Scheduler Activations for Interference-Resilient SMP Virtual Machine Scheduling

Yong Zhao[1], Kun Suo[1], Luwei Cheng[2], and Jia Rao[1]
[1] *The University of Texas at Arlington, {yong.zhao, kuo.suo, jia.rao}@uta.edu*
[2] *Facebook, chengluwei@fb.com*

Abstract

The wide adoption of SMP virtual machines (VMs) and resource consolidation present challenges to efficiently executing multi-threaded programs in the cloud. An important problem is the semantic gaps between the guest OS and the hypervisor. The well-known lock-holder preemption (LHP) and lock-waiter preemption (LWP) problems are examples of such semantic gaps, in which the hypervisor is unaware of the activities in the guest OS and adversely deschedules virtual CPUs (vCPUs) that are executing in critical sections. Existing studies have focused on inferring a high-level semantic state of the guest OS to aid hypervisor-level scheduling so as to avoid the LHP and LWP problems.

In this work, we find a reverse semantic gap – the guest OS is oblivious of the scheduling events at the hypervisor, leaving the potential of addressing the LHP and LWP problems in the guest OS unexploited. Inspired by scheduler activations (SAs) in hybrid threading, we proposed interference-resilient scheduling (IRS), a guest-hypervisor coordinated approach to enhancing load balancing in the guest. IRS informs the guest OS before vCPU preemption happens at the hypervisor to activate in-guest load balancing. As such, critical threads on preempted vCPUs can be migrated to other running vCPUs so that the LHP and LWP problems are all alleviated. Experimental results with Xen and Linux guests show as much as 42%, 43%, and 46% performance improvement for parallel programs with blocking, spinning synchronizations, and multi-threaded server workloads, respectively.

CCS Concepts • **General and reference → Performance**; • **Software and its engineering → Operating systems**;

Keywords Virtualization, Cloud Computing, Multi-tenancy, Semantic Gaps.

1 Introduction

Symmetric Multiprocessing virtual machines (VMs) are becoming increasingly common in cloud datacenters. To fully utilize hardware parallelism, SMP VMs are often used by cloud users to host multi-threaded applications. On the other hand, cloud providers prefer oversubscribing their datacenters by consolidating multiple independent VMs onto a single machine to improve hardware utilization and reduce energy consumptions. For example, in desktop virtualization, VMware suggests a physical CPU (pCPU) can be shared by as many as 8 to 10 virtual desktops [35]. However, oversubscription requires that the CPU be multiplexed among multiple VMs so that each VM receives only a portion of the pCPU cycles.

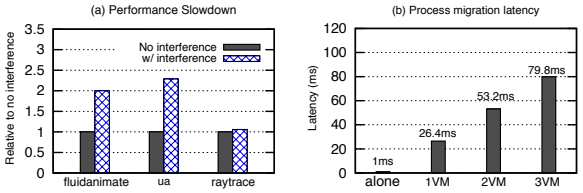

Figure 1. LHP and LWP cause significant slowdown to parallel programs. (a) Programs with user-level load balancing are more resilient to interference. (b) Existing load balancing in the guest OS is ineffective in addressing LHP and LWP.

CPU oversubscription introduces challenges to efficiently executing parallel and multi-threaded programs in SMP VMs. One well-known issue is the lock-holder preemption (LHP) problem [13]. LHP occurs when a vCPU is descheduled by the hypervisor while the thread currently running on that vCPU is holding an important lock. As the performance of parallel applications as a whole depends critically on the cooperation of multiple threads, if one thread holding the lock is preempted, other threads waiting for the lock are unable to make progress until the descheduled vCPU is rescheduled. Thus, the delay of one vCPU will significantly degrade the overall performance of the parallel program. Lock-waiter preemption (LWP) [3, 18, 31] is a similar problem in virtualized environments and can cause severe slowdown.

The root cause of the LHP and LWP problems is the semantic gap between the guest OS and the hypervisor. In virtualized environments, there exist two scheduling domains: (1) the guest OS schedules processes on vCPUs and (2) the hypervisor schedules vCPUs on physical CPUs. The scheduling activities in the guest OS are completely oblivious to the hypervisor. Thus, a vCPU can be preempted at any time by the hypervisor regardless of what this vCPU is executing. If vCPUs with threads waiting for entering into or already in the critical sections are preempted, LWP and LHP will occur respectively. There have been studies narrowing the semantic gap by inferring scheduling events inside VMs at the hypervisor using heuristics [1, 2, 17, 20, 22, 25, 26, 36], or approximating VM co-scheduling to mitigate the LHP problem [10, 21, 30, 32], or allowing the guest OS to assist hypervisor scheduling [9, 11, 13, 24, 34, 38]. These approaches have their respective limitations. Different workloads require distinct heuristics to identify thread criticality; co-scheduling is expensive to implement and causes CPU fragmentation [28]; synchronization-oriented optimizations make the hypervisor scheduling very complex and can possibly compromise fairness between VMs.

In this work, we identify another semantic gap, which is neglected in the literature – the guest OS is also unaware of the

Middleware '17, December 11–15, 2017, Las Vegas, NV, USA

ACM ISBN 978-1-4503-4720-4/17/12…$15.00
https://doi.org/10.1145/3135974.3135975

scheduling events at the hypervisor. If this gap is bridged, the guest OS can proactively migrate a critical thread if the host vCPU is preempted. We ran parallel programs in a 4-vCPU VM and slowed down one vCPU by co-locating another compute-bound VM with the vCPU to create interference. The interfered vCPU had frequent LHPs and LWPs. Figure 1 (a) shows the performance slowdown of three parallel application. `Fluidanimate` from the PARSEC benchmarks [33] and `ua` from the NPB benchmark [7] use blocking and spinning synchronization, respectively, and had significant slowdowns. In contrast, `raytrace` was resilient to LHP and LWP due to its user-level load balancing, which absorbed the slowdown by distributing work to threads having no interference.

Modern OSes are equipped with complex load balancing schemes to efficiently utilize multiprocessor systems. However, load balancing in the guest OS is not effective in virtualized environments. Process migration is the critical operation in load balancing. Figure 1(b) shows the latency of migrating a process in a Xen VM from a vCPU with frequent preemptions to another vCPU without interference. We measured the average latency of 30 migrations. The frequency of preemption was managed by placing different numbers of compute-bound VMs with the source vCPU of the migration. The reference migration latency was obtained when the VM ran alone and there were no vCPU preemptions. Figure 1 (b) suggests that process migration latency increases with the level of contention on the vCPU and the latency jump at each step corresponds to the VM scheduling delay (i.e., 30ms in Xen credit scheduler [8, 39]) incurred by adding one more VM. The results infer that load balancing in the guest OS is unable to address the LHP and LWP problem and itself is affected by vCPU preemptions.

There are two reasons why in-guest load balancing does not help mitigate LHP and LWP, both of which are due to the unawareness of hypervisor scheduling events in the guest OS. First, vCPU preemptions do not cause load imbalance in the guest, thereby the guest is unable to invoke process migration. Second, threads on preempted vCPUs are in "running" state, though they are actually not running, to the guest OS. As a result, the guest OS fails to migrate such "running" threads because it thinks it is unnecessary. Process migrations will only be successful until the preempted vCPU is scheduled again.

To fully unlock the potential of the guest OS in addressing the LHP and LWP problems, we design *interference-resilient scheduling* (IRS), a simple approach to bridging the guest-hypervisor semantic gap and guiding guest load balancing. Inspired by scheduler activations (SA) [4] in hybrid threading, IRS notifies the guest OS and activates in-guest load balancing when a vCPU is to be preempted by the hypervisor. As such, lock holder threads can be promptly migrated to other running vCPUs to avoid LHP and LWP.

We have implemented a prototype of IRS in Xen 4.5.0 and Linux 3.18.4, and performed comprehensive evaluations with various parallel and multi-threaded workloads. Experimental results show that IRS can improve the performance of NPB and PARSEC benchmarks by up to 43% and 42%, respectively, especially for programs with heavy synchronization. Moreover, IRS can reduce the latency of multi-threaded server workloads by as much as 46%.

The rest of the paper is organized as follows. Section 2 discusses previous work on the LHP and LWP problems and presents our motivation. Section 3 and 4 describe the design and implementation of IRS, respectively. Evaluation results and analysis with various parallel applications are given in Section 5. Section 6 discusses limitations and future work. We conclude this paper in Section 7.

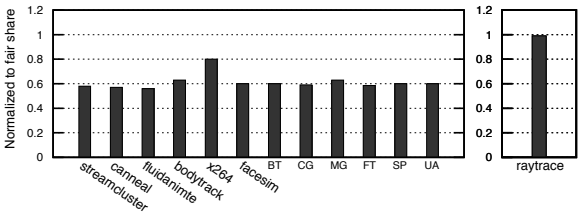

Figure 2. Parallel applications suffer low CPU utilization due to interference. User-level load balancing helps efficiently utilize CPU under interference.

2 Related Work and Motivation

Previous work attempting to eliminate this semantic gap can be divided into two categories: (1) Hypervisor-level approaches that treat the guest OS as a black box and (2) guest OS-assisted approaches employing para-virtualization. This section discusses these works and their limitations, followed by the motivation of our work.

