Demand-Based Coordinated Scheduling for SMP VMs

Hwanju Kim[†], Sangwook Kim[§], Jinkyu Jeong[†], Joonwon Lee[§], Seungryoul Maeng[†]

[†]Computer Science Department, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea
[§]College of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea
hjukim@calab.kaist.ac.kr, swkim@csl.skku.edu, jinkyu@calab.kaist.ac.kr, joonwon@skku.edu, maeng@kaist.ac.kr

Abstract

As processor architectures have been enhancing their computing capacity by increasing core counts, independent workloads can be consolidated on a single node for the sake of high resource efficiency in data centers. With the prevalence of virtualization technology, each individual workload can be hosted on a virtual machine for strong isolation between co-located workloads. Along with this trend, hosted applications have increasingly been multithreaded to take advantage of improved hardware parallelism. Although the performance of many multithreaded applications highly depends on communication (or synchronization) latency, existing schemes of virtual machine scheduling do not explicitly coordinate virtual CPUs based on their communication behaviors.

This paper presents a demand-based coordinated scheduling scheme for consolidated virtual machines that host multithreaded workloads. To this end, we propose communication-driven scheduling that controls time-sharing in response to inter-processor interrupts (IPIs) between virtual CPUs. On the basis of in-depth analysis on the relationship between IPI communications and coordination demands, we devise IPI-driven coscheduling and delayed preemption schemes, which effectively reduce synchronization latency and unnecessary CPU consumption. In addition, we introduce a load-conscious CPU allocation policy in order to address load imbalance in heterogeneously consolidated environments. The proposed schemes are evaluated with respect to various scenarios of mixed workloads using the PARSEC multithreaded applications. In the evaluation, our scheme improves the overall performance of consolidated workloads, especially communication-intensive applications, by reducing inefficient synchronization latency.

Categories and Subject Descriptors D.4.1 [*OPERATING SYSTEMS*]: Process Management—Scheduling

General Terms Algorithms, Experimentation, Performance

Keywords Virtualization, Synchronization, Coscheduling

1. Introduction

Ever-evolving hardware parallelism and virtualization have been enabler technologies for consolidation of independent workloads on a single powerful node. With the prevalence of virtualization technology, an individual workload can be hosted in an isolated container, a virtual machine (VM), in which a user can control entire software stack. As processor architectures have been enhancing their computing capacity by adding more cores, multiple independent VMs can be consolidated in a single machine in order to improve resource utilization. In this environment, efficient management of shared resources is crucial for the performance of consolidated workloads.

On the software side, many applications have been increasing thread-level parallelism in order to take advantage of improved hardware parallelism. Emerging multithreaded workloads such as RMS (recognition, mining and synthesis) applications [6] are continuously evolving their algorithms to make the best use of available cores. The thread-level parallelization, however, typically requires synchronization with respect to the resources shared by multiple threads. Firstly, all threads in a single process share an address space whereby a page table update requires synchronization of corresponding hardware structures such as translation lookaside buffer (TLB). Secondly, multiple threads can share a memory region via shared variables whose updates must be safely serialized by using synchronization primitives. Such synchronization should be efficiently handled because it is a dominant factor affecting the scalability of multithreaded applications.

In the case where a multithreaded workload is hosted on an SMP VM, the synchronization performance could be degraded by uncoordinated scheduling of virtual CPUs (vCPUs). Since multiple vCPUs can time-share a physical CPU (pCPU), their executions are dictated by a hypervisor scheduler. If the scheduler is oblivious to synchronization demands of vCPUs, it could delay the execution of a critical vCPU upon which other ones depend to make progress, thereby aggravating contention for shared resources. Many researchers have addressed this issue by proposing coordinated scheduling such as relaxed coscheduling [26], balance scheduling [24], spinlock-aware schemes [13, 17, 25, 28], and hybrid coscheduling [27, 30]. The previous schemes, however, did not explicitly coordinate vCPUs in the event of synchronization-related communication between vCPUs that host multithreaded workloads.

This paper presents a demand-based coordinated scheduling scheme for consolidated SMP VMs that host multithreaded workloads. Inspired by traditional demand-based coscheduling for workstation clusters [3, 4, 9, 22, 23], we introduce communication-driven scheduling that dynamically coordinates communicating vCPUs in the event of inter-vCPU synchronization, while managing the other ones in an uncoordinated fashion. Such demand-based coordination can effectively reduce synchronization latency without sacrificing the throughput of non-communicating vCPUs. We take inter-processor interrupt (IPI) into account as inter-vCPU communication signal, which is virtualized and therefore can be unobtrusively recognized by the hypervisor. In order to correlate a certain type of IPI with coordination demand, we investigate synchronization behaviors that involve IPI communication in kernel and user spaces based on the experimental analysis of various multithreaded workloads hosted in SMP VMs.

Our findings on the basis of the analysis are summarized as follows: Firstly, uncoordinated scheduling of vCPUs that synchronize TLB states could incur significant performance degradation of applications that intensively manipulate their shared address spaces. Secondly, contention on user-level synchronization primitives can lead to kernel-level spinlock contention, which could result in lock-holder preemption (LHP) by uncoordinated scheduling. Thirdly, such LHPs derived from user-level contention are closely connected with IPI communication for thread wake-up operations. Finally, IPI communication for thread wake-up can guide the hypervisor to coordination of vCPUs that host coscheduling-friendly workloads.

Based on the findings, we propose IPI-driven coscheduling and delayed preemption as communication-driven scheduling schemes. The IPI-driven coscheduling allows a vCPU that receives an urgent IPI to be preemptively scheduled in order to reduce synchronization latency. The IPI-driven delayed preemption enables a vCPU that initiates a thread wake-up IPI to urgently request additional time slice in order to safely release a spinlock, which is likely held for user-level synchronization, thereby being protected from LHP. For these IPI-driven scheduling schemes, we devise *urgent vCPU first scheduling*, which makes a preemption decision in response to an urgent IPI while cooperating with a proportional-share scheduler for inter-VM fairness. Finally, in conjunction with the communication-driven scheduling, we introduce load-conscious balance scheduling that assigns sibling vCPUs, which belong to the same VM, onto different pCPUs in a best-effort manner while avoiding negative effect of load imbalance.

Our proposed scheme was implemented based on Linux Completely Fair Scheduler (CFS) [18] and the Kernel Virtual Machine (KVM) hypervisor [15]. We evaluated our scheme for various mixes of multithreaded and sequential workloads. For the evaluation, we chose the *PARSEC* benchmark suite [6], which includes 13 emerging multithreaded applications with diverse characteristics. From the results, the demand-based coordinated scheduling improves overall performance compared to the uncoordinated and the balance scheduling [24], especially for synchronization-intensive applications. In addition, the load-conscious balance scheduling improves the performance on imbalanced pCPU loads arising when parallel and sequential workloads are consolidated.

The remainder of this paper is organized as follows: Section 2 describes related work on previous coordinated scheduling and contention management schemes for non-virtualized and virtualized environments, and presents our motivation. Section 3 introduces the design and implementation of our proposed scheme based on experimental analysis. In Section 4, we present our evaluation results and analysis with various scenarios of consolidation. Finally, Section 5 discusses alternative approaches complementary to our scheme and Section 6 concludes our work and presents future direction.

2. Related Work and Motivation

2.1 Uncoordinated vs. Coordinated Scheduling

Uncoordinated scheduling, also called local scheduling, allows each per-CPU scheduler to make its own decision on time-sharing among its assigned threads without any coordination with threads on other CPUs. This type of scheduling maximizes CPU utilization while managing local threads with priority-based or proportional share-based policies. For effective utilization of global CPU resources, a load balancer strives to evenly distribute threads onto available CPUs. In this manner, the uncoordinated scheduling achieves high throughput with low overheads due to the independent scheduling decisions. Since this scheme can effectively handle general workloads with simple implementation, it has been widely employed in most commodity OSes [18, 21] and hypervisors [5, 15].

The uncoordinated scheduling, however, has been known to be ineffective for communicating workloads such as multithreaded and parallel applications [20]. The performance of such workloads highly depends on communication (or synchronization) latency between cooperative threads. Since uncoordinated scheduling is oblivious to dependency between threads on different CPUs, it could increase communication latency by preempting a thread on which cooperative ones depend to make progress. Accordingly, a communication-sensitive application needs the underlying scheduler to coordinate its threads in order to minimize communication latency. A large volume of research on coordinated scheduling has been conducted in traditional multiprocessor and cluster environments [3, 4, 9, 10, 20, 22, 23, 29].

Coscheduling [20] is a representative scheme of coordinated scheduling that allows cooperative threads to be synchronously scheduled and descheduled. Such strictly coordinated scheduling gives an illusion that cooperative threads run on a dedicated machine without communication latency. Despite its effectiveness in minimizing communication latency, the strict requirement of synchronous progress can cause CPU fragmentation, since cooperative threads cannot be scheduled until their required CPUs are all available. Many researchers have claimed that the CPU fragmentation problem becomes serious leading to ineffective CPU utilization in an environment where parallel applications are concurrently hosted with sequential workloads [3, 4, 9, 16, 22, 23, 29].

An alternative solution to this problem is demand-based (dynamic [22, 23] or implicit [3, 4, 9]) coscheduling, which dynamically initiates coscheduling only for communicating threads, whereas non-communicating ones are managed in an uncoordinated fashion. The rationale behind this scheme is that communication is a tangible signal of coordination demand for most parallel workloads. In this regard, it can reduce the communication latency of cooperative threads on demand, while retaining high CPU utilization by relaxing strict coscheduling requirement. Many studies showed that demand-based coscheduling achieves higher overall performance, compared to uncoordinated scheduling and strict coscheduling, in network-of-workstation (NOW) environments where various workloads are generally mixed [2–4, 9].

