Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications

Xiaoning Ding* Phillip B. Gibbons† Michael A. Kozuch† Jianchen Shan*
*New Jersey Institute of Technology †Intel Labs Pittsburgh

Abstract

As the number of cores in a multicore node increases in accordance with Moore's law, the question arises as to what are the costs of virtualized environments when scaling applications to take advantage of larger core counts. While a widely-known cost due to preempted spinlock holders has been extensively studied, this paper studies another cost, which has received little attention. The cost is caused by the intervention from the VMM during synchronization-induced idling in the application, guest OS, or supporting libraries—we call this the *blocked-waiter wakeup (BWW)* problem.

The paper systematically analyzes the cause of the BWW problem and studies its performance issues, including increased execution times, reduced system throughput, and performance unpredictability. To deal with these issues, the paper proposes a solution, *Gleaner*, which integrates idling operations and imbalanced scheduling as a mitigation to this problem. We show how *Gleaner* can be implemented without intrusive modification to the guest OS. Extensive experiments show that *Gleaner* can effectively reduce the virtualization cost incurred by blocking synchronization and improve the performance of individual applications by 16x and system throughput by 3x.

1 Introduction

Virtualized environments are ubiquitous, and are increasingly run on multicore nodes, particularly in the cloud. Amazon EC2's *CC2* and *CR1* instances, for example, offer 32 virtual CPUs (vCPUs) running on two 8-core Intel® Xeon® E5-2670 processors with hyper-threading [2]. When computational workloads execute in these virtualized environments, each "guest" operating system (OS) is presented with a VM instance comprised of a set of vCPUs on which it schedules application threads. The virtual machine manager (VMM), or hypervisor, independently schedules the virtual CPUs onto the physical CPUs (pCPUs) of the host machine. The VMM is often required to time-share the pCPUs among co-running VMs, and may deschedule a vCPU belonging to one VM in favor of a vCPU belonging to another VM. Unfortunately, the resulting behavior of the vCPU abstraction does not always match the behavior of physical CPUs for which the applications and OS are designed. In particular, applications and OS may expect that busy CPUs can make continuous progress in parallel and that idle CPUs are ready for use immediately.

The mismatch between the vCPU abstraction and pCPU behavior introduces great challenges to synchronization and causes serious performance issues, particularly for multithreaded applications running on multicore VMs. One such issue, which has been extensively studied [6, 24, 26], is known as the Lock-Holder Preemption problem (LHP). LHP surfaces when, for example, a vCPU is descheduled from the host platform while the thread currently executing on that vCPU is holding a lock. Other threads running in that VM that are waiting on the lock may be prevented from making progress until the descheduled vCPU is rescheduled, even though their vCPU resources are active. Several software [6, 24, 26] and hardware (such as the pause-loop-exiting (PLE) support on Intel® processors [21]) solutions have been proposed to mitigate this issue.

As the number of cores per socket continues to increase in accordance with Moore's law, though, a natural question arises: *Does the mismatch between the vCPU abstraction and real hardware CPUs impose additional "hidden" performance issues associated with synchronization that may prevent multithreaded applications from taking full advantage of larger core counts?*

The Blocked-Waiter Wakeup Problem. One such issue can be viewed somewhat as the dual of the LHP problem; we call this issue the *Blocked-Waiter Wakeup (BWW)* problem. The BWW problem may arise any-

time that a multithreaded application using blocking synchronization executes in a virtualized environment, and it can cause increased execution times, reduced system throughput, and performance unpredictability.

In non-virtualized systems, when an application's thread blocks waiting for a resource to be freed, the CPU resource occupied by that thread is typically returned to the guest OS; if the OS has no other work to schedule onto the CPU, it may halt that CPU, allowing it to enter a low-power state. Later, when the resource is freed, the OS wakes the halted CPU by a signal from another CPU, for example, by issuing an Inter-Processor Interrupt (IPI). Unfortunately, when this occurs in a virtualized environment, the idle vCPU is not simply in a halt state, waiting to be awoken at a moment's notice. Instead, it has often been de-scheduled, and the wakeup IPI, which is comparatively lightweight on physical hardware, must now cause a VM trap, invoke the VMM scheduler, and cause the blocked vCPU to be rescheduled. Furthermore, unlike an idle physical core, which is ready for immediate use once awoken, an idle vCPU can be delayed waiting for a pCPU to free up.