2.1 Hypervisor Level Approaches

To deal with the LHP problem, VMware ESX 2.x [32] proposed the strict VM co-scheduling. This scheme allows vCPUs of the same SMP VMs to be synchronously scheduled and descheduled on different pCPUs. Despite its effectiveness in minimizing synchronization latency, it causes CPU fragmentation and vCPU priority inversion problems [28]. CPU fragmentation can lead to ineffective CPU utilization in environments where parallel applications are simultaneously hosted with sequential workloads [6, 12, 23, 27]. In order to lessen the severity of the CPU fragmentation problem caused by strict co-scheduling, relaxed co-scheduling [32] introduced in VMware ESX 3.x aimed to enable sibling vCPUs to make progress at similar paces and only requires the vCPUs that accrue enough skew to run simultaneously, while balance scheduling [30], a probabilistic co-scheduling scheme, increased the chance of co-scheduling sibling vCPUs by assigning them to different pCPUs. However, relaxed co-scheduling and balance scheduling distribute the sibling vCPUs across different pCPUs without considering the requirement for cooperative scheduling between vCPUs of the same VM and these two approaches still have LHP problem.

Demand-based coordinated scheduling [20] adopted TLB shootdown IPI and reschedule IPI between different vCPUs as heuristics to identify cooperative vCPUs and proposed urgent vCPU first scheduling to prioritize vCPUs that are handling critical threads in the VM. Passive inference of guest OS events at the hypervisor is not applicable to many workloads. For example, IPI-based heuristics are not effective in identifying critical threads for parallel workloads with spinning synchronization.

2.2 Guest OS-Assisted Approaches

To narrow the guest-hypervisor semantic gap, guest OSes are paravirtualized to coordinate with the hypervisor to avoid LHP. Dynamic adaptive scheduling [37] modified the guest OS to detect excessive spinning and report this information to the hypervisor. If a VM has reported frequent high spin waiting time, the hypervisor regards this VM as synchronization intensive and tries co-schedule its vCPUs as much as possible. Uhlig et al., [34] proposed a delay

preemption mechanism to minimize synchronization latency. Before a user-level thread acquires a spin-lock, the guest OS notifies the hypervisor of this pending event on the vCPU on which the thread is running. The notification requests that the vCPU not be preempted for a predefined period of time to avoid LHP.

The common issue of these guest OS-assisted approaches is that the guest OS is only responsible for passing down the information about lock holders and relies on the hypervisor to efficiently schedule vCPUs. As a result, the hypervisor needs to frequently deviate from its existing scheduling algorithm to utilize the semantic information for more efficient scheduling. Such invasive changes to hypervisor-level scheduling not only make hypervisor design more complex but can compromise VM fairness.

2.3 Motivation

As discussed above, existing work, either the hypervisor-level or guest OS-assisted approach, focused on making the hypervisor aware of the synchronization event inside the guest OS to aid scheduling. We show that there is a great potential of the guest OS to address the LHP problem.

Potential of guest OS load balancing We ran representative parallel benchmarks from the PARSEC and NBP benchmark suites in a 4-vCPU Xen VM. The LHP and LWP problems were caused by placing another CPU-bound VM with one of the 4 vCPUs. The vCPUs of the parallel VM and the interfering VM were pinned to separated pCPUs. The vCPU that experienced LHP and LWP shared the same pCPU with the interfering VM. The PARSEC benchmarks were compiled with `pthreads` and NBP benchmarks were compiled using `OpenMP` with `OMP_WAIT_POLICY` set to passive. All benchmarks used blocking synchronization.

Figure 2 shows the CPU utilization of the parallel VM relative to its fair share. Ideally, both VMs should receive the fair share of the pCPU capacity. As shown in the figure, all parallel programs except `raytrace` suffered much lower CPU utilizations compared to their fair shares, indicating that the parallel VM did not fully or efficiently utilize its CPU entitlement. The culprit is that the interfering VM caused frequent LHPs and LWPs to the parallel VM. If a critical thread is preempted, all other threads need to wait for the critical section until the preempted vCPU is rescheduled. With blocking synchronization, the waiting threads are put to sleep and their host vCPUs become idle even when there are sufficient pCPU allocated to the parallel VM.

Programs with spinning synchronization suffer similar performance degradation due to LHP, but do not show low CPU utilizations. Instead of going idle, the waiting vCPUs busily wait on the lock and burn CPU cycles. Although the parallel VM is able to utilize its fair share, most CPU cycles are spent on spinning and few are used to carry out meaningful computation. Hardware-based techniques, such as pause loop exiting (PLE) [14], detect excessive spinning and stop a VM to prevent it from wasting CPU cycles. The effect is equivalent to blocking-based synchronization and the parallel VM will suffer low CPU utilization in the presence of LHP.

In contrast, Figure 2 also shows that `raytrace` was able to fully use its fair share even in the presence of LHP and LWP. This explains its resilience to interference as shown in Figure 1. `Raytrace` implements a work-stealing mechanism at user level and threads that complete their assigned work sooner steal the work originally assigned to slower threads. As such, interference has less impact on the overall performance as meaningful work is migrated to faster or interference-free threads/vCPUs. Similar interference resilience can also be observed in programs compiled with Intel TBB [15] and `OpenMP` using a dynamic thread schedule.

This motivating example demonstrates that load balancing can effectively address the LHP and LWP problems and mitigate the slowdown caused by interference. However, only programs that have specific compiler support or have their own user-level load balancing are resilient to interference. Programs relying on the guest OS, e.g., Linux, for load balancing suffer low CPU utilizations and significant performance slowdowns. In general, there are two approaches in guest OS load balancing: *push migration* and *pull migration*. Push migration periodically checks **load imbalance** and pushes threads from busy to less-busy vCPUs; pull migration occurs when a vCPU becomes idle and steals (or pulls) **excessive** work or ready (but not running) threads from a busy vCPU. Both approaches fail to work effectively in virtualized environments. First, the load imbalance at the hypervisor does not lead to imbalance in the guest OS and push migration is not invoked. Second and most importantly, threads on preempted vCPUs are not considered excessive work by the pull migration as they are in the "running" state.

Issues with hypervisor load balancing Hypervisors also implement complex schemes for balancing vCPUs among pCPUs. Hypervisor level load balancing falls short of addressing the LHP and LWP problems in two ways. First, lacking the information on thread criticality, the hypervisor is unable to precisely identify the vCPU that experiences LHP and LWP. Second, the hypervisor treats vCPUs from different VMs equally and relies purely on the computational load on pCPUs for load balancing. Thus, it is possible that hypervisor places vCPUs from the same VM onto the same pCPU to attain better load balance, thereby causing the CPU stacking problem [30]. Our experimental results show that CPU stacking can incur 10-20x performance degradation to PARSEC benchmarks when the parallel VM and the interfering VM shared the same set of 4 pCPUs but all vCPUs were unpinned. The same issue can also be observed in other hypervisors, such as KVM [19] and VMware [32].

Summary Parallel programs suffer significant performance loss due to LHP and LWP and so are unable to efficiently utilize their CPU allocations. Effective load balancing of parallel threads can greatly alleviate the LHP and LWP problems. These observations motivated us to enhance the guest OS load balancing in virtualized environments so as to make any workload resilient to interference. To this end, we design *interference-resilient scheduling* (IRS), a simple approach to bridging the guest-hypervisor semantic gap and unlocking guest OS load balancing.

3 IRS Design

IRS is a coordinated approach that bridges the guest-hypervisor semantic gap at the guest OS side. The objective is to enhance the guest OS load balancing to make parallel programs resilient to interference between VMs, thereby mitigating the LHP and LWP problems. The heart of IRS design is the mechanism of *scheduler activations* (SA) in response to vCPU preemptions at the hypervisor. Inspired by the classical scheduler activation approach in hybrid threading, in which the OS kernel notifies the user-level scheduler if a user-level thread blocks in the kernel so that the user-level scheduler can pick another ready user thread to execute. Similarly, IRS informs the guest OS once its vCPU is to be preempted. The

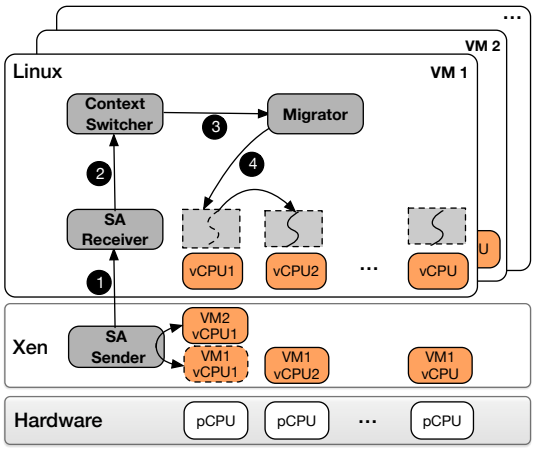

Figure 3. The architecture of IRS.

guest OS then migrates the thread running on the preempted vCPU to another running vCPU to avoid LHP and LWP.

Figure 3 shows the architecture of IRS in a Xen environment. There are four components in IRS: *SA sender*, *SA receiver*, *context switcher* (CS), and *migrator*. Before Xen preempts a vCPU, it sends a notification to the vCPU via SA sender residing in Xen (step ①). Upon receiving the notification, SA receiver in the guest starts the load balancing process (step ②). To enable task migration, the CS deschedules the thread on the preemptee vCPU and marks the thread as migrating (step ③). Last, the migrator moves the thread to a sibling vCPU with least waiting time (step ④). Next, we elaborate on the design of these components in the context of Xen and Linux guests.