2.2 Coordination Issues on SMP VMs

Coordinated scheduling is also a compelling issue on SMP VMs as data centers have increasingly been virtualized today. Since commodity OSes have been meant to be running on bare-metal CPUs, they typically make liberal use of spin-based synchronization primitives (e.g., spinlocks) to protect short critical sections. Once the OSes are virtualized, however, a spinlock-protected critical section can be suspended by an underlying hypervisor scheduler. In particular, preempting a vCPU that holds a contended spinlock (i.e., LHP) could increase synchronization latency while forcing contending vCPUs to unnecessarily consume CPU cycles [11, 24, 25]. Accordingly, uncoordinated scheduling can lead to significant scalability bottleneck in consolidating SMP VMs.

In order to address this problem, coordinated scheduling schemes [7, 13, 17, 24–28, 30] have been proposed for SMP VMs to reduce inter-vCPU synchronization latency. As with traditional job scheduling, most proposals have aimed at loosely coordinated scheduling to avoid the inefficiency of strict coscheduling. The VMware ESX server introduced the *relaxed coscheduling* [26], which enables sibling vCPUs to make progress at similar rates by preventing their runtime from being largely skewed. The *balance scheduling* [24] is a probabilistic coscheduling scheme, which increases the likelihood of coscheduling sibling vCPUs by assigning them to different pCPUs. Those two schemes proactively balance pCPU resources on sibling vCPUs without considering any specific coordination demand.

Dynamically coordinated scheduling for SMP VMs has been mainly focused on alleviating excessive busy-waiting on preempted spinlocks of guest OS kernels. Most schemes selectively manipulate scheduling policies for VMs that involve spinlocks based on explicit (user- or OS-informed) [25, 27, 28, 30] or implicit information [7, 25]. Dynamic adaptive scheduling [28] implemented demand-based coscheduling by regarding OS-informed excessive wait time on spinlocks as an indicator of coordination demand, while hybrid schemes [27, 30] selectively coschedule the vCPUs of a concurrent VM specified by a user. Other than coscheduling, some approaches dynamically adjust preemption policy [25] and the length of time slice [7] in order to minimize the synchronization latency on spinlocks. Uhlig *et al.* [25] proposed *delayed preemption* that defers involuntary context switching of a lock-holder vCPU to minimize synchronization latency. In order to identify a lock-holder vCPU, they proposed an OS-informed approach and an inference technique based on the fact that a spinlock is held only in the kernel mode.

Alternative approaches are helping locks [11] and hardware-assisted contention management [1, 12]. Both approaches have the same purpose that avoids busy-waiting on a likely preempted spinlock. The helping lock approach replaces OS spinlocks with spin-then-block based ones, which allow a vCPU that spins on a lock over a threshold period to sleep until the lock is eventually available [11]. The hardware-assisted scheme enables a pCPU to detect excessive spinning by monitoring PAUSE instruction, which is used within a busy-wait loop; Intel and AMD provide Pause Loop Exiting (PLE) [12] and Pause Filter [1], respectively. Once excessive pause-loop is detected based on spin threshold empirically set, a pCPU raises an exception, which causes the transition into the hypervisor (i.e., VMEXIT), so that the hypervisor can handle the contention. On this exception, the hypervisor scheduler allows a corresponding vCPU to yield its pCPU to another one. Although the helping locks and the hardware-assisted scheme effectively reduce the amount of unnecessary busy-waiting, they are reactive approaches triggered once contention occurs after spinning, albeit short.

2.3 Motivation

As with traditional NOW environments [2], virtualized data centers can embrace diverse workloads including parallel, sequential, and interactive applications. With the emergence of Infrastructure-as-a-Service (IaaS) clouds and virtual desktop infrastructure (VDI), such heterogeneity becomes more general. In addition, as modern architectures have been increasing their computing capacity, high consolidation density of SMP VMs can be realized. In order to efficiently support dynamic and diverse thread-level parallelism of consolidated applications, a hypervisor scheduler should carefully coordinate vCPUs based on their workload characteristics.

Considering heterogeneity of consolidated workloads, demand-based coordinated scheduling is an effective approach for SMP VMs to achieve high overall performance of communicating and non-communicating workloads. Although coordination demand can take place for various purposes in user and kernel layers in a VM, prior work on demand-based coscheduling [28] coordinates vCPUs based only on kernel-level spinlock synchronization by means of an OS-assisted technique. Inspired by traditional demand-based coscheduling in NOW environments [3, 23], we investigate that inter-vCPU communication can be a useful signal for coordination demands from the broad viewpoint of kernel-level and user-level synchronization.

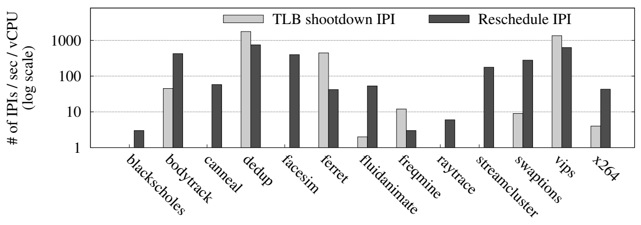


Figure 1. The number of IPIs per second per vCPU of the PARSEC applications: In the case of Linux, TLB shootdown and reschedule IPIs are dominantly used, whereas the other types of IPIs such as function-call IPIs are rarely generated.

3. Demand-Based Coordinated Scheduling

This section presents the design and implementation of the proposed demand-based coordinated scheduling. Fundamentally, our scheme aims at non-intrusive design without collaboration of specific guest-side software (e.g., kernel, user-level applications, and libraries). In addition, our communication-driven coordination manipulates only time-sharing decisions while vCPU-to-pCPU assignment is carried out independently. We present the experimental analysis and implementation for our communication-driven scheduling and the load-conscious balance scheduling for adaptive vCPU-to-pCPU assignment in the following subsections.

3.1 Communication-Driven Scheduling

As mentioned, our hypothesis is that inter-vCPU communication can be used as an indicator of coordination demand. For non-intrusive design, we take IPIs into account as inter-vCPU communication signals, which can be unobtrusively observed by the hypervisor. In order to investigate the implications of IPIs for consolidated multithreaded workloads, we conducted experimental analysis with the PARSEC 2.1 benchmark suite [6], which is comprised of various types of emerging multithreaded applications. Using *native input*, each application ran with eight threads in an 8-vCPU VM, which is consolidated on two quad core processors; the detailed environment is explained in Section 4. Figure 1 shows IPI rates (the number of IPIs per second per vCPU) observed by the KVM hypervisor while each PARSEC application is solely running inside a VM. As shown in the figure, the applications have diverse characteristics in terms of the rates and types of IPIs. The following subsections explain the role of each IPI type and addresses how those low-level signals are related to the demands of kernel- and user-level coordination.

3.1.1 Kernel-Level Coordination Demands

We investigate kernel-level coordination demands and how they are correlated with inter-vCPU communication. In order to identify kernel-level coordination demands, we examine how much the ratio of CPU time spent in the kernel is affected by contention between VMs under uncoordinated scheduling. If the kernel time ratio is largely amplified compared to that without contention, we can figure out that kernel-level contention is not properly resolved by uncoordinated scheduling. We used the Linux CFS scheduler as an uncoordinated scheduler and *streamcluster*, which consumes considerable CPU with heavy communication, as a contending workload. To measure the amplification, we compare the two cases: *solorun* (without contention) and *corun* (with contention).

Figure 2 shows that the ratio of CPU time consumed in kernel and user spaces for each application. As shown in the figure, the ratio of kernel-level CPU time in the corun case is largely amplified by up to 30× compared to the solorun. Interestingly, such amplifi-

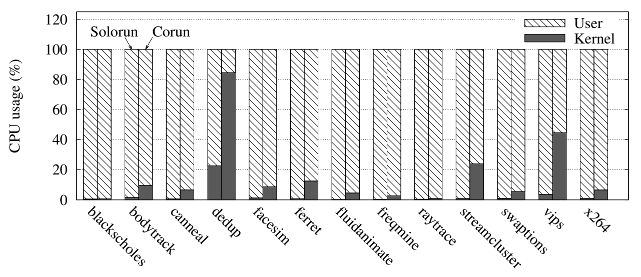


Figure 2. The amplification of kernel time ratio for PARSEC applications in the case of corun with *streamcluster*.

Application	TLB shootdown		Lock spinning	
	% for total CPU usage	% for kernel CPU usage	% for total CPU usage	% for kernel CPU usage
bodytrack	1.96	24.75	**4.56**	**57.58**
canneal	0.02	0.42	**4.02**	**85.17**
dedup	**42.55**	**51.44**	**35.82**	**43.30**
facesim	0.02	0.35	**4.33**	**75.57**
ferret	**8.5**	**75.9**	1.97	17.81
fluidanimate	0.02	0.60	**3.11**	**92.56**
streamcluster	0.02	0.18	**10.35**	**91.11**
swaptions	0.81	13.11	**5.24**	**84.79**
vips	**41.35**	**87.74**	4.32	9.17
x264	0.08	1.03	**6.98**	**89.72**

Table 1. The function-level profiling of CPU usage with respect to the applications whose kernel time ratio is largely amplified in the case of corun with *streamcluster*: The kernel CPU cycles are dominantly consumed for TLB shootdown and lock spinning (bold numbers represent significant amplification for each function).

cation takes place even in the applications that spend most of their time in user space in the case of solorun. In addition, the applications that show largely amplified kernel time are communication-intensive workloads: all applications except *blackscholes*, *freqmine*, and *raytrace* (refer to their IPI rates in Figure 1). This result implies that kernel-level coordination is required for communication-intensive applications even though they are not inherently kernel-intensive. In order to identify the cause of the amplification, we conducted function-level profiling of CPU cycles consumed in the guest kernel by using *perf*. As shown in Table 1, amplified kernel CPU time is mostly spent on two synchronization functions: 1) TLB shootdown and 2) lock spinning.