As we show, the increased latency associated with this wakeup path can significantly increase the execution time of virtualized multithreaded applications (e.g., by 517% for dedup on 16 pCPUs) relative to their unvirtualized performance, even when the application's VM has dedicated use of the underlying physical hardware and does not use emulated resources. Note that the offending synchronization may not even be explicit at the application level—in our experiments, we found that the problematic blocking synchronization may arise in the guest OS code.

Our Solution: Gleaner. To mitigate the detrimental performance effects of the BWW problem, we propose an approach, called *Gleaner*, which consolidates short idle periods on multiple vCPUs into long idle periods on fewer cores, thereby lessening the frequency that vCPUs enter/exit idle loops. Two key insights motivate this as a solution. First, applications vulnerable to the BWW problem are likely to see many such idle-busy cycles and, hence, may be under-utilizing their CPU resources. Second, activating/switching threads at the OS level within the VM is much lower overhead than activating/switching vCPUs at the VMM level outside the VM. Our experiments with a prototype *Gleaner* implementation indicate that this approach significantly mitigates the BWW problem.

To date, the BWW problem has been under-studied. It was first discussed briefly as part of a broader technical report by Song, *et al.* [22] and then expanded by our previous short work [4]. This paper represents the first full work dedicated to the problem and makes the following contributions: (1) a systematic characterization of the BWW problem, an important issue for virtualized multicore systems, (2) the design of an effective approach, *Gleaner*, for mitigating the BWW problem, and (3) an experimental validation of *Gleaner*'s effectiveness by demonstrating improvements of application performance by up to 16x and improvements of system throughput by up to 3x.

Table 1: Performance of dedup under *Native* (unvirtualized), *Dedicated* (virtualized, no other load), and *Shared* (two equal sized VMs) configurations.

	cores	run time (s)	Slowdown Relative to Native	Slowdown Relative to Dedicated
Native	1	23.5	–	–
	4	7.1	–	–
	16	7.6	–	–
Dedicated	1	26.4	1.1	–
	4	13.5	1.9	–
	16	46.9	6.2	–
Shared w/ streamcluster	4	52.3	7.4	3.9
	16	81.6	10.7	1.7
Shared w/ matmul	4	65.1	9.2	4.8
	16	577.2	75.9	12.3

2 Motivating Example

Table 1 presents an illustrative example of the above virtualization problems with the dedup benchmark from the PARSEC-3.0 suite. We measured the performance[1] of dedup under three settings. In the *Native* setting, dedup executed alone on physical hardware. In the *Dedicated* hardware setting, dedup ran in a VM with a dedicated pCPU allocated to each vCPU. In the *Shared* hardware setting, dedup executed in a VM sharing hardware resources with another VM (in which either streamcluster or matmul ran[2]), both sized to occupy the entire host.

The table shows that the virtualization penalty increases as the number of vCPUs scales up– reaching a factor of 6.2 for 16 cores (*Dedicated* scenario). We observe similar trends for other PARSEC benchmarks, though the penalties are not as dramatic (Section 7.1).

The performance of the *Shared* setting indicates that dedup suffers even more due to hardware resource contention than one might expect. In the experiment, each of the two VMs has the same number of vCPUs as the number of pCPUs and contend for all the hardware CPU resources. With two VMs of the same size competing for the same set of resources, one may expect the slowdown to be approximately 2X relative to the correspond-

[1] Full details of the experimental setup used throughout this paper appear in Section 7.

[2] streamcluster is another benchmark in PARSEC-3.0. matmul is a micro-benchmark multiplying two matrices of 8000×8000 integers.

ing *Dedicated* setting by assuming that each VM would get half of the physical CPU time. However, as shown in the table, the slowdown factor is typically more than 2X and can reach as high as 12.3X relative to *Dedicated*. The table also shows that the degree to which dedup slows down can be greatly affected by the application running in the other VM. Note also that the 16-vCPU execution times differ for the two co-running applications significantly more than they do in the 4-vCPU experiments. We explore these effects more deeply in Section 7.2.

3 Analysis of the BWW Problem

We investigated the possible causes for the performance degradation and variation, and discovered that the applications suffering the most were the ones in which vCPUs were frequently idling due to blocking synchronization in either the benchmarks or the guest system software (OS or supporting libraries).