3.1 SA Sender and Receiver

SA sender and receiver together establish a communication channel between the hypervisor and the guest OS. Algorithm 1 shows the interactions between the SA sender and receiver. SA sender is on the critical schedule path of the hypervisor. Whenever the hypervisor decides to preempt a current running vCPU, it sends a notification to the preemptee vCPU to allow the guest OS to respond to the preemption. Only vCPUs that are involuntarily preempted and are still willing to run (i.e., `runnable`) will be notified (line 4-5). To avoid duplicate notification, the SA sender also needs to check if there is an SA notification pending for a vCPU in the guest OS. The SA notification is per-vCPU. After the notification is sent, the hypervisor delays the preemption and allows the preemptee vCPU to continue running and process the notification (line 7).

The SA receiver resides in the guest OS and takes three steps to respond to the SA notification: (1) deschedule the current running task on the preemptee vCPU and perform a context switch (line 12). The return value of the context switcher determines the response to the hypervisor; (2) asynchronously wake up the migrator thread to move the descheduled task to a different vCPU (line 13); (3) return the control back to the hypervisor (line 15). Once the hypervisor receives the response, it clears the SA pending flag of the vCPU to enable the next round of SA.

The hypervisor-guest communication uses Xen's event channel for SA notification. To ensure timely delivery of the SA, we design the notification as a virtual interrupt (vIRQ) for the guest OS. The SA receiver is essentially the interrupt handler of the new

Algorithm 1 Send and acknowledge SA event.

```
1:  Variables: The vCPU to be preempted v; The SA acknowledgement sent by the guest OS ops.
2:  /* Hypervisor: send SA event */
3:  procedure SEND_SA_EVENT(v)
4:      if vcpu_runnable(v) and sa_pending(v) then
5:          send_guest_vcpu_virq(v_c, VIRQ_SA_UPCALL)
6:          set_sa_pending(v)
7:          return continue_running(v)
8:      end if
9:  end procedure
10: /* Guest OS: acknowledge SA completion */
11: procedure ACK_SA_EVENT(void)
12:     ops = context_switcher()
13:     wake_up_migrator()
14:     /* Return the control back to hypervisor and clear SA pending flag on the host vCPU */
15:     HYPERVISOR_sched_op(ops, NULL)
16: end procedure
```

vIRQ. Note that one change to hypervisor level scheduling is necessary for enabling SA – any vCPU preemption needs to be delayed until the guest OS completes the processing of SA. This change may affect existing scheduling in the hypervisor, such as fairness and I/O prioritization. To minimize the impact, the SA receiver should complete fast. The context switching of the current running task should be performed on the preemptee vCPU and the vCPU needs to be active. Once the context switch is done, the migrator is asynchronously invoked and can run on other vCPUs. Thus, the required delay at the hypervisor only includes the time to handle the vIRQ and perform one task context switch in the guest. Our profiling suggests that IRS adds 20-26 μs delay to the hypervisor scheduling. Since the time slice of hypervisor scheduling is in the granularity of milliseconds, e.g., 30ms in Xen, 6ms in KVM, and 50ms in VMware, the delay is negligible from the perspective of fair CPU allocation. However, if vCPU preemption is due to prioritizing an I/O-bound vCPU, the delay will add to I/O latency.

3.2 Context Switcher

The purpose of the context switcher is to faithfully reflect the status of a vCPU in the guest OS to bridge the semantic gap. For example, if a vCPU is preempted and put back to the runqueue of a pCPU, the task currently running on the vCPU in the guest OS should also be descheduled. After a context switch, the vCPU should be put into a proper state so as not to affect hypervisor-level scheduling. In Xen, vCPUs are in one of the following three states: `running`, `runnable`, and `blocked`. While `running` means a vCPU is executing on the pCPU, `runnable` indicates that the vCPU has been preempted but it has a task to run. If a vCPU is idle or waiting for I/O completion, it has no tasks to run and will be put in the `blocked` state. Xen devises different scheduling policies for different vCPU states. For example, a vCPU waking up from a `blocked` state will be considered latency sensitive and be prioritized.

To preserve the scheduling policy at the hypervisor, the SA receiver should respond differently to the hypervisor depending on the execution state of the vCPU after task context switch. The

Algorithm 2 Migrate task from preempted vCPU.

```
1:  Variables: The task to be migrated p; the least loaded vCPU
    v_min in the guest OS; the state of a vCPU s; the runqueue of a
    vCPU rq_v.
2:  /* Guest OS: migrate task to least loaded vCPU */
3:  procedure MIGRATE_TASK(p)
4:      v_min = NULL
5:      for each online vCPU v do
6:          rq_min = rq_of(v_min)
7:          s = get_vcpu_runstate(v)
8:          if s == IDLE then
9:              v_min = v
10:             break
11:         end if
12:         if s == RUNNING then
13:             rq_v = rq_of(v)
14:             if rq_v.rt_avg <rq_min.rt_avg then
15:                 v_min = v
16:             end if
17:         end if
18:     end for
19:     if v_min ≠ NULL then
20:         __migrate_task(p, v_min)
21:         return SUCCESS
22:     else
23:         return FAIL
24:     end if
25: end procedure
```

context switcher returns different operations to the SA receiver (Algorithm 1, line 12). If there is no runnable task left in the runqueue of the vCPU after the current running task is descheduled, the `idle` task will be put on the vCPU and the context switcher returns operation `SCHEDOP_block`. In contrast, if there are other runnable tasks, the vCPU should be put in the `runnable` state in hypervisor. In this case, context switcher returns operation `SCHEDOP_yield`, which does not change the vCPU state but simply yields to the hypervisor.

3.3 Migrator

The migrator is responsible for distributing the descheduled task from a preempted vCPU to another running vCPU so that the task does not need to wait for the original vCPU to be scheduled so as to run. If the descheduled task is a lock holder or lock waiter and is scheduled sooner due to load balancing, the LHP and LWP problems are alleviated. As discussed in Section 2.3, load balancing in the guest OS is not effective due to the two semantic gaps: load imbalance at hypervisor does not trigger load balancing in guest OS; task migration does not apply to "running" tasks even though the host vCPU is preempted. The context switcher addresses the second gap by descheduling the task upon vCPU preemption. The migrator bridges the first gap.

Since the guest OS, e.g., Linux, has implemented complex load balancing schemes, we design the migrator to be minimally intrusive to the existing balancing algorithm. Since the guest load balancer is unable to sense the load imbalance at the hypervisor, the migrator does not consider the load balance in the guest and

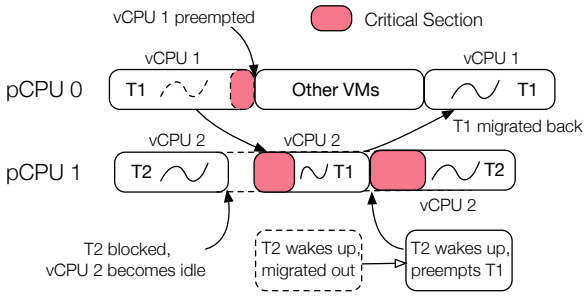

Figure 4. Pingpong migration caused by IRS and a simple approach to preserve locality.

forcibly move the descheduled task to a different vCPU. The goal is to migrate the task to the least loaded vCPU. Algorithm 2 shows how to find the least loaded vCPU. The migrator iterates over all online vCPUs of the guest OS until it finds a target vCPU for migration. Note that preempted vCPUs also appear to be "online" to the guest OS. Therefore, the migrator needs to call down to the hypervisor to check the actual vCPU state (line 7). Ideally, the migrator finds an idle sibling vCPU and the search will end as the task can run immediately on this vCPU (line 8-10).

If there are no idle vCPUs, the migrator tries to find the least loaded vCPU (line 12-17). As there are two levels of load balance in virtualized systems, i.e., the balance in the guest OS and in the hypervisor, the migrator aims to find a lightly loaded vCPU, which not only has few tasks on the vCPU runqueue in the guest but also experiences little contention from other VMs on the pCPU. We rely on the real time estimate of runqueue load (i.e., `rt_avg`) in Linux to measure vCPU busyness. The metric `rt_avg` considers the weighted process load in the guest OS as well as the contention on the pCPUs. It uses `steal` time, which measures the time during which a vCPU is runnable but unable to run due to contention, to quantify hypervisor-level CPU contention. The migrator compares vCPUs using the `rt_avg` of their runqueues to pick the least loaded vCPU.

Another challenge in designing the migrator is to ensure that load is balanced between sibling vCPUs when preempted vCPUs come back online. To minimize intrusive changes to the guest OS, the migrator relies on the existing load balancer in Linux to move tasks back to the rescheduled vCPU. However, one drawback of task migration is the loss of cache locality. The migrator aims to preserve cache locality as much as possible. Besides the push and pull migrations in Linux, there is another scenario in which task migration is necessary and related to parallel programs. For workloads with blocking synchronization, such as `pthread mutex` and `barrier`, the Linux kernel checks load balance when waking up waiting threads, e.g., lock waiters. If the vCPU where the waiting thread slept on is running another task, the waking task is migrated to a different vCPU. Figure 4 illustrates this problem and the migrator's simple solution.