TLB shootdown is a kernel-level operation for TLB synchronization via inter-CPU (inter-vCPU) communication. In native systems, the kernel ensures that a TLB entry invalidated on one CPU is synchronized with the corresponding entries on the other CPUs for the coherent view of a shared address space. To this end, commodity OSes such as Linux and Windows use an IPI to notify a remote CPU of TLB invalidation. A CPU that initiates TLB shootdown starts busy-waiting until its all recipient CPUs acknowledge IPIs for the sake of TLB consistency. The busy-waiting is efficient in native systems due to the low latency of hardware-based IPIs and high-priority IPI handlers.

Once virtualized, however, a busy-waiting vCPU could consume excessive CPU cycles if one of the recipient vCPUs is not immediately scheduled. This problem becomes serious, as multi-threaded applications typically multicast (or broadcast) TLB shootdown IPIs to the vCPUs associated with a shared address space. Note that the applications that show large amplification of TLB shootdown time involve a considerable traffic of TLB shootdown IPIs; *dedup*, *ferret*, and *vips* pressure their shared address spaces

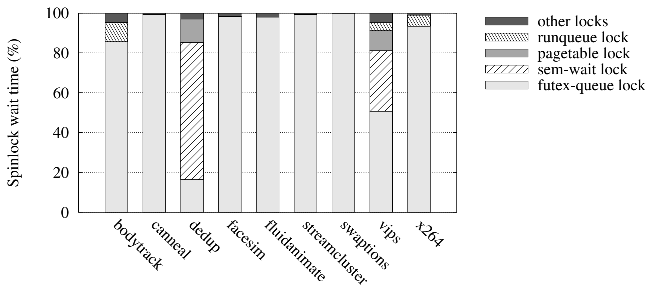


Figure 3. The breakdown of spinlock wait time for the applications where lock spinning time is largely amplified in the case of corun with *streamcluster*.

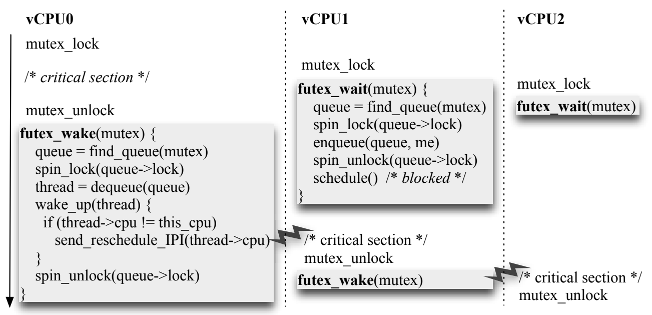
with intensive TLB shootdown operations at the rate of 1761, 443, and 1350 IPIs/sec/vCPU, respectively. Therefore, a TLB shootdown IPI is regarded as a performance-critical signal of inter-vCPU communication that needs to be urgently handled for reducing unnecessary busy-waiting.

Next, the excessive lock spinning, which is another source of kernel time amplification, has been well known as inefficient kernel-level synchronization arising from uncoordinated scheduling. This phenomenon typically stems from the LHP problem where a vCPU that is holding a spinlock is involuntarily descheduled before releasing it. Unlike TLB shootdown, unfortunately, spinlock-based synchronization itself does not entail an explicit signal of inter-vCPU communication. It is important to note, however, that excessive lock spinning happens in the workloads with a large traffic of inter-vCPU communication, especially reschedule IPIs (see Figure 1). A reschedule IPI is used to notify a remote CPU of the availability of a thread newly awakened by a local CPU. Based on this observation, we analyze which type of locks lead to pathological spinning and why this situation likely occurs in reschedule-IPI-intensive applications.

In order to pinpoint where excessive lock spinning occurs, we used *lockstat* [1], which reports holding and waiting time statistics for kernel synchronization primitives. Figure 3 shows the ratio of spinlock wait times for the applications that show considerable amplification of lock spinning time. As shown in the result, a *futex-queue* spinlock mostly results in problematic waiting time; the average wait time of the futex-queue spinlock is 192–13687μs, which is abnormal considering that spinlock-protected critical sections generally last for a few microseconds [25]. The futex is the kernel-level support to provide user-level applications with synchronization primitives such as mutex, conditional variable, and barrier. A futex-queue spinlock is used to protect a wait queue associated with a user-level synchronization object. Accordingly, synchronization-intensive applications lead to contention on the futex-queue lock by requesting aggressive queueing operations.

We take a closer look at the futex-queue lock contention from the perspective of inter-vCPU communication. Figure 4 depicts a typical procedure of how a user-level application interacts with the kernel-level futex support for synchronization. Once a thread exits a critical section, it releases a mutex and notifies, if any, a waiting thread that the lock is available through a futex system call. Then the kernel locates the futex queue associated with the mutex and tries to acquire the queue's spinlock (i.e., futex-queue

[1] The current lockstat in the Linux kernel replaces the default spinlock, *ticket spinlock*, with the old unfair lock by the lock debugging feature. We modified the lock debugging feature to use the ticket spinlock for consistent analysis.



```
vCPU0
mutex_lock
/* critical section */
mutex_unlock
futex_wake(mutex) {
   queue = find_queue(mutex)
   spin_lock(queue->lock)
   thread = dequeue(queue)
   wake_up(thread) {
     if (thread->cpu != this_cpu)
        send_reschedule_IPI(thread->cpu)
   }
   spin_unlock(queue->lock)
}
```

```
vCPU1
mutex_lock
futex_wait(mutex) {
   queue = find_queue(mutex)
   spin_lock(queue->lock)
   enqueue(queue, me)
   spin_unlock(queue->lock)
   schedule()  /* blocked */
}
/* critical section */
mutex_unlock
futex_wake(mutex)
```

```
vCPU2
mutex_lock
futex_wait(mutex)
/* critical section */
mutex_unlock
```

Figure 4. The interaction between user-level synchronization and the kernel support (futex): a gray region represents kernel context and a lightening mark is a reschedule IPI sent from left to right.

lock) to safely dequeue and wake up a thread waiting on the queue. If the thread is decided to be scheduled on a remote CPU by the scheduler, a reschedule IPI is triggered in order to inform the CPU of the newly runnable thread. Note here that the reschedule IPI is sent with the futex-queue lock held. At this point, futex-queue LHP can happen if the waking vCPU is preempted right after sending a reschedule IPI before releasing the lock by either its recipient or another one.

From the analysis, a reschedule IPI can give the hypervisor scheduler a hint that its initiating vCPU likely holds a spinlock. Although the explained procedure is related to a futex-queue lock, our finding is generalized to the relationship between a wait-queue lock and reschedule IPI. Most OSes provide wait-queue APIs for thread-level communication and synchronization [18, 21]. In the Linux kernel, for example, general wake-up functions, prefixed with *__wake_up*, traverse a wait-queue and wake up one or more threads blocked in the queue with its corresponding lock held. In addition, block-based synchronization primitives in the kernel such as *mutex* and *semaphore* maintain their own wait-queues and wake up a waiting thread while holding a spinlock. Note that *dedup* and *vips*, which put significant pressure on their shared address spaces, suffer from excessive spinning on the wait-queue lock (*sem-wait lock*) of *mm*'s semaphore, which is used to protect a shared address space.

Given this hint, the hypervisor can delay the preemption of a vCPU that initiates a reschedule IPI when another vCPU makes a preemption attempt. The amount of delay should be appropriately chosen to allow a vCPU to safely release a likely held spinlock. Since a spinlock-protected critical section is generally short [11, 25], the delay can be empirically determined. However, a critical section that entails IPI transmission, which causes VMEXIT, could be prolonged by hypervisor intervention. To figure out a suitable delay, we conducted sensitivity analysis in Section 4.1.1. Although a sufficiently large value helps avoid LHP, it may prolong the execution of other urgent vCPUs, for example a recipient of a TLB shootdown IPI. A previous delayed preemption scheme that is triggered wherever in kernel space [25] has a larger time window of preemption delay, which may degrade the performance of other urgent vCPUs.

Finally, a wait-queue lock can also be held in a wait procedure other than wake-up operations. As shown in Figure 4, however, a critical section in the wait procedure (*futex_wait*) is extremely short without being interposed by any VMEXIT. Hence, we suppose that wait-queue LHP is unlikely to happen in a wait procedure. With our kernel instrumentation that informs the hypervisor of lock-holding locations, described in Section 4.1.1, we notice that negligible LHPs (near zero) occur in the futex wait function while each PARSEC application is running with *streamcluster*; almost all

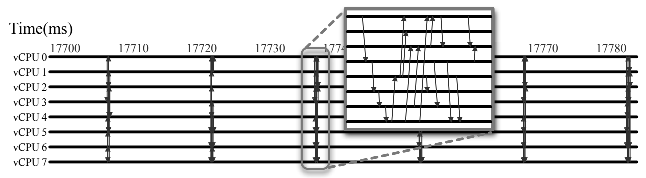


Figure 5. A synchronization behavior of *streamcluster* identified as reschedule IPI transmissions (vertical arrows) for about 80ms.

LHPs happen in *futex_wake* and *futex_requeue*, both of which entail reschedule IPI transmission within a critical section.