There are two basic types of inter-thread synchronization primitives: *spinning*, where a waiting thread repeatedly checks some condition to determine if it can continue (possibly remaining in user space), and *blocking*, where a waiting thread yields its execution resources and relies on system software to wake it up when it can continue executing. Often, synchronization libraries combine the two approaches: a thread spins for a brief period of time, and if the desired condition has not been satisfied, the thread blocks.

One effect of blocking synchronization is that the number of *active* application threads may change dynamically, and consequently, the number of cores actively employed by that application may change accordingly. When the number of active threads drops below the number of active cores, some cores will become idle. When the number of active threads increases beyond the number of active cores, idle cores must be activated. For example, when a thread calls *pthread_mutex_lock()* to request a lock that is held by another thread, it will block itself through appropriate library/system calls, waiting for the release of the lock. If there are no other threads ready to run in the system, the core running the thread becomes idle. With conventional OS design, an idle core executes the idle loop, which typically calls a special instruction (e.g., `HLT` on Intel® 64 and IA-32 architecture ("x86") platforms) that may lead to the core entering a low power state. When the lock is released, the threads waiting for it are woken up. To maximize throughput, the OS may activate idle cores to schedule waking threads onto them.

In a virtualized environment, some of the operations executed by guest software during blocking synchronization routines must be handled by the VMM, even though they would be carried out directly by hardware in a non-virtualized environment. When software issues the special instruction to place a particular core in the idle state, that processor will raise an exception and trap into the VMM. The VMM may take this opportunity to reschedule other vCPUs—perhaps from other VMs—onto the idling physical core. Thus, the "low power" mode for a vCPU is actually the suspension of its execution. When a thread is again ready to run on that vCPU, the VMM must activate the vCPU by rescheduling it onto a physical core. This suffers much higher cost than it does in a non-virtualized environment, in which switching a core back from low power mode can be very fast. For example, switching from C1 to C0 states takes less than 1 µs on contemporary Intel® Xeon® CPUs.[3]

Table 2: Time to wake up a thread on a physical machine (PM) and a virtual machine (VM) under different settings. The last column shows in parentheses the major operations needed to wake up a thread in a VM: RT=rescheduling thread, IPI=handling reschedule IPI, RV=rescheduling vCPU, and PV=preempting a vCPU.

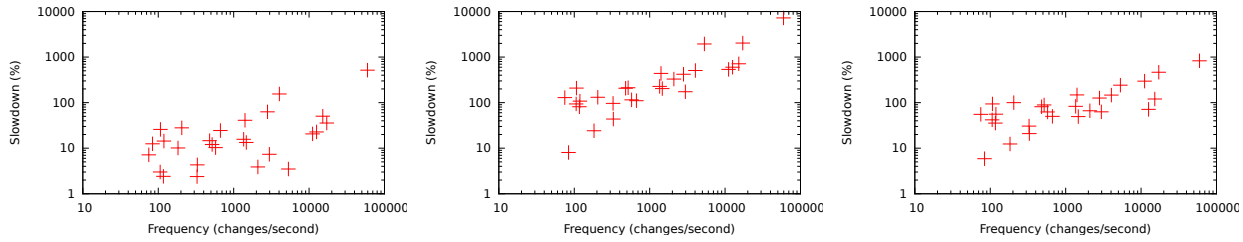
Setting	PM	VM
A: same core	4 µs	6 µs (RT)
B: diff cores, spinning	8 µs	17 µs (IPI, RT)
C: diff cores, blocking	8 µs	37 µs (IPI, RV, RT)
D: diff cores, blocking (2 apps or 2 VMs)	17 µs	>96 µs (IPI, PV, RV, RT)

This is the heart of the Blocked-Waiter Wakeup (BWW) problem: in a virtualized environment, the increased cost of thread wakeup operations may significantly degrade overall performance. To see this, we first show the increased cost of wakeup operations (Section 3.1) and then the correlation between performance and the frequency of idleness transitions (Section 3.2).