As shown in Figure 4, when vCPU-1 is preempted, task-1 (T1) is migrated to idle vCPU-2, on which task-2 (T2) is blocked and waiting for the lock held by T1. Once T1 releases the lock, T2 is woken up. Because T1 is currently running on vCPU-2, T2's the host vCPU, T2 will be migrated out, likely to vCPU-1 as it is idle now. This design is to avoid unnecessary preemptions of a running task if there exist idle vCPUs. However, the wake up balancing causes pingpong migrations between vCPUs, which leads to poor

cache locality. Waking tasks are frequently migrated away from their original vCPU because the migrator distributes tasks from preempted vCPUs to idle vCPUs.

The migrator employs a simple approach to address this issue. Instead of migrating the waking task, the wakeup load balancer in Linux is modified to check the status of the current running task to determine if the waking task should preempt the current task. The migrator tags each task that is migrated due to preempted vCPU. If the current running task is tagged, the wakeup balancer allows the waking task to preempt the current task. The dotted box in Figure 4 shows the original Linux design and the arrow points to the new design. This simple solution guarantees that waiter tasks always wake up from their host vCPU to preserve locality. We rely on the Linux load balancer to migrate the tagged task back to the preempted vCPU when it is scheduled again. This design only applies to blocking workloads. For spinning workloads, the Linux balancer will migrate the tagged task back to its original vCPU as its runtime on the new vCPU is short and it is not "cache hot".

4 Implementation

We have implemented IRS in Xen 4.5.0 and Linux 3.18.4. We intend to make the changes to the hypervisor and guest OS minimally intrusive and use existing scheduling and load balancing primitives. IRS requires small changes to Xen (less than 30 lines of code) and Linux guest kernel (about 130 lines of code). Next, we describe the modifications to Xen and Linux in detail.

4.1 Modifications to Xen Hypervisor

For SA notification, we add a new virtual interrupt `VIRQ_SA_UPCALL` in Xen and use a dedicated event channel for SA communications between Xen and the guest OS. The credit scheduler in Xen is modified to temporarily delay the preemption of vCPUs until the guest OS acknowledges the completion of SA. Once Xen relinquishes the control of the vCPU scheduling, it relies on the guest OS to respond to the SA notification and return the control back to Xen. This may create security issues if malicious guests never return to the hypervisor. As discussed in Section 3.1, SA processing typically takes 20-26 μs, so the hypervisor can set a hard limit for SA completion to prevent rogue users from exploiting SA.

4.2 Modifications to Linux Guest OS

The main functionalities of IRS are implemented in the guest OS. We implement SA receiver as the interrupt handler of the new `VIRQ_SA_UPCALL` interrupt. Since interrupt handlers should be kept small, SA receiver delegates the SA response to Xen to the context switcher. We implement the context switcher as the bottom half of the `VIRQ_SA_UPCALL` vIRQ. We create a new softirq called `UPCALL_SOFTIRQ` in the guest OS and assigned the context switcher as its handler. In Linux, softirqs have different priorities. We set the `UPCALL_SOFTRIQ` to a lower priority than the `TIMER_SOFTIRQ`, which is responsible for handling periodic timer events because the Linux kernel relies critically on timer interrupts to perform task scheduling. When timer interrupt and SA interrupt arrive at the same time, we ensure that the timer interrupt, which may trigger task switching in the Linux scheduler, is handled prior to the SA interrupt. This is to prevent tasks that were to be descheduled at the timer interrupt from being migrated.

The context switcher uses existing scheduling primitives in Linux to pick the next task when the current running task is descheduled. After the context switch is completed, it asynchronously invokes the migrator to distribute the descheduled task to another vCPU for load balancing. Before the migration is performed, the context switcher calls hypercall `HYPERVISOR_sched_op` with either `SCHEDOP_block` or `SCHEDOP_yield` as the command to return control to Xen. The migrator is implemented as a system-wide kernel thread. It borrows the idea from existing migration function `migration_cpu_stop` but need not require to run on the vCPU from where the task is migrated. This greatly shortens the amount of time the preemptee vCPU needs to be active, thereby reducing the delay at the hypervisor scheduler. The migrator probes the runtime states of vCPUs via the hypercall `HYPERVISOR_vcpu_op` to determine the least loaded vCPU for migration. If a target vCPU is found, the migrator invokes function `__migrate_task` to migrate the task.

5 Evaluation

In this section, we present an evaluation of IRS using various parallel and multi-threaded workloads. We study the effectiveness of IRS in improving the performance of various parallel workloads with different types of synchronization (§ 5.2). We then extend the evaluation to multi-threaded workloads with little synchronization (§ 5.3). We also investigate how well IRS improves overall system efficiency when consolidating multiple parallel workloads (§ 5.4) and perform a scalability and sensitivity analysis of IRS in response to various levels of interference (§ 5.5). Finally, we study the potential of IRS in mitigating the vCPU stacking problem (§ 5.6).

5.1 Experimental Settings

Our experiments were performed on a DELL PowerEdge T420 server, equipped with two six-core Intel Xeon E5-2410 1.9GHz processors, 32GB memory, one Gigabit Network card, and a 1TB 7200RPM SATA hard disk. We ran Linux kernel 3.18.4 as the guest and dom0 OS, and Xen 4.5.0 as the hypervisor. We created two VMs, each configured with 4 vCPUs and 4GB memory. One VM was used to run parallel and multi-threaded workloads and the other was the interfering VM. We enabled para-virtualized spin-locks in the guest kernel but it had no effect on NPB performance as OpenMP uses its user-level spin implementation.

CPU pinning We first created a controlled environment to study the benefit of IRS by disabling vCPU load balancing at the hypervisor. Both VMs were set to share four cores in one of the two processors. Each vCPU is pinned to a different pCPU. Thus, two vCPUs from the two VMs share the same pCPU. Note that if vCPUs were unpinned, VM oblivious load balancing at the hypervisor causes CPU stacking problem and incurs significant performance degradation and unpredictability to parallel workloads. In Section 5.6, we evaluated IRS performance in an unrestricted environment with all vCPUs unpinned.

Workloads We selected the following workloads and measured their performance with IRS and three representative scheduling strategies.

- *PARSEC* [33] is a shared memory parallel benchmark suite. We compiled the benchmarks using `pthread` and used the native

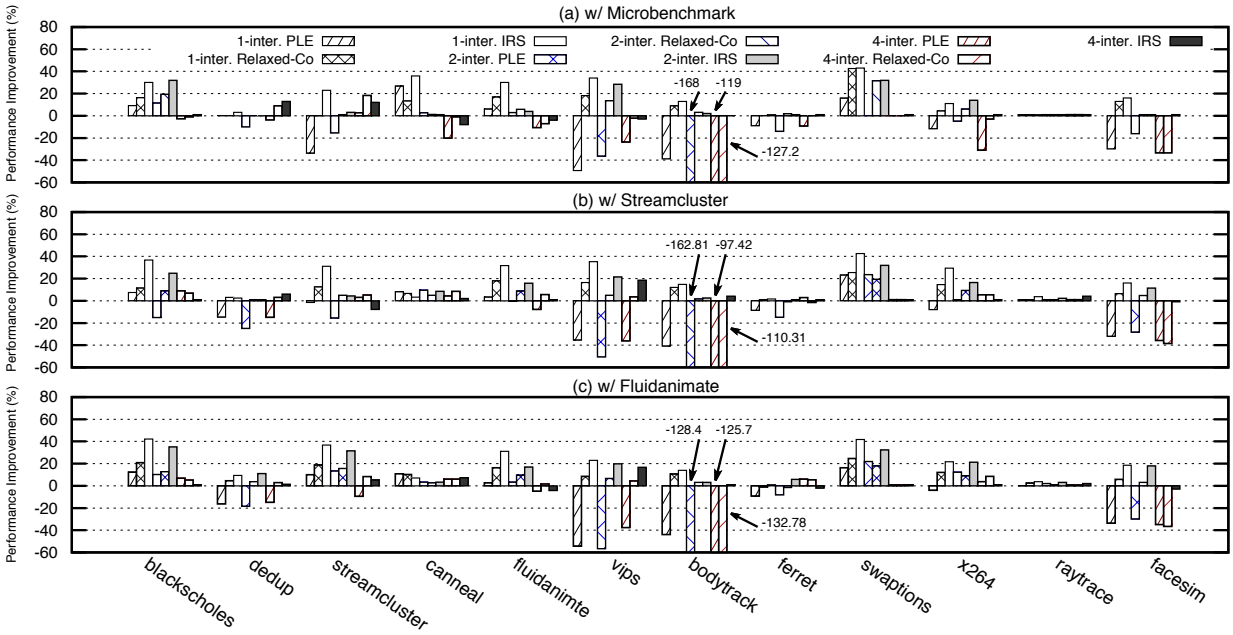

Figure 5. Improvement on PARSEC performance (blocking).

input for all benchmarks. PARSEC benchmarks use various blocking synchronization primitives (e.g., mutexes, condition variables and barriers).

- *NASA parallel benchmarks (NPB)* [7] include 9 parallel programs derived from computational fluid dynamics applications. We used the OpenMP implementation of the benchmarks and set the problem size to class C. Environment variable `OMP_WAIT_POLICY` was set to active to enable spinning synchronization between threads.
- *SPECjbb2005* [29] is a multi-threaded client/server benchmark. Synchronization is occasionally needed when customer requests compete for the same database table. Performance is measured by the throughput of the server, e.g., business operations per second (bops) and the latency of the common request type, e.g., the new order transaction.
- *Apache HTTP server benchmark* [5] stress tests the throughput and latency of a webserver using a large number of requests. Threads servicing client requests are independent and do not require synchronizations.