3.1.2 User-Level Coordination Demands

Traditional demand-based coscheduling in workstation clusters [3, 4, 9, 22, 23] had dealt with the coordination for user-level applications in which multiple threads heavily communicate with each other. When a thread sends its counterpart a message in order to synchronize a part of parallel computation, the kernel scheduler boosts the priority of the recipient thread so that the communicating threads are coscheduled. The implicit coscheduling [3, 4, 9] takes advantage of an underlying priority-based scheduler, which typically raises the priority of a blocked thread when waking it up. Such demand-based coscheduling was implemented in messaging libraries and firmware of network interface cards for special types of parallel workloads such as bulk synchronous parallel programs.

In virtualized environments, the hypervisor cannot be aware of the actual semantic of user-level communication without the support of threading or messaging libraries. From the viewpoint of the hypervisor, instead, user-level communication could accompany a reschedule IPI if a blocked thread is woken up immediately by a communication message. Since user-level synchronization typically employs block or spin-then-block based primitives, communication between threads can be recognized as reschedule IPIs by the hypervisor. Hence, the hypervisor scheduler can make use of reschedule IPIs to coordinate user-level communication.

Figure 5 depicts a trace of reschedule IPIs obtained by the hypervisor while *streamcluster* is running inside an 8-vCPU VM. The *streamcluster* application makes heavy use of barrier-based synchronization where each thread locally computes its job until a synchronization point, at which all the threads wait for the next stage. This type of application shows no communication during local computation, but involves bulk synchronization at a barrier. As shown in the figure, this behavior is captured via reschedule IPI communication; two to four barrier synchronizations occur at a time within 1ms in a fine-grained manner. If vCPUs are coscheduled in response to reschedule IPIs, all the threads can initiate their computations simultaneously on the coscheduled vCPUs.

The performance impact of coscheduling driven by a reschedule IPI depends on how a hosted application manages contention for its parallel computation. Firstly, a parallel application could involve less efficient synchronization when its hosting vCPUs are not coscheduled. For example, in the event of contention, a spin-then-block synchronization primitive allows a thread to busy-wait for a short period of time until blocked. If the contention is resolved during the spinning, it can avoid a blocking operation, which is expensive due to OS and hypervisor involvements. When such an application runs on coscheduled vCPUs, contention is likely resolved in spin phase without expensive blocking operations. Secondly, depending on algorithms to coordinate parallel computation, additional synchronizations can be induced when threads run on uncoordinated vCPUs. If a scheduler delays the execution of a vCPU that hosts a thread on which other ones depend to make progress, more threads can be blocked with additional contention.

Coscheduling reduces such execution delay so that the number of unnecessary contentions can be effectively curtailed.

The reschedule-IPI-driven coscheduling, however, may not affect the performance of the workloads whose contention management is insensitive to coscheduling. Since a reschedule IPI is generally used for thread scheduling while not confined to the use of synchronization, coscheduling driven by every reschedule IPI cannot improve the performance involving context switch overheads unless a hosted workload is coscheduling-friendly. For efficiency, reschedule-IPI-driven coscheduling can be selectively applied to the VMs that run coscheduling-friendly workloads, while the other scheduling schemes that resolve kernel-level contention are globally enabled. To this end, we enable the hypervisor to expose a knob to selectively enable each feature of the IPI-driven scheduling a per-VM basis. In this work, we use a priori information about coscheduling-friendly characteristics such as spin-then-block synchronization and leave hypervisor-level estimation as future work; the feasibility of hypervisor-level estimation is discussed in Section 4.2.

3.1.3 Urgent vCPU First Scheduling

Our analysis shows that IPIs are the communication signals that enable a hypervisor scheduler to coordinate communicating vCPUs for reducing unnecessary contention. Firstly, when a TLB shootdown IPI is initiated, its recipient vCPU can be urgently scheduled to reduce the amount of busy-waiting of a sender vCPU. Secondly, when a reschedule IPI is initiated, its sender vCPU, which is currently running, can be protected from preemption by another vCPU to reduce the amount of lock spinning due to wait-queue LHP. In addition, its recipient vCPU can be coscheduled to alleviate inefficient or unnecessary user-level contention. In order to handle these coordination demands, we introduce urgent vCPU first (UVF) scheduling, which performs preemptive scheduling and delayed preemption in response to corresponding IPI signals, called *urgent IPIs*.

The UVF scheduling does not replace but complements the proportional-share scheduler in order for its scheduling decision to comply with inter-VM fairness. To this end, per-pCPU FIFO queue, named *urgent queue*, is added over the primary runqueue of the proportional-share scheduler. When a vCPU requests to enter urgent state in response to an IPI, it is inserted into an urgent queue while being included in the primary runqueue; a currently running vCPU can also request to enter urgent state. When a scheduler picks a next vCPU, before inspecting the primary runqueue, it firstly checks whether a vCPU is waiting on the urgent queue and is eligible to run immediately without violating inter-VM fairness by consulting the proportional-share scheduler. If so, the urgent vCPU is preemptively scheduled so that its urgent operation can be promptly handled. Otherwise, a next vCPU is selected from the primary runqueue by the proportional-share scheduler.

In response to an urgent IPI, a corresponding vCPU can request to enter urgent state in two ways: 1) event-based and 2) time-based requests. Firstly, the event-based request is used for a vCPU to be retained in urgent state until pending urgent events are all acknowledged. A TLB shootdown IPI uses the event-based request to keep its recipient vCPU in urgent state until acknowledged in the event of end-of-interrupt (EOI) for its corresponding vector; an EOI signal is triggered right after a requested TLB entry is invalidated. Secondly, the time-based request allows an IPI to specify a time during which a corresponding vCPU can run in urgent state. A reschedule IPI uses the time-based request to preserve its sender vCPU in urgent state until its requested time is taken to release a wait-queue lock. In addition, a recipient vCPU of a reschedule IPI can also be urgently scheduled during a requested time for user-level coordination by means of the time-based request.

The UVF scheduling employs its own time slice, named *urgent tslice*, for urgent vCPUs to expedite pending requests. Since multiple VMs can involve urgent IPIs concurrently, the urgent tslice should be a short time period to improve overall responsiveness. The vCPUs waiting on an urgent queue are served in a round-robin manner with the urgent tslice, during which an urgently running vCPU is protected by preemption. If an urgent vCPU cannot handle all requests during the time slice, it is requeued at the tail of an urgent queue retaining urgent state. Although extremely short urgent tslice improves overall turnaround time, it needs to be long enough for an urgent vCPU to handle at least one urgent request for useful work in the time slice. For example, the urgent tslice can be preferably set greater than or equal to the preemption delay requested by a reschedule IPI sender.

The UVF scheduling introduces an additional knob, called *urgent allowance*, for a vCPU to borrow an urgent tslice from its future CPU allocation by trading short-term fairness for overall efficiency. This mechanism is similar to the Borrowed-Virtual-Time scheduling [8] and Partial Boosting [14] in that a latency-sensitive vCPU (i.e., urgent vCPU) is given dispatch preference while not disrupting long-term CPU fairness. For example in the CFS scheduler, vCPU execution time is monitored in *virtual runtime*, which proceeds at a rate inversely proportional to a given share. If the virtual runtime of a vCPU is larger than that of the currently running one, it cannot be preemptively scheduled while waiting until its virtual runtime becomes minimum in the runqueue [18]. This strict short-term fairness inhibits efficiency by increasing unnecessary busy-waiting due to the prolonged scheduling latency of urgent vCPUs. In order to address this issue, an urgent vCPU is allowed to preemptively run by borrowing the urgent tslice from its future CPU time, only if its time is greater than that of the currently running one within a urgent allowance. In the case of CFS, the urgent allowance is represented in the form of virtual time; in the *Xen Credit scheduler*, it can be specified as *credit*. With a short urgent tslice under one millisecond, the urgent allowance can improve efficiency with a little time lag.

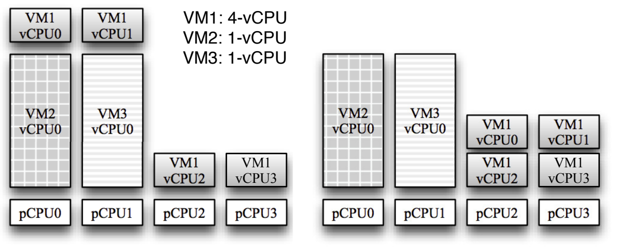
3.2 Load-Conscious Balance Scheduling

In our scheme, vCPU-to-pCPU assignment is separately carried out being decoupled from the communication-driven scheduling. This subsection describes how load imbalance typically happens in heterogeneous workloads and presents our load-conscious balance scheduling scheme for adaptive vCPU-to-pCPU assignment.

3.2.1 Load Imbalance in Consolidated Workloads

In uncoordinated or loosely coordinated scheduling, a load balancer distributes the loads imposed by runnable vCPUs onto available pCPUs as evenly as possible. The load balancing minimizes idle CPU fragmentation so that high CPU throughput and responsiveness can be achieved. Since frequent vCPU migration leads to inefficient use of warm hardware state such as CPU caches, most schemes adopt lazy algorithms to balance global loads. Firstly, once a pCPU goes idle, it tries to steal a waiting vCPU from a busier pCPU like a work-stealing mechanism. In the case where all pCPUs are busy, secondly, waiting vCPUs are periodically migrated onto less loaded pCPUs. Since the load balancing operations are loosely triggered, pCPU loads could be temporarily imbalanced between the invocations of the load balancer.