3.1 Blocking Synchronization Cost in VMs

To understand the cost of blocking synchronization in VMs and analyze how the cost is increased by switching and scheduling vCPUs, we report the time to wake up a thread blocked in *pthread_mutex_lock()* on both a physical machine (PM) and a virtual machine (VM) under four different settings, as shown in Table 2. In setting *A*, the thread calling *pthread_mutex_unlock()* and the thread blocked in *pthread_mutex_lock()* are pinned to the same core on the PM or to the same vCPU in the VM. In the other three settings, the two threads are pinned to different cores on the PM or different vCPUs in the VM. Thus when the thread is blocked in *pthread_mutex_lock()*, the corresponding core/vCPU will become idle. In setting *B*,

[3] Waking up a core from deep sleep modes (e.g., C3 and C4 states) can take more than 100 µs. But these modes are not used unless the system ensures that the core will stay idle for a long time. Waking up a core from C1E state takes about 10 µs, but system administrators often disable the state for better performance [27].



(a) setting *C* (b) setting *D* with matmul (c) setting *D* with freqsleep

Figure 1: Correlation between the performance degradation of the SPLASH-2X and PARSEC-3.0 benchmarks when virtualized and the frequency at which vCPUs transition to idle (when run alone).

lower power modes are disabled by keeping idle physical cores polling on the PM and by running low priority threads repeatedly calling *sched_yield()* on the VM. In setting *C*, lower power modes are enabled. Specifically, when a physical core becomes idle it enters C1 state, and when a vCPU becomes idle it calls `HLT` to suspend itself. In setting *D*, we measure wake-up times when another application is contending for CPU resources. In the PM experiment, we run a matmul thread on every core. In the VM experiment, we run a second VM with a matmul thread on every vCPU; the number of vCPUs in each of the two VMs is same as the number of physical cores.

Table 2 clearly demonstrates that virtualization significantly increases the cost of blocking synchronization. Waking up a thread in a VM has different costs under the four settings due to the different operations involved. Under setting *A*, waking up a thread in the VM involves only a context switch between the threads inside the guest OS. Thus, it incurs similar overhead as on the physical machine. Under setting *B*, waking up a thread on a vCPU is initiated by an IPI (inter-processor interrupt) made by another vCPU in the same VM (e.g., the one observing the *pthread_mutex_unlock()*). In a non-virtualized environment, the IPI is delivered by hardware, but in a virtualized environment, the VMM must intercept and deliver the IPI. Thus, waking up a thread on a VM incurs higher overhead than on a PM. Under setting *C*, waking up the thread in the VM takes 37 µs, **4.6** times the latency for the same operation on the PM. Under setting *D*, waking up the thread in the VM takes the longest time among these settings, at least **2.6** times longer than under setting *C* and **5.6** times longer than either on the PM or waking up a thread on an active vCPU (setting *B*). Waking up a thread in the VM requires a complete switch between vCPUs from different VMs — suspending a vCPU running matmul, activating and rescheduling the vCPU to run the thread calling *pthread_mutex_lock*. A complete vCPU switch in setting *D* incurs a higher cost than resuming a vCPU on an idle physical core in setting *C* not only because it is between different VMs, but also because the working set of a vCPU may be evicted from the pCPU caches and TLBs when it is de-scheduled. The time (96µs) is measured when the vCPU switch takes place immediately after *pthread_mutex_unlock* is called. Depending on vCPU scheduling policies, the vCPU switch may be delayed, further increasing the time to wake up a blocked thread.

3.2 Problems Caused by Accumulated Blocking Synchronization Cost

The runtime overhead incurred by blocking synchronization in a virtualized environment increases with the frequency of active-idle state transitions of application threads, and in particular, state transitions of vCPUs. Such transitions arise frequently for synchronization-intensive applications. The accumulated overhead can significantly degrade performance and increase application performance variation.

To show the correlation between performance degradation and the overhead incurred by blocking synchronization in a virtualized environment, we run SPLASH-2X and PARSEC-3.0 benchmarks under settings *C* and *D*, and measured the frequencies at which vCPUs transition to idle during their executions.[4] Under setting *D*, we run one SPLASH-2X or PARSEC-3.0 benchmark in the first VM, and run either matmul or freqsleep in the second VM. In these experiments, the number of threads in each application, the number of vCPUs in each VM, and the number of physical cores are each 16.