Interfering workloads We used two types of interfering workloads to create contention between VMs. We first used a micro-benchmark to generate synthetic interference. The micro-benchmark consisted of a varying number of CPU hogs that compete for the CPU cycles and had almost zero memory footprint. The use of the micro-benchmark is to perform controlled experiments that has persistent interference to the workloads under test. In addition to the micro-benchmark, we also co-located PARSEC and NPB benchmarks with two realistic background interfering workloads respectively. `streamcluster` and `ua` have fine-grained synchronizations at the granularity of 20-30ms and 1-2s while `fluidanimate` and `lu` have coarse-grained synchronizations every 6 and 30 seconds.

Scheduling strategies We compare the performance of IRB with three state-of-the-art scheduling strategies for parallel programs.

- `Xen`: we used the default credit scheduler without any optimizations for parallel programs as the baseline.
- `PLE`: pause-loop exiting is a hardware-level mechanism for spin detection. It detects the execution of excessive `PAUSE` instructions, which are commonly found in spin lock implementation, and causes trapping (via `VM-exit`) into the hypervisor. In Xen, the credit scheduler switches to a different vCPU if the current vCPU is stopped by PLE. To enable PLE, all workloads were run in hardware-assisted virtualization (HVM) VMs.
- `Relaxed-Co`: we implemented VMWare's relaxed co-scheduling in Xen. `Relaxed-Co` monitors the execution skew of each vCPU and stops the vCPU that makes significantly more progress than the slowest vCPU. A vCPU is considered to make progress when it executes guest instructions or it is in the `IDLE` state. Since VMWare documentation does not reveal further details about relaxed co-scheduling, we implemented an optimization for parallel programs – when a VM's leading vCPU is stopped, the hypervisor switches it with its slowest sibling vCPU to boost the lagging vCPU.

5.2 Improving Parallel Performance

In this section, we evaluate the effectiveness of IRS in improving parallel performance for various parallel workloads. All benchmarks were run with 4 threads, matching the number of vCPUs in the VM. The results were the average of 5 runs.

Figure 5 and 6 show the performance improvement due to IRS for PARSEC and NPB. Performance improvement is relative to the vanilla Xen and Linux. We varied the level of interference (denoted as *1-inter.*, *2-inter.*, and *4-inter.*) and caused LHP and LWP problems on different numbers of vCPUs of the parallel VM. For example, *2-inter.* refers to the scenario in which either two CPU hogs or 2-thread real applications compete for CPU cycles with two vCPUs of the parallel VM on two pCPUs. PARSEC includes a wide spectrum of parallel programs with different synchronization primitives, (e.g., mutexes, condition variables, and barriers), parallel programming models, (e.g., data parallel and pipeline parallel), and task assignment polices, (e.g., static and dynamic assignments).

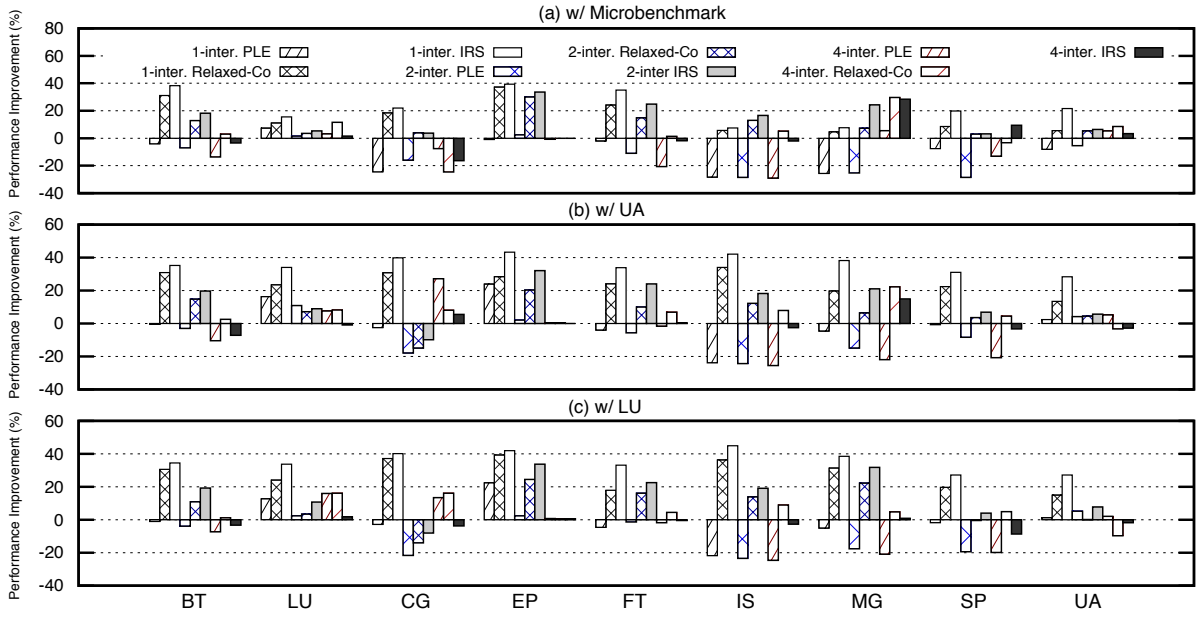

Figure 6. Improvement on NPB performance (spinning).

Figure 5 shows the effectiveness of IRS for all PARSEC benchmarks. We have the following **observations** about IRS performance:

First, most PARSEC benchmarks benefited from IRS with as much as 42% improvement over vanilla Xen/Linux. However, IRS was not quite effective for some workloads with marginal improvement, i.e., `dedup`, `ferret`, and `raytrace`. `Dedup` and `ferret` employ pipeline parallelism and use multiple threads (i.e., 4 threads) for each pipeline stage (4 stages in `dedup` and 5 stages in `ferret`). Thus, there were multiple threads running on each vCPU. The Linux scheduler was able to balance these threads as most threads will be in the ready state, leaving little room for performance improvement. Similarly, `raytrace` implements user-level load balancing and does not need much help from IRS.

Second, performance improvement decreased as the level of interference increased. While IRS had significantly improved performance for the *1-inter.* and *2-inter.* cases, it can degrade performance in the *4-inter.* case. When a few vCPUs were under interference, IRS was able to migrate threads onto vCPUs without interference. The more interference-free vCPUs, the more likely for IRS to find idle vCPUs that can run migrated threads immediately. In contrast, when all vCPUs were under interference, the vCPU onto which a thread was migrated can be preempted soon, which triggers another round of migration. Frequent migration violates cache locality and may incur performance degradation, especially for memory-intensive workloads. This **overhead** explains the slowdown of some programs under IRS in the *4-inter.* case.

Third, IRS was also effective when interferences were real parallel workloads. The results were similar to those with the synthetic interference except that IRS had slightly better performance in the *4-inter.* case. When the interference was a real parallel program, it demanded less CPU than the synthetic interference because the interfering workload also suffered from LHP or LWP, thereby having low CPU utilizations.

Compared to IRS, `PLE` and `Relaxed-Co` had improved parallel performance to a certain extent, though not as much as IRS in most cases, but incurred considerable performance degradation to some workloads. Since blocking primitives, such as mutex and condition variable, only spend a very short period of time spinning when performing wait queue operations, `PLE` does not help much on preventing excessive spinning. As shown in Figure 5, `PLE` had limited performance improvement for `blackscholes` and `swaptions` but incurred considerable slowdown to `vips`, `bodytrack`, and `facesim`. The reason is that `PLE` avoids futile spinning but does not prevent LHP or LWP from occurring. When a spinning vCPU is stopped by `PLE`, the vCPU from the competing VM will be scheduled. Currently, there is no mechanism in Xen for prioritizing the siblings, which are likely the lock holder or waiter, if a spinning vCPU yields CPU due to `PLE`. This explains why for some workloads, `PLE` caused slowdown.

In contrast, `relaxed-Co` is specially designed to balance the progress of sibling vCPUs. In our implementation, we monitored the progress of all sibling vCPUs belonging to the same VM in every accounting period in Xen (every 30ms) and stopped the leading vCPU to boost the most lagging vCPU. However, results in Figure 5 show that it attained less performance improvement compared to IRS in almost all PARSEC workloads. The results also suggest that `relaxed-Co` can be destructive, especially in the *4-inter.* cases. For example, it caused more than 132% performance degradation for `bodytrack` in Figure 5 (c). Overall, `relaxed-Co` was less effective or even destructive for blocking workloads than spinning workloads (shown in Figure 6). The culprit was that the idle period during which a blocking workload waits for synchronization is considered as making progress by `relaxed-Co`, thereby not counted as skew. As will be discussed in § 5.6, the idleness caused by out of synchronization is not recognized by existing CPU schedulers, which constitutes the main limitation of `relaxed-Co`.

