completion time of KernBench due to following different scheduling approaches: the default CFS scheduler (*DFT*), balanced scheduler (*BAL*), process based co-scheduling (*PROCCO*), CPU based co-scheduling (*CPUCO*) and our *SBCO*.

Figure 4(a) shows that, due to the heavy lock contention, the default CFS scheduler performs worst compared with *BAL* and *SBCO*, both of which split vCPUs to different run queues to reduce the overhead of LHP. *SBCO* achieves 14% performance improvement over *DFT* and 6% over *BAL*. From Figure 4(a), we also observe *CPUCO* leads to performance degradation significantly. This demonstrates that allow one pCPU to lead other pCPUs to co-schedule sibling vCPUs is not necessarily feasible as expected, thus we remove *CPUCO* for comparison in our remaining evaluation. Kerbench also provides the count number of context switches during the execution. Figure 4(b) shows *SBCO* is capable of reducing the number of context switches by at least 3%, that is equivalent to a large amount of context switches given *SBCO* cause as many as 76400 context switches. *CPUCO* leads to performance loss due to the tremendous increase of the number of context switches.

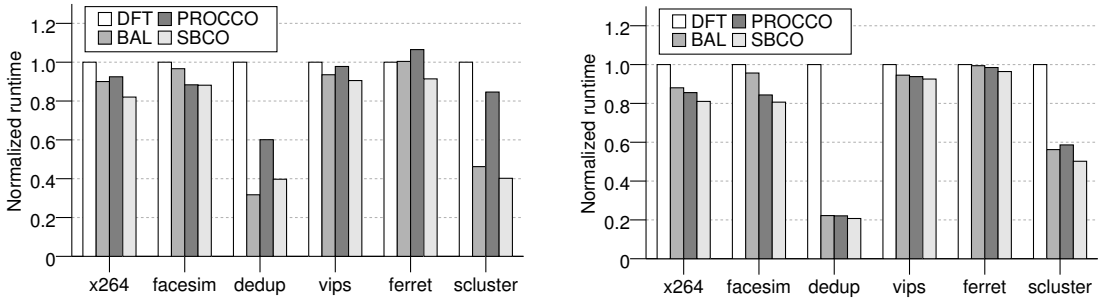
Figure 5(a) shows the normalized performance of Parsec benchmark due to different scheduling approaches in one VM. We also ran three other CPU-intensive SuperPI VMs to make the average queue size of each pCPU length stays at eight. Though different workloads have different average runtimes, *SBCO* outperforms *DFT* as well as other approaches in all test cases. More specifically, for the *dedup and scluster* workloads, *SBCO* improves performance by up to 68% and 52% respectively compared with the *DFT* case. At the same time, *SBCO* outperforms *BAL* by 7% and 9% respectively. Note that the *dedup* benchmark uses a pipelined programming model to parallelize the compression to mimic real-world implementations [11]. Both *SBCO* and *BAL* avoids the LHP issue resulted from frequency synchronization between pipeline steps. Similarly, *scluster* gains benefit from *SBCO* while processing large amounts of continuously produced data. We also observed *BAL* performs closely to our *SBCO* with workloads such as *x264, facesim, ferret*. There are two reasons for such close performance improvement. First, *SBCO* is also built on distributing sibling vCPUs to different pCPUs, which is the core of *BAL*. Therefore, *SBCO* works like *BAL* unless there is large scheduling distance detected. Second, *SBCO* involves marginal additional cost to minimize the scheduling distance between sibling vCPUs(analyzed in V-B3). If the workload itself does not have large amount of synchronization between threads, the balancing only affects short-term fairness. In Figure 5(b), we ran four more VMs running CPU-intensive workload so as to increase the average run queue size to be twelve. *dedup* achieved even higher performance gain (up to 70% over *DFT*) compared with its performance gain in four VMs case in Figure 5(a). Such phenomenon demonstrates scheduling distance can contribute to significant performance loss when the average pCPU run queue size increases.

2) Throuhput: To evaluate the effect of *SBCO* on applications' throughput, we kept one VM running SPECjbb benchmark and compared the average throughput due to different scheduling approaches. Meanwhile, we increase the number of disturbing VMs from one to five. All VMs was configured with four vCPUs, the same as the total number of pCPUs. Each disturbing VM ran the CPU-intensive Supper PI workload to keep pCPUs busy so as to maintain the same amount of average run queue size. These CPU-intensive applications usually keep occupying CPU resource and get preempted by the scheduler once they use up their time slices. Therefore the more the disturbing VMs, the longer the run queue, resulting in large waiting time due to large scheduling distance. We used one single JVM instance for SPECjbb benchmark and gradually



(a) Average runtime of kernbench (b) Context switches numbers of kernbench in Log scale.