In addition to the transient load imbalance, pCPU loads can be biased in a situation where VMs are consolidated with different or dynamic loads on their vCPUs. Independent VM instances can have different numbers of vCPUs while holding fair shares. Given the equal shares, a vCPU of a single-core VM has twice shares (i.e., load) than that of a dual-core VM. Although all VMs have the same number of vCPUs with fair shares, each vCPU



(a) Balance scheduling (b) LC balance scheduling

Figure 6. The vCPU-to-pCPU assignment of the balance and LC balance scheduling on four pCPUs whose loads are imbalanced by one 4-vCPU VM and two 1-vCPU VMs: All VMs are given fair shares and the size of each vCPU represents the amount of shares. The vCPU of 1-vCPU VM has shares four times more than that of 4-vCPU VM.

could be given different shares depending on the number of active vCPUs. Proportional-share schedulers for SMP VMs (e.g., CFS group scheduler and Xen Credit scheduler) monitor recent idleness of existing vCPUs in order to distribute a VM's shares only to its active vCPUs. For example, if shares, `S`, are given to a 4-vCPU VM that runs only a sequential workload on a vCPU while idling the others, the active vCPU is allotted whole amount of the shares, `S`. Once the workload enters a parallel phase busying all vCPUs, `S/4` is evenly distributed to each vCPU. This adaptive allocation reduces ineffective shares wasted by inactive vCPUs, but can increase the extent of load imbalance by different amounts of vCPU shares.

3.2.2 Load-Conscious Balance Scheduling

A simple and effective algorithm of vCPU-to-pCPU assignment is the balance scheduling [24], which assigns sibling vCPUs onto different pCPUs in order to prevent them from time-sharing (contending for) a pCPU; the time-sharing of sibling vCPUs is so-called *vCPU stacking* in [24]. Avoiding vCPU stacking can increase the likelihood of coscheduling sibling vCPUs, compared to uncoordinated scheduling, so that negative effect on synchronization latency can be relieved. To this end, this scheme restricts the pCPU affinity of an awakened vCPU to the set of pCPUs to which no sibling vCPU is assigned, while letting the underlying scheduler choose an appropriate pCPU (e.g., least-loaded pCPU) in the specified affinity.

The balance scheduling, however, could degrade synchronization latency if pCPU loads are imbalanced at the moment of assignment. In this case, since the algorithm does not allow vCPU stacking without considering global loads, a vCPU can be assigned to an overloaded pCPU when underloaded ones are all occupied by its sibling vCPUs. Figure 6(a) shows the situation in which the four vCPUs of VM1 are assigned to different pCPUs while loads are imbalanced by the vCPUs with larger shares of two 1-vCPU VMs; all VMs are given fair shares and the size of each vCPU represents the amount of shares. As shown in the figure, although pCPU2 and pCPU3 are sufficiently underloaded, vCPU0 and vCPU1 of VM1 are confined to the overloaded pCPUs (pCPU0 and pCPU1) in order to avoid vCPU stacking. As a result, synchronization latency can be prolonged due to high scheduling latency on the overloaded pCPUs. Moreover, the balance scheduling of VM1 can hurt the performance of 1-vCPU VMs (VM2 and VM3) by contending for the overloaded pCPUs; in this example, each 1-vCPU VM is entitled to monopolize a pCPU without interference for inter-VM fairness.

We propose load-conscious (LC) balance scheduling, which is an adaptive assignment policy based on the balance scheduling and load balancing. In order to avoid ineffective assignment on imbalanced pCPU loads, this scheme selectively allows vCPU stacking in the case where the balance scheduling can aggravate load imbalance. When a vCPU is woken up, the algorithm obtains a set of candidate pCPUs to which no sibling vCPUs are assigned. Then, it decides whether each candidate pCPU is overloaded by checking if the load of each pCPU is higher than the average load of all pCPUs. If at least one underloaded pCPU exists in the set, the set is determined as the pCPU affinity of the vCPU as the balance scheduling does. Otherwise (i.e., all the candidates are overloaded), the affinity is set to all pCPUs without any restriction so that the vCPU can be assigned to an underloaded pCPU. Finally, vCPU stacking is also allowed when the load balancer tries to migrate the vCPU to an underloaded pCPU to which its sibling vCPU has been assigned. Figure 6(b) shows the assignment by the LC balance scheduling where vCPU stacking is allowed on skewed loads.

Although the LC balance scheduling allows vCPU stacking only on underloaded pCPUs, synchronization latency can be adversely affected by contention between sibling vCPUs that communicate with each other. For example, when a vCPU receives a reschedule IPI and is woken up on the pCPU where its sender vCPU is running, wake-queue LHP can happen if the woken vCPU immediately preempts the sender. Therefore, in conjunction with the LC balance scheduling, our communication-driven scheduling is essential to alleviate such negative effect of vCPU stacking by coordinating communicating vCPUs as described in Section 3.1.3; this impact is evaluated in Section 4.1.2.

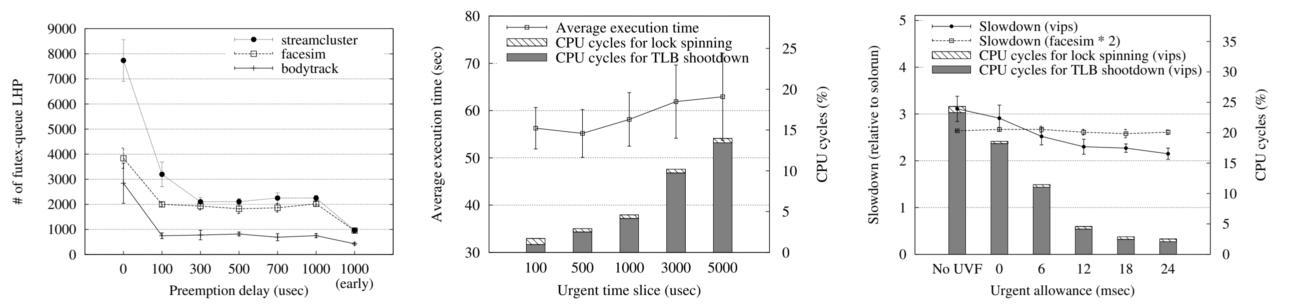
4. Evaluation

We implemented our proposed scheme based on the Linux CFS scheduler and the KVM hypervisor [15] in the Linux kernel 3.2.0. For proportional sharing for VMs, we used the CFS group scheduling, which proportionally distributes given shares to each VM. All per-VM threads including vCPUs are grouped together via *cgroup* interface [19]. For fair sharing of pCPUs, equal shares are given to each VM (group); the shares were set to default shares (1024) multiplied by the number of pCPUs. The prototype was installed on Dell PowerEdge R610, equipped with two quad-core Intel Xeon X5550 2.67GHz processors and 24GB RAM; eight physical cores are available with hyperthreading disabled. We used Ubuntu 11.04 Linux with the kernel version 3.2.0 as a guest OS.

In order to show the impact of each proposed scheme, we conducted experiments for the following UVF scheduling policies:

- `Resched-DP`: In response to a reschedule IPI, its initiating vCPU enters urgent state for delayed preemption using a time-based request.
- `TLB-Co`: In response to a TLB shootdown IPI, its recipient vCPU enters urgent state for coscheduling using an event-based request.
- `Resched-Co`: In response to a reschedule IPI, its recipient vCPU enters urgent state for coscheduling using a time-based request.

As mentioned, the first two schemes coordinate kernel-level contention, while the last one is for the coordination of user-level contention. For vCPU-to-pCPU assignment, the LC balance scheduling (`LC Balance`) was used with the UVF scheduling schemes. We used the default CFS scheduler (denoted as `Baseline`) as an uncoordinated scheduler. In addition, the balance scheduling (denoted as `Balance`), which implements probabilistic coscheduling, was also compared with our schemes. We ran each mixed workload repeatedly (at least three times) in order to fully overlap their executions. We disabled the dynamic tick feature in the host kernel (i.e., hyper-



(a) Preemption delay (b) Urgent time slice (c) Urgent allowance

Figure 7. Parameter sensitivity of the UVF scheduling

visor), since this energy-related feature affects the performance of several applications in the case of solorun.

4.1 Coordination for Kernel-Level Contention

4.1.1 Parameter Sensitivity

As described in Section 3.1.3, the UVF scheduling uses three parameters: 1) preemption delay, 2) urgent tslice, and 3) urgent allowance. Firstly, preemption delay specifies a time during which a vCPU that initiates a reschedule IPI is allowed to release a wait-queue lock before being preempted. Secondly, urgent tslice decides the turnaround time of multiple urgent vCPUs waiting on a pCPU. Finally, urgent allowance determines how much short-term fairness is traded for overall efficiency by allowing an urgent vCPU to borrow its future time slice. This subsection presents sensitivity analysis on these parameters; for every analysis, an 8-vCPU VM with 4GB memory was used to host a parallel application and the LC balance scheduling was used for vCPU-to-pCPU assignment.

Firstly, with respect to the preemption delay, we chose three communication-intensive applications (*streamcluster*, *facesim*, and *bodytrack*) in which futex-queue locks are dominantly contended with considerable reschedule IPIs. The *dedup* application was used as a corunning workload to generate intensive preemptions, since it induces significant thread wake-ups by fine-grained communication. `Resched-DP` was applied only to a main workload while a corunning one is not affected by delay parameters for consistent interference; all UVF features but `Resched-DP` are disabled. We measured the number of LHPs by enabling the hypervisor to identify which spinlock is held at the time when a vCPU is preempted. To this end, we instrumented the Linux spinlock functions to record an instruction pointer where a spinlock is acquired in a per-vCPU variable shared with the hypervisor; the variable maintains multiple instruction pointers for nested lock acquisitions.