For spinning workloads, the migrator in IRS was unable to find any idle vCPUs to migrate preempted threads as threads never block when waiting for synchronization. IRS can only find vCPUs that are less loaded for migration. Thus, the migrated thread inevitably needs to share the destination vCPU with another thread. Counter-intuitively, Figure 6 shows that on average IRS attained higher

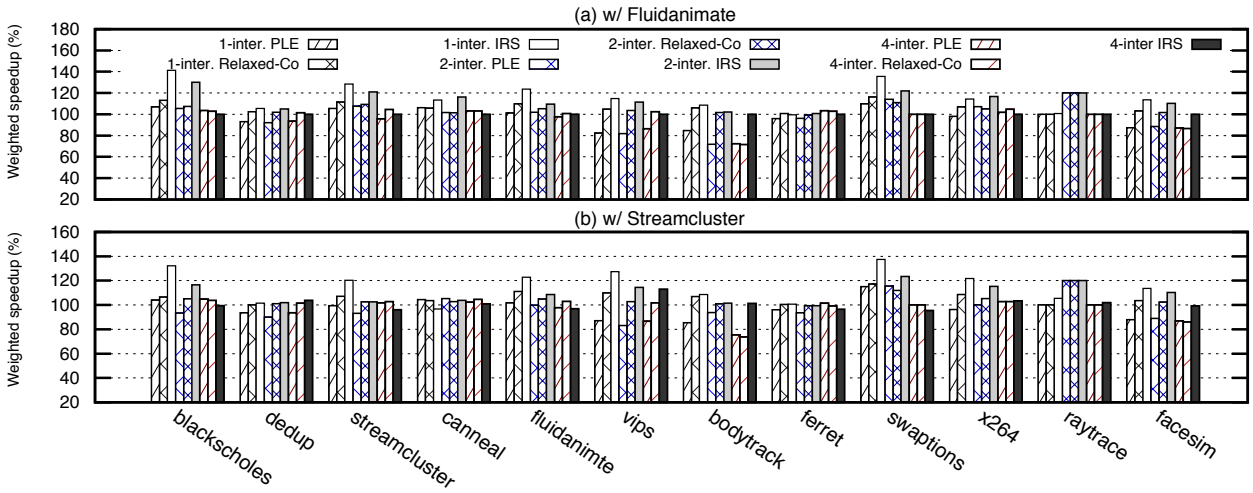

Figure 7. Weighted speedup of two PARSEC applications (blocking, higher is better).

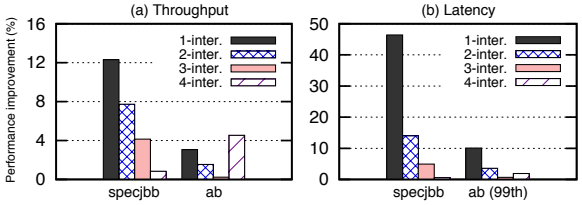

Figure 8. Improvement on server throughput and latency.

performance improvement over the baseline. In vanilla Xen/Linux, a preempted thread needs to wait one time slice in Xen, i.e., 30ms, before its host vCPU is scheduled again, leading to long lock wait time. In contrast, IRS migrates the lock holder thread to another vCPU. Although the thread still needs to wait until it is scheduled by the guest OS, the scheduling happens much sooner. Not only does Linux guest OS use finer grained time slices (i.e., 6ms), but also the migrated task likely has smaller virtual runtime than the existing task on the destination vCPU and would be prioritized by Linux completely fair scheduler (CFS). However, a similar trend was observed – the performance gain due to IRS diminished as interference ramped up.

`PLE` and `relaxed-Co` were more effective for spinning workloads than blocking workloads. In most cases, they achieved close but less improvement compared to IRS. Nevertheless, they still performed poorly for some workloads, e.g., `CG`, `IS`, `MG`, and `SP`. In contrast, although IRS can cause slowdowns, the degree of degradation is not as much as `PLE` and `relaxed-Co`. It clearly demonstrates the weakness of hypervisor-level scheduling optimizations – it is challenging to design a "one-size-fits-all" approach for different types of workloads. An optimization effective for one workload could be harmful to other workloads.

The improvement on parallel performance was mainly due to much improved CPU utilization under interference. IRS was able to boost the utilization of parallel workloads close to their fair share under CPU contention. The enhanced load balancing in the guest OS helped parallel workloads utilize the idle or wasted CPU cycles in vanilla Xen/Linux.

5.3 Improving Multi-threaded Performance

We have shown that IRS is effective for boosting various parallel workloads. In this subsection, we study its performance with more general multi-threaded programs with little or no synchronization. We show that these workloads can also benefit from IRS. We used two different server benchmarks. *SPECjbb2005* executes complex business transactions and its request processing time are in the range of a few hundreds of millisecond to a few seconds on our testbed. We set the number of warehouses to 4 so that there was a one-to-one mapping between threads and vCPUs. *Apache benchmark* (`ab`) tests webserver performance using a large number of threads, each requesting a small file from the server. We set the number of connections to 1000 and `MaxClient` in Apache `httpd` to 512. Thus, there were 512 concurrent threads in the webserver.

Figure 8 shows the improvement in throughput and latency due to IRS relative to vanilla Xen/Linux. Since request processing in server workloads has little dependency and requires little synchronization, `PLE` and `relaxed-Co` have little effect and their results are not reported. The interference was one to four CPU hogs. Since SPECjbb performs little synchronization and `ab` had no synchronization, their CPU utilizations can achieve the fair share and IRS does not improve utilization as it did for the parallel workloads above. However, as shown in Figure 8, IRS was still able to improve the throughput of *SPECjbb* by up to 12%, though did not help much in `ab` (by as much as 4%). While IRS did not help increase utilization, it did improve request latency, which contributed to throughput improvement. Figure 8 (b) shows that the average latency of the new order transaction in *SPECjbb* was improved by as much as 46%. *SPECjbb* measures throughput based on the number of requests that satisfy a service level objective (SLO) on latency. Thus, improvement on latency avoided many SLO violations and increased effective throughput.

In contrast, IRS had marginal improvement on `ab` latency. Figure 8(b) shows that there was only slight improvement on the tail latency (99^{th} percentile) of `ab`. The average latency (not shown in the figure) was not much improved. The difference between `ab` and *SPECjbb* is that `ab` had many more threads than the number of vCPUs and each request was short. Since Linux is able to sense the contention at the hypervisor by dynamically updating the `rt_avg` load on each vCPU, the load balancer in the guest OS was able

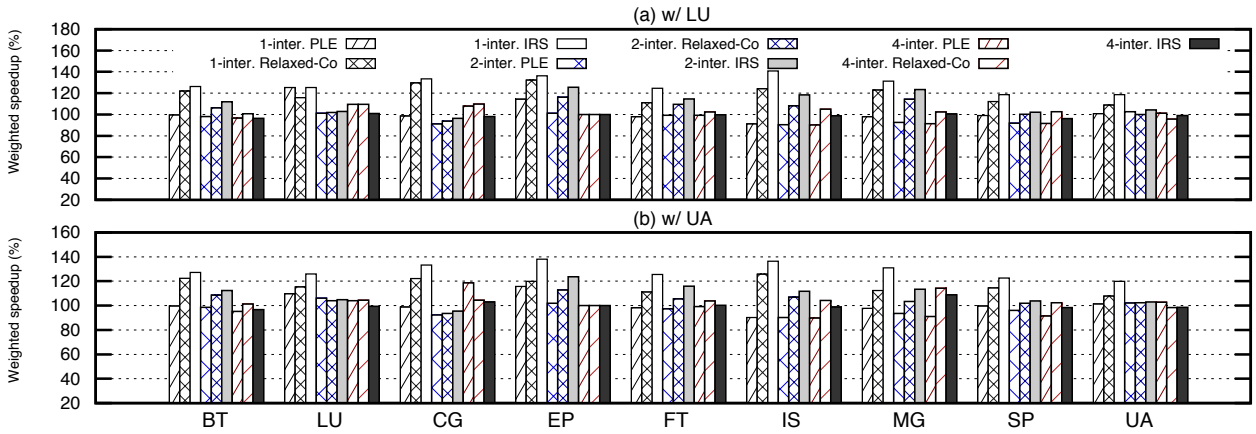

Figure 9. Weighted speedup of NPB applications (spinning, higher is better).

to distribute threads on vCPUs based on the level of interference experienced by each vCPU. Therefore, IRS can only help the thread that was running when its host vCPU was preempted. Given that ab had a large number of threads, improvement on a few threads did not contribute much to the overall throughput but helped the tail latency to some extent.

This experiment demonstrates that for multi-threaded workloads with little/no synchronization, IRS is most effective for workloads that have an equal number of or fewer threads than the number of vCPUs because IRS can migrate "running" threads from preempted vCPU. For workloads with many more threads than the number of vCPUs, load balancing in Linux is sufficiently effective, leaving no room for IRS improvement. Similarly, as interference ramps up, the performance gain due to IRS diminishes.

5.4 System Fairness and Efficiency

The objective of IRS is to allow the guest OS to more efficiently utilize its CPU allocation via enhanced in-guest load balancing. Since it requires some changes to both the guest OS and the hypervisor, we are interested in studying the system-wide fairness and efficiency when multiple realistic applications are co-located. The foreground VM ran various parallel workloads and had IRS enabled. We selected representative parallel programs as the interfering workloads running in the background VM. The interfering VM ran a vanilla Linux kernel and thus IRS had no effect on it [1]. We define the speedup of an application as its performance under IRS normalized to the performance in vanilla Xen/Linux. We use the weighted (average) speedup of the foreground and background applications to measure the overall system efficiency. The higher the weighted speedup, the higher system efficiency. A weighted speedup of 1 indicates the same performance as vanilla Xen/Linux. The foreground and background workloads were both repeated at least five times to ensure their execution completely overlapped with each other.