Fig. 4. Average runtime and context switches of running kernbench.



(a) Parsec performance with average rq size is eight (b) Parsec performance with average rq size is twelve

Fig. 5. Normalized performance of running Parsec benchmark when with different average run queue size.

increased SPECjbb workload by increasing the its warehouses numbers. The average throughput is shown in Figure 6. It can be seen that *PROCCO*, *BAL* and *SBCO* outperform the default CFS due to their alleviation of the synchronization latency problem. *SBCO* achieves about 6% higher throughput than the *PROCCO* due to the reduction of context switching cost. It also yields 4% higher throughput compared with *BAL* because of the mitigation effect of scheduling distance. From Figure 6, we can also observe SBCO's performance gain is higher in the five disturbing VMs case compared with there is only one disturbing VM. It is because long run queue tends to incur relatively high synchronization delay.

3) Scalability: To study the scalability of *SBCO*, we ran different Parsec workloads in one VM and increased the average pCPU run queue size by launching more CPU-intensive applications on the physical host. *SBCO* identifies a vCPU process by checking the name of a thread. It always dispatch sibling *qemu-kvm* processes to different run queues. In our experiment, we note that running a large amount of disturbing VMs requires huge physical memory space. Instead, we ran multiple four threads CPU-intensive applications and assign threads' name to be *qemu-kvm*. Therefore these disturbing threads are also treated like vCPUs and they are dispatched to different run queues. The average run queue size is increased gradually with more disturbing threads being launched . Figure 7 shows the normalized completion time of different Parsec workloads with respect to the default *DFT*. As suggested by

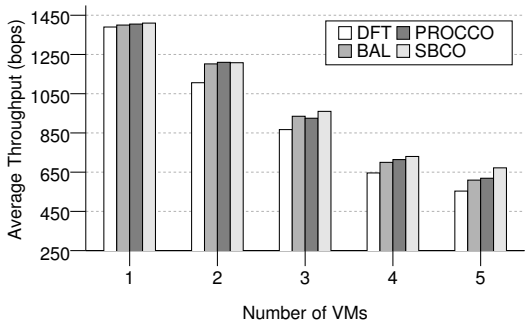


Fig. 6. Performance of SPECjbb benchmark.

the normalized numbers in the figure, *SBCO* is able to improve the performance by 61% with *dedup*. The cost of iterating a vCPU's sibling vCPUs and calculating the scheduling distance remains unchanged in all the cases. *dedup* benefits from *SBCO* most due to its heavy synchronization overhead.

4) Fairness: In this section, we show the effectiveness of SBCO in VM level fairness. We ran four VMs with multi-threaded CPU intensive NPB workload to saturate vCPUs. On the physical host, we implemented a kernel thread to periodically sample the total execution time of each VM by summing up each vCPU's execution time. The sampling period varies from 1s, 5s, to 120s and each sample calculates the maximum difference, referred as lag, between VMs. The configurable sample period is open to user applications through

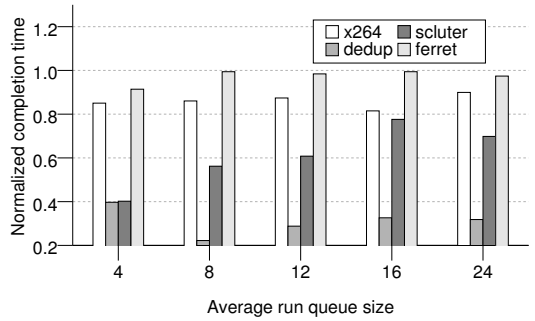


Fig. 7. Scalability of SBCO on Parsec workloads.

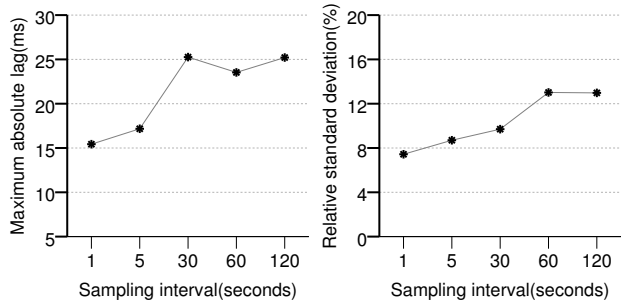


Fig. 8. Relative Standard Deviation(RSD) of the Maximum Absolute Lag(MAL) of each VM.