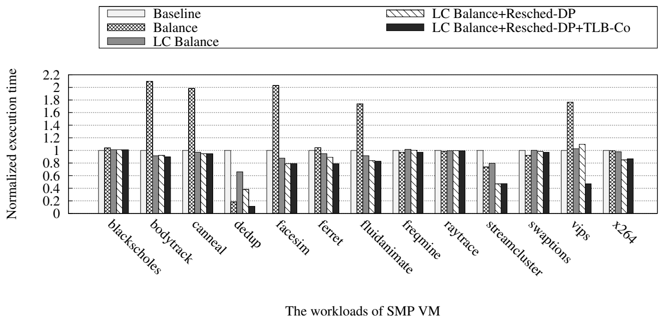
Figure 7(a) shows the number of futex-queue LHPs averaged on five runs as the amount of preemption delay increases. As shown in the figure, the number of LHPs is significantly reduced by `Resched-DP` (up to 75%). With the delay parameters larger than 300μs, the numbers of LHPs become stable without further noticeable reduction. One thing to note is that there still remain LHPs even though the delay is increased up to 1ms. In order to find the source of the remaining LHPs, we also obtained an instruction pointer at the time of preemption. From the analysis, many remaining LHPs happened during the preparation for reschedule IPI transmission. Such preparation involves multiple APIC (Advanced Programmable Interrupt Controller) accesses, which cause transitions to the hypervisor via VMEXIT. By multiple hypervisor interventions during the preparation, a critical section that includes IPI transmission is prolonged and thus likely to be suspended by preemption before firing an IPI. To verify this, we applied *early delayed preemption*, which is triggered on the first access to the APIC register for IPI transmission (i.e., ICR read in x86 APIC). As shown in the figure, the early delayed preemption further reduces the remaining LHPs.

Secondly, we analyzed the impact of urgent tslice by using a TLB-shootdown-intensive application, *vips*, since TLB shootdown latency is sensitive to urgent tslice when multiple recipient vCPUs concurrently request urgent scheduling on a pCPU. In order for multiple vCPUs to contend in an urgent queue, we ran three *vips* VMs with `TLB-Co` and `Resched-DP` (with 500μs preemption delay) enabled. Figure 7(b) shows the execution time and CPU cycles averaged on ten runs as urgent tslice is increased. As expected, a larger time slice increases the scheduling latency of TLB-IPI-recipient vCPUs leading to a larger amount of CPU cycles consumed for TLB shootdown. Although a short time slice, 100μs, results in the lowest consumption of CPU cycles for TLB shootdown, the performance is less than that for 500μs due to the overheads caused by frequent context switches.

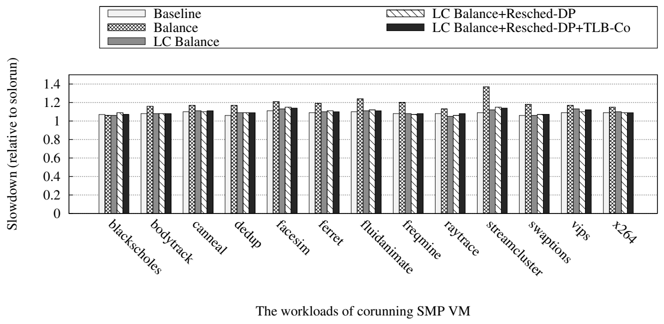
Finally, we investigated the effectiveness of urgent allowance considering the dependency on the proportional-share scheduler. In the CFS scheduler, urgent allowance is represented as virtual time. It allows a vCPU whose virtual runtime is greater than that of the currently running one within an urgent allowance to preemptively run during an urgent tslice, which is borrowed from its future CPU time. For evaluation, we chose *vips* and *facesim* as main and corunning workloads, respectively. The reason we select *facesim* as a corunner is that it can impede the urgent scheduling of the *vips* VM by repeated sleep and wake-up of its worker threads. We coran two *facesim* VMs with `TLB-Co` and `Resched-DP` enabled; preemption delay and urgent tslice were set to 500μs.

Figure 7(c) shows the slowdown relative to solorun of both workloads and the CPU cycles of the *vips* VM. As shown in the result, larger urgent allowance improves the performance of *vips* while not sacrificing that of *facesim*. The CFS scheduler places an awakened vCPU before the minimum virtual runtime by a half of *sched_latency*, which is 24ms in our default configuration. Accordingly, the awakened vCPUs of the *facesim* VMs are frequently placed 12ms before those of the *vips* VM. Considering this policy, an urgent allowance value larger than 12ms helps an urgent vCPU effectively borrow future time slice to preemptively handle urgent IPIs.

On the basis of the analysis, we chose 500μs as preemption delay and urgent tslice, and 18ms as urgent allowance in the remaining evaluations. In addition, we did not apply the early delayed preemption to `Resched-DP`. Since the type of an IPI to be sent cannot be identified at the first ICR read access, early delayed preemption is performed at every type of IPI transmission. In order to evaluate the impact of delayed preemption specifically for reschedule IPI transmission, we excluded this optimization from `Resched-DP`.



(a) Normalized execution time of SMP VM's workloads



(b) Slowdown relative to solorun of *x264* in a UP VM

Figure 8. The performance for the mix of an SMP (8-vCPU) VM for parallel workloads and four UP VMs for sequential workloads.

4.1.2 Mix of Parallel and Sequential Workloads

We evaluated our proposed scheme in the environment where sequential and parallel workloads are consolidated. This environment represents typical consolidation scenarios in data centers that embrace heterogeneous workloads (e.g., IaaS clouds and VDI). For this type of mixed workloads, we ran an 8-vCPU VM for a parallel workload from the PARSEC suite, while corunning four 1-vCPU VMs (called UP VMs), each of which hosted a sequential workload, *x264* single-threaded version. As mentioned in Section 3.2.1, the mix of sequential and parallel workloads intrinsically incurs load imbalance. Although a UP VM was used in this evaluation, an SMP VM that runs a sequential workload can cause similar load imbalance by activeness-based share distribution.

Figure 8(a) shows the normalized execution time of each parallel workload running in an 8-vCPU VM. The first thing to note is that the balance scheduling degrades the performance compared to the baseline in some cases: *bodytrack*, *canneal*, *facesim*, *ferret*, *fluidanimate*, and *vips*. As mentioned in Section 3.2.2, this result is caused by ineffective vCPU-to-pCPU assignment of the balance scheduling in the case where pCPU loads are imbalanced; a vCPU of the 8-vCPU VM can be assigned to an overloaded pCPU where a UP VM's vCPU with larger shares (up to 8×) is running. Nevertheless, the baseline case does not always outperform the balance scheduling due to vCPU stacking by uncoordinated scheduling.

The LC balance scheduling resolves the problem of ineffective vCPU-to-pCPU assignment of the balance scheduling by preventing a vCPU from being assigned to an overloaded pCPU. Since the LC balance scheduling, however, allows vCPU stacking on imbalanced pCPU loads, unnecessary busy-waiting can happen due to the contention between sibling vCPUs. In the case of *dedup* and *streamcluster*, the LC balance scheduling shows lower performance than the balance scheduling, since the negative effect of vCPU stacking outweighs the benefit from avoiding load imbalance.

As shown in the figure, the UVF scheduling improves the performance (by -1–89% compared to the baseline and 0–83% compared to the LC balance scheduling) by effectively coordinating sibling vCPUs that contend with each other. In more detail, `Resched-DP` improves the performance of the applications that highly contend for wait-queue locks (*dedup*, *facesim*, *ferret*, *fluidanimate*, *streamcluster*, and *x264*), while `TLB-Co` contributes the performance improvement of TLB-shootdown-intensive applications (*dedup*, *ferret*, and *vips*). As a result, the UVF scheduling along with the LC balance scheduling achieves the best performance among all the scheduling schemes.

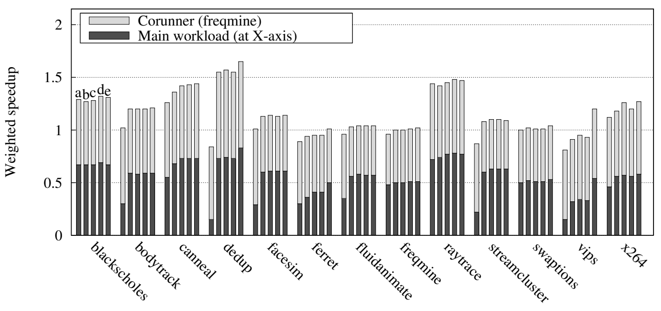
We also evaluated the slowdown relative to solorun of the *x264* single-threaded applications in four UP VMs. In this experiment where eight pCPUs are fairly shared by five VMs, the workload of a UP VM is unlikely to suffer slowdown, since each UP VM is entitled to monopolize a single pCPU without time-sharing; in practice, despite a dedicated pCPU, slowdown could exist by contentions for other types of resources such as shared caches and memory bandwidth. Figure 8(b) shows the average slowdown of *x264* in the UP VMs depending on the mixed parallel workloads (on the X-axis) and scheduling schemes. As shown in the figure, the balance scheduling results in noticeable slowdown of UP VMs by up to 1.37×, while the baseline and our schemes show a little slowdown close to one. As mentioned in Section 3.2.2, the balance scheduling can degrade the performance of UP VMs by assigning some vCPUs of the SMP VM to the pCPUs where those of UP VMs are running. Such nontrivial slowdown implies that the load-oblivious assignment of the balance scheduling can compromise inter-VM fairness.