Due to space limits, we briefly report the fairness between the foreground and background VMs. IRS did not compromise fairness and the two VMs had a fair share of the pCPUs. The only change IRS made to the scheduling algorithm at the hypervisor is the delay added to each vCPU preemption for the guest OS to process SA notifications. Experimental results show that IRS improved the utilization of the foreground VM but the CPU consumption never exceeded the fair share.

Figures 7 and 9 show the weighted speedup for PARSEC and NPB benchmarks due to different scheduling strategies with a varying degree of interference. The weighted speedup follows the same trends of performance improvement in Figure 5 and 6. For PARSEC benchmarks (as shown in Figure 7), IRS had marginal or no speedup in dedup and `ferret`. For other workloads, IRS improved the system-wide speedup by as much as 40% and the average speedup across all workloads was 18% and 22% when the background workloads were `fluidanimate` and `streamcluster`, respectively.

An examination of the performance of foreground and background workloads revealed that the gain on system weighted speedup was mainly due to the performance improvement in foreground applications. In most cases, the background application had speedup in the range of −5% to 6%, with an exception for the case in which `raytrace` and `fluidanimate` were co-located and `fluidanimate` had 27% improvement. The performance degradation of the background application (as much as 5%) was due to the improved utilization of the foreground application. Thus, the background application had less CPU allocations. These results suggest that IRS did not change the way the background VM was scheduled by the hypervisor and most performance improvement of the foreground VM was due to more efficient load balancing in the guest OS. IRS never degraded the background performance significantly but had unexpected improvement for some background workloads, e.g., `fluidanimate` when running with `raytrace`.

Compared to IRS, `PLE` either had marginal improvement on the weighted speedup or hurt the overall system efficiency. For example, `PLE` degraded the weighted speedup considerably for `vips`, `bodytrack`, and `facesim`. Note that both the foreground and background VM had `PLE` enabled. The frequent trap into the hypervisor and the lack of coordination between the VMs were the culprits of degraded system efficiency. `Relaxed-Co` achieved better performance than `PLE`, but still hurt overall system efficiency when running `bodytrack` and `facesim`.

Similar results can be observed in Figure 9. For example, IRS improved system speedup for most application combinations except the `SP+UA` and `UA+UA` experiments. For spinning workloads, `PLE` and `relaxed-Co` had better worst-case results. For all experiments including those in Figure 7, IRS had no significant impact on system

[1] Without implementing the `VIRQ_SA_UPCALL` interrupt, the background VM ignores the SA notification sent by the hypervisor.

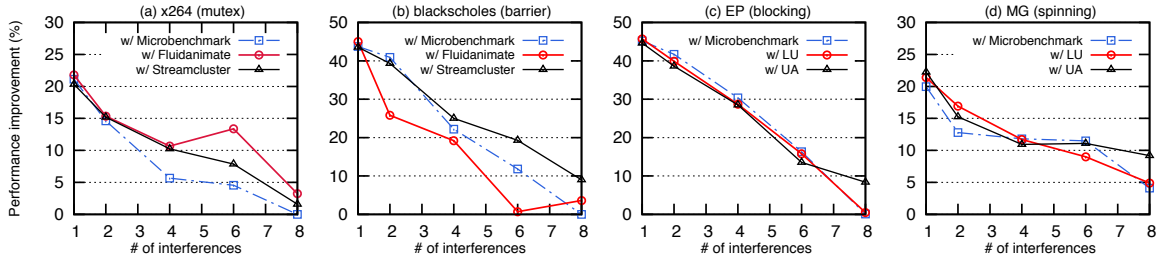

Figure 10. The trend of IRS performance improvement with a varying number of interferences.

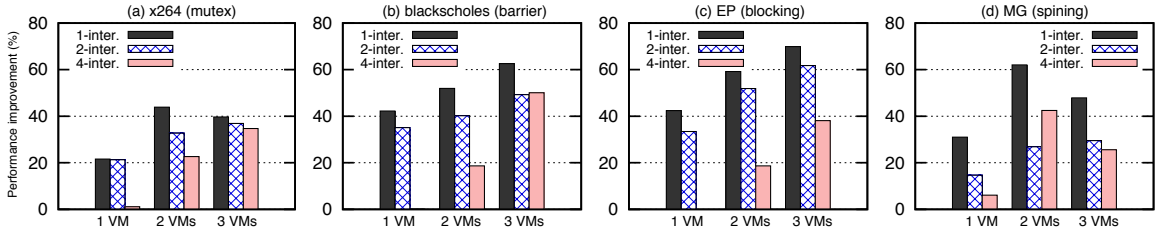

Figure 11. The trend of IRS performance improvement with a varying degree of interferences.

speedup in the *4-inter.* cases. The degradation or improvement of weighted speedup was within the range of −5% to 5%.

5.5 Scalability and Sensitivity Analysis

We have shown the effectiveness of IRS when two VMs shared the pCPUs and the hardware parallelism in SMP VMs was up to 4 vCPUs. In this section, we extend the evaluation to a larger number of vCPUs per VM and consolidating more VMs. We are interested in quantitatively measure the effectiveness of IRS for different types of parallel workloads. All results were the average of five runs.

First, we created two 8-vCPU VMs and configured them to share 8 pCPUs. The foreground VM ran 8-thread parallel workloads while the background VM executed three different types of interferences: one CPU-bound synthetic workload and two real parallel applications. The level of interference varied, starting from one vCPU with interference to all vCPUs (8-vCPU) with interference. We selected four benchmarks from PARSEC and NPB for evaluation. `X264` exclusively uses `pthread mutexes` for point-to-point synchronization and `blackscholes` uses `pthread barriers` for group synchronization between threads. NPB benchmarks employ a data-parallel programming model and use barrier-like synchronizations. `EP` performs less synchronization and uses blocking synchronization. We set `MG` to use spinning synchronization.

Figure 10 shows the trends of performance improvement due to IRS relative to vanilla Xen/Linux. We have the following observations: (1) performance gain diminishes as the number of vCPUs having interference increased. When all vCPUs are experiencing interference, the average gain is marginal at about 4%. (2) Parallel workloads with different types of synchronizations respond differently to IRS. Programs with group synchronization, such as barriers, suffer more from LHP, thereby benefiting more from IRS. The performance gain of point-to-point synchronizations, e.g., mutexes, is less than that of group synchronizations. IRS is more effective for mitigating LHP problems in blocking synchronization than in spinning synchronization. (3) overall, the stated trends apply to all three types of interfering workloads.

Next, we fixed the number of vCPUs in the foreground VM to 4 and varied the number of interfering VMs from 1 to 3. For example, *1-inter.* with three interfering VMs refers to the case that one vCPU of the foreground VM has interference and there are 3 VMs competing for the CPU cycles on the same pCPU. Figure 11 shows that as the degree of interference increases on each interfered foreground vCPU, the performance gain of IRS increases in most cases. Another important observation was that IRS has significant improvement under high degree of interference even all vCPUs of the parallel VM experience interference (i.e., *4-inter. + 3VMs*). We conclude that IRS can be more useful in a highly consolidated scenario with many VMs sharing the same pCPUs.

5.6 Mitigating CPU Stacking

We found that existing SMP schedulers, including the native Linux process scheduler and Xen's hypervisor-level vCPU scheduler, suffer from a severe CPU stacking problem when parallel workloads with frequent blocking are co-located with applications with persistent CPU demands. For example, if a four-thread parallel workload with blocking synchronization, e.g., `streamcluster`, shares 4 CPUs with three persistent CPU hogs, the parallel threads or vCPUs running these threads will be stacked on a single or a small number of CPUs, leaving much of the hardware-level parallelism unexploited by the parallel program. According to our PARSEC experiments with Linux CFS and Xen's credit scheduler, the four parallel threads were stacked on one or two cores and had 5-20x slowdown compared to the case in which threads or vCPUs were pinned to separate cores.

Root causes of CPU stacking Modern SMP schedulers are designed for scalability and proportional fair sharing. Each CPU in an SMP system runs an independent fair-sharing scheduler and relies on thread/vCPU migration for load balancing. The objective of load balancing is to evenly distribute workload onto multiple CPUs. and oftentimes the level of load is measured by the CPU utilization of a thread/vCPU. CPU stacking occurs if threads of a parallel program are placed on the same CPU and multiplexed in a time-sharing manner. As a result, the stacked threads cannot execute simultaneously, leading to the loss of parallelism. CPU stacking occurs due to two reasons. **First**, Thread or vCPU scheduling is oblivious of the dependencies between parallel threads or vCPUs. Thus, placing sibling threads/vCPUs on the same CPU is legitimate as long as it satisfies fair sharing and load balancing. **Second**

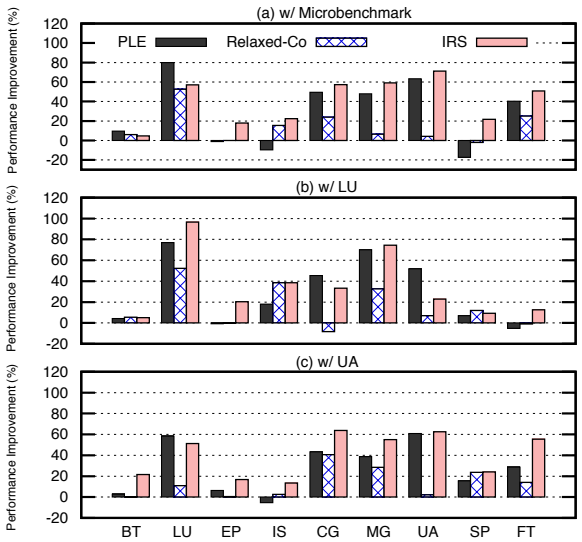

Figure 12. NPB performance in response to CPU stacking.