Linux's *sysctl* interface, and the sampling thread is assigned with highest priority to avoid competing CPU resource with vCPUs. Let T_{vm} represent the sum of all vCPUs' execution time in a VM and Lag_t be the maximum difference of the execution time of all the VMs at time t, denoted as maximum absolute lag(MAL). We repeated the experiment for five times and present the average lag value in Figure 8.

$$T_{vm} = \sum_{1}^{n} T_{vCPU_i}, n = 4;$$

$$Lag_t = Max(T_{vm_i}) - Min(T_{vm_j}), i, j \in [1, 4].$$

Recall that each VM's vCPUs are distributed into different run queues, thus each pCPU run queue only has one vCPU thread of every VM. Figure 8 shows the MAL and the relative standard deviation(RSD) with respect to different sample intervals. As shown in Figure 8, the lag varies a little with different sample periods. More specifically, when the period goes from 1s to 120s, the average maximum absolute lag varies from 15ms to 25ms. Compared with the average 20ms maximum lag with default CFS scheduler, our SBCO has a negligible impact on scheduling fairness between VMs. According to the RSD, the variation ranges from 7% to around 14% and the overall RSDs are bounded to 15%.

VI. Related Work

The issue of preempting a parallel process which holds a lock was studied intensively in the past, see [8], [18], [23] for example. Virtualization makes the synchronization delay problem even more challenging due to LHP issue. In general, there are two approaches to address this issue: hardware assisted approaches and pure software scheduling solutions.

The hardware assisted approach detects lock holder with the low level hardware and assistant scheduler to schedule in and out the proper vCPUs dynamically to mitigate LHP problem. Modern processors provide architectural support for heuristically detecting contended spinlocks [7]. For instance, PAUSE instruction is used by commodity OSes(e.g. Windows) in the spin lock for power efficiency consideration, therefore by identifying the execution of PAUSE instruction, the spin lock holder can also be detected [7]. In [28], the authors proposed a hardware assisted spin-lock mechanism to detect the cases in which a vCPU is not performing useful work and to suggest scheduler to preempt that vCPU to run a different, more productive vCPU. The heuristic lock-holder detection may cause frequent vCPU preemption. Moreover, this type of hardware assisted lock holder detection usually requires modifying guest OS, which is only possible with para-virtualization. This solution is not always feasible for guest OSes like Windows which is hard to instrument.

A typical software approach is co-scheduling, which was originally proposed to schedule concurrent threads simultaneously [13], [20], [24]. Previous works [9], [16], [26], [29] applied co-scheduling to SMP VMs to facilitate the vCPU communication and reduce application synchronization latency. They alleviate the LHP issue because all sibling vCPUs are scheduled simultaneously. However, classic co-scheduling algorithm has its inborn drawbacks such as CPU fragmentation, priority inversion and execution delay [17]. In addition, co-scheduling is likely to cause costly vCPU preemption during context switching in virtualized environment.

To avoid the limitations of classic co-scheduling, an improved co-scheduling algorithm named balanced scheduling (*BAL*) was proposed in [22]. In stead of preventing LHP, *BAL* alleviates the effect of LHP issue by distributing sibling vCPUs to different pCPUs without forcing the vCPUs to be scheduled at the same time. It never delays execution of a vCPU to synchronize with other sibling vCPUs. Our *SBCO* inherits the advantages of traditional co-scheduling such as minimizing synchronization latency and speedup the communication between vCPUs. It coarsely re-adjusts the sibling vCPUs position in their run queues and facilitate sibling vCPUs to be scheduled coarsely at the same level. *SBCO* not only dynamically adjusts the affinity of vCPUs to avoid sibling vCPUs from being dispatched into the same run queue, but also it balances the sibling vCPUs in different run queues by shorten their scheduling distance. This further reduces the synchronization latency in CPU over committed case. Our *SBCO* requires no hardware support and can be easily implemented.

VMware also developed a few versions of co-scheduling solution for ESX server, from strict co-scheduling [26] to relaxed co-scheduling [25]. The later one was further refined in ESX 4.x to stop only advanced vCPUs, instead of all vCPUs. In all these co-scheduling approaches, scheduler tends to forcibly start or stop some vCPUs which incurs significant

context switching cost. Our SBCO balances sibling vCPUs and avoids arbitrary forcing vCPUs co-scheduling. In addition to these co-scheduling approaches, PACMan [21] provided some insights for performance aware VM consolidation. Matrix [12] proposed an approach to achieve predicable performance in cloud with machine leaning. Difference from these works that considers multiple contributing factors to performance, our approach focused on the LHP issue and attempted to alleviated its impact on performance and resource utilization. Gleaner [14] introduced an interesting idea to solve the blocked waiter wakeup(BWW) problem. Our work shares the same goal of reducing costly vCPU context switch. However, Gleaner consolidates short idle periods on multiple vCPUs into long idle periods on fewer cores, thus reducing the frequency that vCPU enter/exit idle loops. This approach may have limited improvement in heavy loaded cloud with CPU intensive applications. In those cases, vCPUs are busy most of time and they get rescheduled mainly due to running out of scheduling period other then entering idle loops. Our approach optimizes vCPU scheduling especially in over loaded cloud with high VM consolidation ratio.