4.1.3 Mix of Parallel Workloads

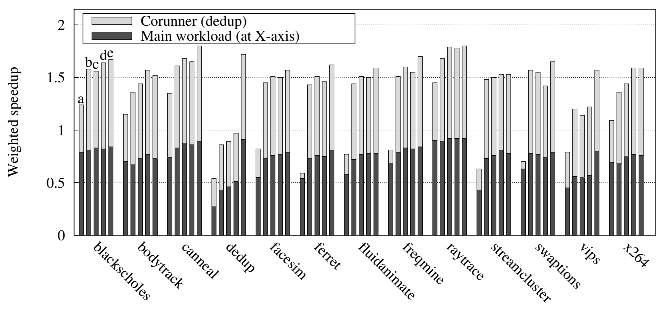
We also evaluated the mix of parallel workloads, each of which ran in an 8-vCPU VM. In this type of mix, every pCPU can be time-shared by vCPUs of different VMs. We chose two corunning applications with different characteristics: *freqmine* and *dedup*. The *freqmine* application is CPU-saturated with a little communication between threads. Accordingly, this workload consumes high pCPU bandwidth, but infrequently preempts the currently running vCPUs. The *dedup*, on the other hand, has varying CPU consumption with a significant amount of communication between threads, thereby inducing intensive preemptions.

Figure 9 shows the weighted speedup of corunning workloads. The weighted speedup is the sum of the speedups relative to solorun of each workload (i.e., $\Sigma(Time_{solorun}/Time_{corun})$) . As shown in Figure 9(a), communication-intensive applications suffer from unfair performance degradation while running with the CPU-bound *freqmine* application under uncoordinated scheduling (i.e., baseline). The balance scheduling mostly resolves such degradation, since the communication-intensive workloads are likely to be coscheduled by preempting the CPU-bound *freqmine* on balanced loads, where the LC balance scheduling takes similar effect. The UVF scheduling contributes to more performance improvement especially for TLB-shootdown-intensive applications (*dedup*, *ferret*, and *vips*). `Resched-DP` shows little effect because of infrequent preemptions of *freqmine*.

In Figure 9(b), the performance of *dedup* (corunner), which is sensitive to TLB-shootdown latency with heavy communication traffic, drops significantly in the case of the baseline. Similarly, the balance scheduling alleviates such large performance degradation by spreading sibling vCPUs. LC balance scheduling achieves higher performance of both workloads than the balance scheduling, since loads are frequently imbalanced by the varying loads of *dedup*. Furthermore, the UVF scheduling shows the best overall

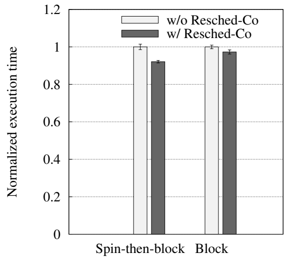


(a) Corun with *freqmine*

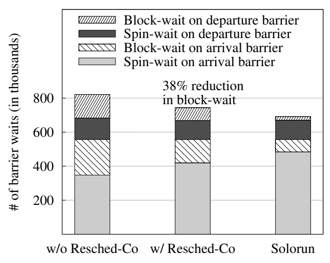


(b) Corun with *dedup*

Figure 9. The weighted speedup of two parallel workloads (a: `Baseline`, b: `Balance`, c: `LC Balance`, d: `LC Balance+Resched-DP`, e: `LC Balance+Resched-DP+TLB-Co`).



(a) Performance



(b) The number of barrier waits

Figure 10. The performance impact of Resched-Co and the breakdown of spin-then-block barrier waits of *streamcluster* that coruns with *bodytrack*.

performance along with the LC balance scheduling. Some drops in weighted speedup in the case of `Resched-DP` alone stems from the performance degradation of *dedup*, since the delayed preemption of a main workload defers the TLB-shootdown of *dedup*. This problem is resolved in the case where `TLB-Co` is applied.

4.2 Coordination for User-Level Contention

As mentioned in Section 3.1.2, inter-thread communication in a synchronization-intensive application can be recognized as bulk traffic of reschedule IPIs. For this type of synchronization-intensive applications, user-level contention can be reduced by coscheduling vCPUs in response to their reschedule IPIs. For the evaluation, we chose *streamcluster*, which intensively uses its in-house barrier, from the PARSEC suite. The *streamcluster* application is coscheduling-friendly, since it adopts a spin-then-block synchronization in its in-house barrier designed for the fine-grained synchronization. Using this primitive, when a thread reaches a barrier, it firstly spins for a short period (approximately 0.1ms) and is then blocked if all threads have not arrived at the barrier yet. If all the threads are coscheduled and their computation loads are not skewed, more barrier waits avoid blocking, which induces context switch and VMEXIT, at the smaller expense of spinning. We coran *bodytrack*, by which *streamcluster* suffers larger interference than any other mixes. For `Resched-Co`, when a reschedule IPI is initiated, its recipient vCPU enters urgent state for 500μs, which is the same as the urgent tslice.

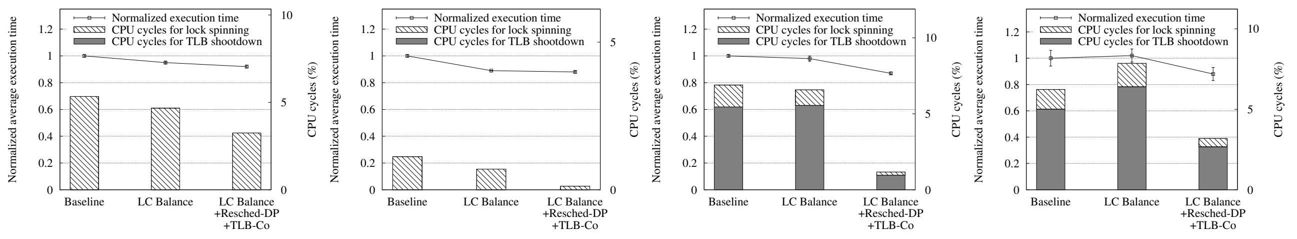
Figure 10(a) shows the performance improvement of *streamcluster* to which `Resched-Co` is applied; `Resched-DP` and `TLB-Co` were enabled in both cases. As shown in the figure, `Resched-Co` improves the performance by 8% in the case of spin-then-block barrier while the performance of *bodytrack* is not affected (the result for *bodytrack* is omitted from the graph for brevity). As expected, this improvement is higher than that of block-based barrier (2.7%). This result demonstrates that spin-then-block synchronization is more coscheduling-friendly. In order to identify the effect of coscheduling, we obtained the information of barrier waits with regard to how many waits are resolved in spin phase and whether additional contentions occur. Figure 10(b) shows the breakdown of barrier waits in *streamcluster*.

The in-house barrier internally uses a *pthread* conditional variable in two ways. Firstly, an arrival barrier is used for all threads to wait for synchronously proceeding to a next stage. Secondly, a departure barrier is used to make sure all threads to entirely depart from a previous stage before starting a next arrival phase. The number of arrival barrier waits is deterministic in the program, whereas departure barrier waits can be increased if the execution of an awakened thread on the arrival barrier is delayed until another thread arrives at the barrier after completing the current stage. Accordingly, the number of departure barrier waits could be increased by the skewed execution of threads. As shown in the figure, block-waits are reduced by 38% due to the parallel executions on coscheduled vCPUs by `Resched-Co`. In addition, the number of departure barrier waits is also reduced by 29%, since the progress of threads is less likely biased by coscheduling. As a result, the wait behavior of coscheduling is close to the solorun, compared to that without coscheduling.

The reduction in blocking operations leads to less reschedule IPIs required for synchronization because a reschedule IPI is triggered to wake up a blocked thread. In this experiment, `Resched-Co` decreases the number of reschedule IPIs for *streamcluster* by 21%. Such IPI reduction alleviates the cost of hypervisor interventions for IPI communication (e.g., VMEXIT and APIC virtualization). Furthermore, the association between the reduction in reschedule IPIs and the benefit from coscheduling can allow the hypervisor to infer coscheduling-friendly workloads. If the bulk traffic of reschedule IPIs is reduced once vCPUs are coscheduled by `Resched-Co`, the hypervisor can infer that the coscheduling has a positive effect on the applied workload.

4.3 Effectiveness with Hardware-Assisted Contention Management

We evaluated our scheme on Intel PLE-enabled processors [12], which support hardware-assisted contention management described in Section 2.2. For this experiment, our prototype was installed on Dell PowerEdge R710 equipped with two quad-core Intel Xeon E5607 2.27GHz processors. For the processors to detect busy-



(a) streamcluster (b) facesim (c) ferret (d) vips

Figure 11. Normalized execution time and CPU cycles of parallel workloads in an SMP VM mixed with four UP VMs running *x264* on PLE-enabled processors.

Application	streamcluster	facesim	ferret	vips
Reduction in pause-loop exits (%)	44.5	97.7	74.0	37.9

Table 2. Reduction in pause-loop exits by the UVF scheduling compared to the baseline.

waiting, PLE_Gap and PLE_Window [12] were set to 128 and 4096, respectively; these values are used in KVM and Xen by default. In PLE-enabled processors, the UVF scheduling cancels urgent state once a pause-loop exit occurs, since this exit indicates that the urgent vCPU starts unnecessary busy-waiting. We evaluated the mix of a parallel and four sequential workloads as in Section 4.1.2 with two futex-intensive applications (*streamcluster* and *facesim*) and two TLB-shootdown-intensive applications (*ferret* and *vips*) as parallel workloads.