Freqsleep creates a thread on each core, which repeatedly calls *nanosleep(1)*. We select matmul and freqsleep because they have consistent behavior during their execution. So the impact of their execution to the performance of SPLASH-2X and PARSEC-3.0 benchmarks does not change with the execution times of the benchmarks. At the same time, though both matmul and freqsleep can saturate the physical cores, they affect the performance of the benchmarks running in the first VM by different degrees.

As shown in Figure 1, there appears to be a strong correlation between the slowdowns and blocking frequencies. The correlation is more evident when the pCPUs

[4] One may also want to measure the frequencies at which application threads block, but unfortunately, such measurements are challenging because the threads may block inside the OS kernel.

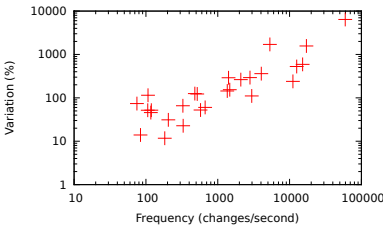


Figure 2: Correlation between the performance variation of the SPLASH-2X and PARSEC-3.0 benchmarks when virtualized and the frequency at which vCPUs transition to idle (when run alone).

are oversubscribed (setting *D*) than when they are not (setting *C*). There are some applications that experience trivial slowdowns under setting *C*, but suffer significant slowdowns under setting *D*. For example, streamcluster is slowed down by 13% under setting *C*. But, it suffers 1660% and 465% slowdowns under setting *D* when matmul and freqsleep run in the second VM, respectively. This can be explained as follows.

To correlate the performance variation of applications in VMs and the frequency at which the vCPUs in the VMs transition their status, for each benchmark, we compare its slowdowns under setting *D* when different applications (i.e., matmul and freqsleep) run in the second VM. Figure 2 shows the absolute value of the difference between the two slowdowns for each benchmark versus the frequency at which the vCPUs become idle during its execution (when run alone). It is evident that benchmarks causing more frequent vCPU status transitions usually experience larger performance variation than other benchmarks.

4 Reducing Harmful Context Switches

To reduce the impact of the BWW problem, we seek to reduce the frequency of harmful switches (those that require the activation of a de-scheduled vCPU). There are at least two possible approaches, *resource retention* and *consolidation scheduling*, and we propose to employ both in an intelligent hybrid design (Section 5).

4.1 Resource Retention

A natural approach to reducing harmful context switches is to avoid releasing resources to lower levels in the software stack. For example, a guest application thread could spin at a problematic synchronization point rather than yield to the guest OS. If the thread becomes unblocked in less time than would be required to transition into the guest OS and back, overall efficiency may be improved by spinning rather than yielding. To avoid application changes, the guest OS may also spin rather than halting. Besides spinning, the guest OS has the additional option of leveraging an operation like the x86 `MWAIT` instruction, which can place the physical core in a low-power state directly (if permitted by the VMM) such that a simple store to memory will reactivate the core.

However, such resource retention approaches must be employed with some care. Having these operations high in the software stack may lead to under-utilization. In particular, it may prevent layers lower in the stack from improving utilization by reallocating idle resources or placing those resources in a low-power state.

When a system is not oversubscribed and hardware resources are not contended, such idling operations will not hurt overall system performance as the resources were underutilized. However, when a system is oversubscribed, resource retention may prevent other VMs from making better use of those resources and reduce system throughput significantly (by as much as 8x [19]).

To improve utilization, these idling operations can be enhanced with a timeout value such that, if the thread or vCPU cannot change its state back from idling within the timeout period, the occupied resource will be released to the control of lower software layers.[5] However, the fundamental tension between improved individual VM performance and overall system throughput remains, albeit with the timeout value as a potential tuning knob.

4.2 Consolidation Scheduling

The second type of mechanism for reducing harmful context switches is based on the observation that, while context switches due to blocking synchronization may be inevitable, the resolution of such switches need not be the responsibility of lower software layers. If higher levels of software can manage the switches, application performance may be improved without adversely affecting overall throughput.

Schedulers (user-level schedulers or guest OS schedulers) determine how tasks are scheduled on execution entities (threads or vCPUs). They have direct influence on whether and when the execution entities would become idle. Thus, they can be improved to reduce the frequency at which threads or vCPUs transition between busy and idle by coalescing tasks onto fewer resources. When tasks are coalesced, the resource onto which they would be scheduled is more likely to be active.