VII. Conclusions

In this work, we propose *SBCO*, a new scheduling scheme for performance optimization in virtualized SMP environment. *SBCO* first inherits the advantages of traditional co-scheduling such as minimizing synchronization latency and speedup the communication between vCPUs. Meanwhile, it avoids the scheduling fragmentation and priority inversion issue because *SBCO* does not require co-scheduling all the sibling vCPUs precisely at the same time. Instead, it coarsely adjusts the sibling vCPUs position in their respective run queues for balance purpose and facilitate sibling vCPUs to be scheduled coarsely at the same level. We implemented the prototype of *SBCO* based on CFS scheduler and conducted evaluations with KVM VM. Our experimental results show that *SBCO* brings more than 10% performance improvement for many applications. Depending on the usage of spin-lock for synchronization, the impact of scheduling distance may also vary with applications. In the future, we plan to further study applications' sensitivity to the scheduling distance of sibling vCPUs and propose an online adaptive threshold for the purpose of dynamically balance sibling vCPUs with different granularity.

References

[1] Ec2 instance type. http://aws.amazon.com/ec2/.

[2] *Inside the Linux 2.6 Completely Fair Scheduler*. http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/.

[3] *Kernbench Benchmark*. http://freecode.com/projects/kernbench.

[4] *SPECjbb*. http://www.spec.org/jbb2005/.

[5] *SuperPI*. http://www.superpi.net/.

[6] *lmbench*. http://lmbench.sourceforge.net/, 2001.

[7] Intel 64 and ia-32 architectures software developer's manual volume 3., December 2011.

[8] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 1992.

[9] Y. Bai, C. Xu, and Z. Li. Task-aware based co-scheduling for virtual machine system. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, New York, NY, USA, 2010.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008.

[12] R. C. Chiang, J. Hwang, H. H. Huang, and T. Wood. Matrix: Achieving predictable virtual machine performance in the clouds. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 45–56, Philadelphia, PA, June 2014. USENIX Association.

[13] G. S. Choi, J.-H. Kim, D. Ersoz, A. B. Yoo, and C. R. Das. Coscheduling in clusters: Is it a viable alternative? In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 16–, Washington, DC, USA, 2004. IEEE Computer Society.

[14] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 73–84, Philadelphia, PA, June 2014. USENIX Association.

[15] Y. Dong, X. Zheng, X. Zhang, J. Dai, J. Li, X. Li, G. Zhai, and H. Guan. Improving virtualization performance and scalability with advanced hardware accelerations. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1 –10, dec. 2010.

[16] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 126–136, New York, NY, USA, 2007.

[17] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of i/o for gang scheduled workloads, 1997.

[18] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling algorithms. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 226–236, New York, NY, USA, 1990.

[19] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007.

[20] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *In Proc. of the 3rd International Conference on Distributed Computing Systems,*, Ft. Lauderdale, FL, USA, October 1982.

[21] A. Roytman, A. Kansal, S. Govindan, J. Liu, and S. Nath. Pacman: Performance aware virtual machine consolidation. In *10th International Conference on Autonomic Computing (ICAC)*. USENIX, June 2013.

[22] O. Sukwong and H. S. Kim. Is co-scheduling too expensive for smp vms? In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 257–272, New York, NY, USA, 2011.

[23] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[24] S. Vaddagiri, B. B Rao, V. Srinivasan, B. P Janakiraman Singh, and V. K Sukthankar. Scaling software on multi-core through co-scheduling of related tasks. In *in Ottawa Linux Symposium*, 2009.

[25] VMware. *VMWare vSphere 4: The CPU Scheduler in VMWare ESX 4 White Paper*. http://www.vmware.com/pdf/perf-vsphere-cpu-scheduler.pdf, 2010.

[26] VMWare. *Co-scheduling SMP VMs in VMware ESX server*, May,2008.

[27] VMWare. *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. "http://www.vmware.com/files/pdf/VMware-paravirtualization.pdf", May,2009.

[28] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 124–133, New York, NY, USA, 2006.

[29] C. Weng, Z. Wang, M. Li, and X. Lu. The hybrid scheduling framework for virtual machine systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 111–120, New York, NY, USA, 2009.