Figure 11 shows the average execution time normalized to the baseline and CPU cycles for lock spinning and TLB shootdown. As shown in the result, the performance is improved by 8–13% in our scheme (LC balance scheduling with `Resched-DP` and `TLB-Co`). It is because our scheme proactively alleviates the number of busy-waits, whereas the hardware-assisted scheme reactively resolves contention by yielding a pCPU once busy-waiting is detected. Note that our proactive coordination significantly reduces the CPU cycles for lock spinning and TLB shootdown. Alleviating the number of busy-waits, furthermore, leads to the reduction in pause-loop exits. As shown in Table 2, the number of pause-loop exits is reduced by 37.9–97.7%, which means much lower overheads of hypervisor intervention.

Nevertheless, the hardware support to notify the hypervisor of prolonged busy-waiting is an essential fallback, since scheduling-oriented schemes cannot completely eliminate all the excessive busy-waiting in guest OSes. Although `Resched-DP` is effective for communication-intensive applications, it cannot resolve the LHPs that happen in the kernel without reschedule IPI communication. Other than lock spinning, furthermore, busy-waiting has been prevalently used in the kernel for performance optimization based on the assumption that CPUs are physically dedicated. In this regard, the hardware-assisted scheme is complementary to our coordinated scheduling for efficient contention management.

In addition, the current yielding scheme on a pause-loop exit can be refined with the aid of the UVF scheduling. Currently, at every pause-loop exit, a vCPU tries to yield the pCPU to another regardless of the type of busy-waiting. The UVF scheduling, however, can allow a vCPU that sends a TLB shootdown IPI to busy-wait for more time favorably without yielding the pCPU if its recipient vCPUs can be urgently scheduled. Since the hypervisor can identify if the current busy-waiting is for TLB synchronization by checking the transmission of TLB shootdown IPIs, it can determine whether a vCPU yields its pCPU or busy-waits for additional time slice based on the urgent scheduling of recipient vCPUs.

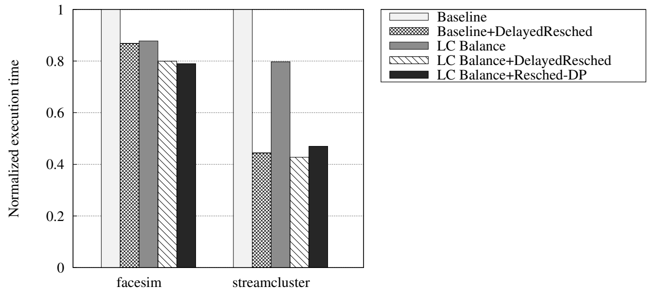


Figure 12. The performance impact of the delayed transmission of reschedule IPIs (denoted as `DelayedResched`) by means of OS modification.

5. Discussion

With regard to wait-queue LHP, OS re-engineering is a possible approach to eliminate the root cause of the problem. As explained, most wait-queue LHPs arise from reschedule IPI transmission inside a spinlock-protected critical section, since the IPI transmission can break the critical section by expensive VMEXITs while involving vCPU wake-up. In order to avoid this problem, the kernel can delay the IPI transmission until exiting the critical section. Since a reschedule IPI is not mandated to be tightly coupled with thread wake-up, it can be safely deferred right after the end of a spinlock-protected critical section, which is generally short and non-preemptable. The engineering cost, however, is nontrivial because every critical section within which thread wake-up is invoked should be modified. We simply modified futex-related wake-up procedures (*futex_wake* and *futex_requeue*), which are hot spots of wait-queue LHP, to use the delayed reschedule IPIs.

Figure 12 shows the performance of *streamcluster* and *facesim*, which are futex-intensive, in the same consolidation scenario as Section 4.1.2 (mix of a parallel and four sequential workloads). The delayed transmission of reschedule IPIs is denoted as `DelayedResched`. As expected, `DelayedResched` improves the performance of the baseline and the LC balance scheduling by excluding reschedule IPIs from critical sections protected by futex-queue spinlocks. The performance improvement is comparable to the case of our scheduling-based solution, `Resched-DP`. Based on the result, we believe that the likelihood of LHP due to IPI transmissions and VMEXITs can be effectively curtailed by virtualization-friendly OS re-engineering.

In addition, virtualization-friendly spinlocks are compelling OS re-engineering. The default spinlock implementation of the Linux kernel is *ticket spinlock*, which enforces FIFO-based lock acquisitions for fairness. The ticket spinlock does not fit for virtualized OSes, since a vCPU can excessively busy-wait for not only a preempted lock-holder, but also preempted lock-waiters that precede the vCPU in FIFO order. For this reason, traditional unfair spinlocks are considered to be efficient for virtualized OSes by allowing lock acquisitions regardless of wait order. Furthermore, as mentioned in Section 2.2, helping locks such as spin-then-block locks are more virtualization-friendly by reducing the amount of unnecessary spinning; those locks have analogous effect to hardware-assisted contention management. Although OS modification would be nontrivial task, compact and well-designed re-engineering has become highly advocated along with hypervisor-level solutions.

6. Conclusions and Future Work

This paper proposes a demand-based coordinated scheduling, which dynamically manipulates time-sharing for coscheduling and delayed preemption in response to inter-vCPU communication, IPIs. On the basis of in-depth analysis on the relationship between synchronization behaviors and IPI communications for consolidated multithreaded workloads, we argue that IPIs are effective signals for the hypervisor to coordinate vCPUs. A TLB shootdown IPI can notify the hypervisor of urgent scheduling demand of a recipient vCPU, while a reschedule IPI implies that an initiating vCPU likely holds a wait-queue spinlock, and that a recipient one possibly involves user-level synchronization. These coordination demands related to IPIs are dominantly found in emerging multithreaded applications. In addition, our load-conscious balance scheduling is essential in the situations where global loads are transiently or intrinsically imbalanced. We believe that such load imbalance could frequently happen in heterogeneously consolidated environments such as IaaS clouds and VDI.

We plan to extend our coordinated scheduling to support automatic detection of user-level synchronization demands and expand the coverage of inferring lock-holder vCPUs. Currently, we rely on a priori information about coscheduling-friendly applications with regard to user-level synchronization. We can enable the hypervisor to monitor the rate of reschedule IPIs for the inference about coscheduling-friendly workloads. In addition, as can be seen in the early preemption delay, there are more chances to infer a lock-holder vCPU that involves hypervisor intervention. We will reinforce the delayed preemption by increasing the coverage that infers lock-holder vCPUs. Finally, we are exploring cooperation with paravirtualized approaches and hardware-assisted contention management.

Acknowledgments

The authors would like to thank the anonymous reviewers for their feedback and suggestions. This work was partly supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 2012-0000148), and by the IT R&D program of MKE/KEIT [10041244, Smart TV 2.0 Software Platform].

References

[1] AMD. Amd64 architecture programmer's manual volume 2: System programming, 2010.

[2] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proc. SIGMETRICS*, 1995.

[3] A. C. Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM TOCS*, 19(3): 283–331, 2001.

[4] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with implicit information in distributed systems. In *Proc. SIGMETRICS*, 1998.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. SOSP*, 2003.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. PACT*, 2008.

[7] H. Chen, H. Jin, K. Hu, and J. Huang. Scheduling overcommitted VM: Behavior monitoring and dynamic switching-frequency scaling. *FGCS*, 2011.

[8] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. SOSP*, 1999.

[9] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *Proc. SIGMETRICS*, 1996.

[10] D. Feitelson. Gang scheduling performance benefits for fine-grain synchronization. *JPDC*, 16(4):306–318, 1992.

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

[12] Intel. Intel 64 and ia-32 architectures software developer's manual. volume 3b: System programming guide, part 2, 2010.

[13] W. Jiang, Y. Zhou, Y. Cui, W. Feng, Y. Chen, Y. Shi, and Q. Wu. CFS optimizations to KVM threads on multi-core environment. In *Proc. ICPADS*, 2009.

[14] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for I/O performance. In *Proc. VEE*, 2009.

[15] A. Kivity, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Proc. OLS*, 2007.

[16] W. Lee, M. Frank, V. Lee, K. Mackenzie, and L. Rudolph. Implications of I/O for gang scheduled workloads. In *Proc. IPPS*, 1997.

[17] Y. Lee, W. Son, S. Park, G. Lee, D. Howard, and D. Slezak. Design and implementation of a locking-aware scheduler for multiprocessor environments. *Convergence and Hybrid Information Technology*, 6935: 384–390, 2011.

[18] W. Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., 2008.

[19] P. B. Menage. Adding generic process containers to the Linux kernel. In *Proc. OLS*, 2007.

[20] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. ICDCS*, 1982.

[21] M. Russinovich and D. A. Solomon. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*. Microsoft Press, 5th edition, 2009.

[22] P. Sobalvarro, S. Pakin, W. Weihl, and A. Chien. Dynamic coscheduling on workstation clusters. In *Proc. IPPS*, 1998.

[23] P. G. Sobalvarro and W. E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Proc. IPPS*, 1995.

[24] O. Sukwong and H. S. Kim. Is co-scheduling too expensive for SMP VMs? In *Proc. EuroSys*, 2011.

[25] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proc. VM*, 2004.

[26] VMWare. VMware, Inc. VMware vSphere 4: The CPU scheduler in VMware ESX 4.1. Technical report, 2010.

[27] C. Weng, Z. Wang, M. Li, and X. Lu. The hybrid scheduling framework for virtual machine systems. In *Proc. VEE*, 2009.

[28] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic adaptive scheduling for virtual machines. In *Proc. HPDC*, 2011.

[29] Y. Wiseman and D. G. Feitelson. Paired gang scheduling. *IEEE TPDS*, 14(6):581–592, 2003.

[30] Y. Yu, Y. Wang, H. Guo, and X. He. Hybrid co-scheduling optimizations for concurrent applications in virtualized environments. In *Proc. ICNAS*, 2011.