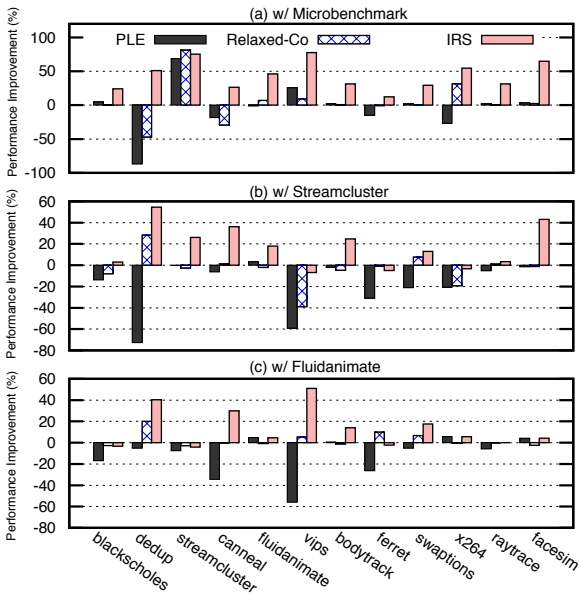

Figure 13. PARSEC performance in response to CPU stacking.

and most importantly, there exists a deficiency in existing SMP schedulers when scheduling blocking workloads. Due to LHP and LWP, parallel threads frequently block and wait to enter the critical section. Since blocked threads do not consume any CPU cycles, they exhibit *deceptive idleness* (DI) to the scheduler. This situation is similar to DI in disk scheduling [16] and causes blocking threads to be consolidated on a small number of CPUs due to their low CPU utilization.

Figure 12 shows the performance of NPB benchmarks under PLE, `relaxed-Co`, and IRS when all the vCPUs of the foreground and background VMs were unpinned. Performance is normalized to that in vanilla Linux/Xen and the interference was *4-inter.* CPU hogs. Since NPB benchmarks never block, the DI situation does not occur. As shown in Figure 12, all strategies were effective in improving NPB performance over the baseline and the degree of improve was significantly higher than that in Figure 6, indicating a mitigation of the CPU stacking problem. Among these scheduling strategies, PLE prevented excessive spinning and `relaxed-Co` balanced the progress of sibling vCPUs, thereby helping spreading them onto separate cores. Compared to PLE and `relaxed-Co`, IRS achieved overall higher performance gain, showing that in-guest load balancing is more resilient to CPU stacking caused by oblivious vCPU scheduling.

Figure 13 shows the performance of PARSEC benchmarks. Note that the stacking of PARSEC application threads was due to deceptive idleness caused by LHP or LWP. The figure shows that neither PLE nor `relaxed-Co` was generally effective in alleviating CPU stacking but exacerbated the performance slowdown. For example, PLE incurred up to 78% performance degradation compared to the baseline in `dedup`. Because the CPU stacking of blocking workloads is due to DI, PLE, which stops spinning vCPU and yields to competing vCPUs, caused more idling of the PARSEC workload. `Relaxed-Co` also caused considerable slowdown in some cases, e.g., `dedup`, `vips`, and `canneal`, because it only switched the leading vCPU and the lagging vCPU and was unable to address the stacking problem. In contrast, IRS proactively pushes threads from preempted vCPUs to idle or less loaded vCPUs, preventing these vCPU from idling. As discussed in § 5.2, With the help of IRS, blocking workloads avoided unnecessary idling and exhibited their factual CPU demand to the SMP scheduler. This helped prevent the DI problem and the resulted CPU stacking.

6 Discussion

Limitation IRS proactively migrates preempted threads to another vCPU based on the estimation of load on the target vCPU. It cannot eliminate all vCPU idle time or achieve perfect load balancing because the load estimate can be inaccurate. The ideal migration should be pull-based and happen when a vCPU becomes idle. This calls for a new mechanism of task migration – migrating a "running" task from a preempted vCPU.

Autonomous guest resource management IRS allows the guest to migrate tasks among its sibling vCPUs and leads to more efficient utilization of its CPU allocation. Although small changes are needed in the guest kernel and hypervisor, we have shown that the changes do not affect the core resource scheduling algorithms at the hypervisor. We believe that hypervisors should provide such interfaces to the guest OS for autonomous and efficient guest resource management.

7 Conclusion

This paper demonstrates that the semantic gap between the guest OS and hypervisor leaves the potential of addressing the LHP and LWP problems in the guest unexploited. We design IRS, a simple approach based the classical concept of scheduler activations to bridging the semantic gap and enhancing in-guest load balancing. Experimental results show that IRS is especially effective for workloads that have a portion of threads with interference in a highly consolidated environment.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported by U.S. National Science Foundation grants CNS-1649502 and The University of Texas STARs program.

References

[1] Kernel patch. https://patchwork.kernel.org/patch/9441007.

[2] Kvm and big vms. https://www.linux-kvm.org/images/5/55/2012-forum-Andrew-Theurer-Big-SMP-VMs.pdf.

[3] Ahn, J., Park, C. H., and Huh, J. In *Proceedings of the 47th International Symposium on Microarchitecture (Micro)* (2014).

[4] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst. 10*, 1 (1992).

[5] Apache HTTP Server Benchmark. https://httpd.apache.org/docs/2.4/programs/ab.html.

[6] Arpaci-Dusseau, A. C. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Trans. Comput. Syst. 19*, 3 (2001).

[7] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. The nas parallel benchmarks-summary and preliminary results. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)* (1991).

[8] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, L., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[9] Cheng, L., Rao, J., and Lau, F. C. vScale: Automatic and efficient processor scaling for smp virtual machines. In *Proceedings of European Conference on Computer Systems (Eurosys)* (2016).

[10] Co-scheduling SMP VMs in VMware ESX Server. http://communities.vmware.com/docs/DOC-4960.

[11] Ding, X., Gibbons, B. P., Kozuch, A. M., and Shan, J. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2014).

[12] Dusseau, A. C., Arpaci, R. H., and Culler, D. E. Effective distributed scheduling of parallel workloads. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (1996).

[13] Friebel, T., and Biemueller, S. How to deal with lock holder preemption. In *Xen Developer Summit* (2008).

[14] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. December 2009.

[15] Intel TBB. http://software.intel.com/en-us/intel-tbb.

[16] Iyer, S., and Druschel, P. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)* (2001).

[17] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2006).

[18] Kashyap, S., Min, C., and Kim, T. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APsys)* (2015).

[19] Kernel Based Virtual Machine. http://www.linux-kvm.org.

[20] Kim, H., Kim, S., Jeong, J., Lee, J., and Maeng, S. Demand-based coordinated scheduling for smp vms. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[21] Kontothanassis, L. I., Wisniewski, R. W., and Scott, M. L. Scheduler-conscious synchronization. *ACM Trans. Comput. Syst. 15*, 1 (1997).

[22] Novakovic, D., Vasic, N., Novakovic, S., Kostic, D., and Bianchini, R. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2013).

[23] Ousterhout, J. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS)* (1982).

[24] Ouyang, J., and Lange, J. R. Preemptable ticket spinlocks: Improving consolidated performance. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)* (2013).

[25] Rao, J., Wang, K., Zhou, X., and Xu, C.-Z. Optimizing virtual machine scheduling in numa multicore systems. In *Proceedings of the International Conference on High Performance Computer Architecture (HPCA)* (2013).

[26] Rao, J., and Zhou, X. Towards fair and efficient smp virtual machine scheduling. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)* (2014).

[27] Sobalvarro, P., Pakin, S., Weihl, W. E., and Chien, A. A. Dynamic coscheduling on workstation clusters. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)* (1998).

[28] Song, X., Jicheng, S., Haibo, C., and Binyu, Z. Scheduling processes, not vcpus. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APsys)* (2013).

[29] Spec Java Server Benchmark. http://www.spec.org/jbb2005.

[30] Sukwong, O., and Kim, S. H. Is co-scheduling too expensive for smp vms? In *Proceedings of European Conference on Computer Systems (Eurosys)* (2011).

[31] Teabe, B., Nitu, V., Tchana, A., and Hagimont, D. In *Proceedings of European Conference on Computer Systems (Eurosys)* (2017).

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

[33] The Princeton Application Repository for Shared-Memory Computers (PARSEC) . http://parsec.cs.princeton.edu/.

[34] Uhlig, V., LeVasseur, J., Skoglund, E., and Dannowski, U. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Conference on Virtual Machine Research and Technology Symposium (VM)* (2004).

[35] VMware. Vmware horizon view architecture planning 6.0. In *VMware Technical White Paper* (2014).

[36] Wells, P. M., Chakraborty, K., and Sohi, G. S. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2006).

[37] Weng, C., Liu, Q., Yu, L., and Li, M. Dynamic adaptive scheduling for virtual machines. In *Proceedings of the 20th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2011).

[38] Weng, C., Zhang, Z., Li, M., and Lu, X. The hybrid framework for virtual machine systems. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)* (2009).

[39] Xen Credit Scheduler. http://wiki.xen.org/wiki/Credit_Scheduler.