One aspect of consolidation scheduling can be illustrated with Figures 3(a) and 3(b). In (a), the top schedule shows the guest OS scheduling each of threads 1, 2, and 3 immediately when it is ready to run. Thread 1 becomes ready and is scheduled first. It is blocked before thread 2

[5] In addition to the timeout method, an alternative approach can use heuristics to predict the length of the coming idle period and select the action that incurs lower cost: either spinning/MWAIT or yielding hardware resources to lower layers in the software stack.

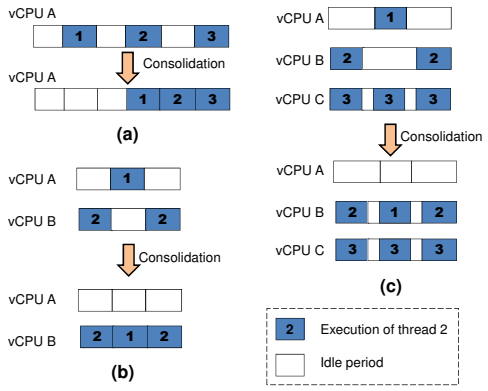


Figure 3: Consolidation scheduling in guest OS (numbers in blocks are thread IDs).

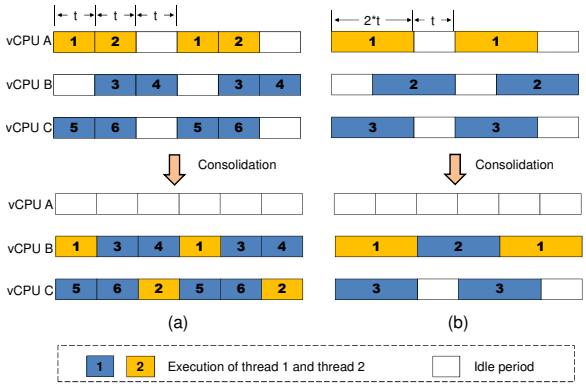


Figure 4: (a) An effective consolidation. (b) Overly-aggressive consolidation overloads vCPU B (the second block of thread 2 remains unscheduled).

becomes ready, changing the vCPU's state to idle. When thread 2 becomes ready, it is scheduled and the vCPU's state is changed back to busy. The vCPU experiences another round of state changes after thread 2 is blocked and before thread 3 becomes ready. Thus, to reduce state transitions, instead of scheduling a thread immediately when it becomes ready to run, the guest OS scheduler could choose to delay the scheduling of the thread to accumulate enough workloads to keep the vCPU busy for a while, as shown in the bottom schedule. We call this techique to achieve consolidation *delayed scheduling*.

In Figure 3(b), the top schedule shows the guest OS scheduler distributing threads 1 and 2 onto vCPUs A and B for load balance, and neither of the vCPUs can be kept always busy or always idle, increasing the number of state transitions. In contrast, *imbalanced scheduling* consolidates both threads onto one vCPU and keeps the other vCPU idle. While *delayed scheduling* is more suitable to single-vCPU VMs, *imbalanced scheduling* is more suitable to VMs with multiple vCPUs.

While consolidation scheduling tends to improve overall system throughput on an oversubscribed system, it may reduce the resources available to a particular VM and may delay computation and/or overload a busy vCPU if not carefully controlled.

5 Gleaner: Basic Idea and Design

Gleaner is a hybrid approach that combines resource retention and consolidation scheduling techniques to leverage the advantages of both techniques. Resource retention is used to mangae short periods of idleness, and consolidation scheduling is used to both coalesce tasks and reduce the number of long idle periods that resource retention cannot handle efficiently.

The combination can be illustrated with Figure 3(c). Before applying imbalanced scheduling, each vCPU runs one thread. Thread 3 running on vCPU C has short idle periods between its tasks that can be efficiently handled through resource retention without delaying the tasks. Threads 1 and 2 on vCPUs A and B have long idle periods. Holding the hardware with idling operations will incur higher cost than halting the vCPUs. With imbalanced scheduling, the threads are consolidated to vCPU B. Though they cannot fully keep vCPU B busy, running two threads on it reduces the length of idle periods, making resource retention viable. For brevity, the paper refers to the vCPUs with workload threads as *active vCPUs* and refers to other vCPUs as *deactivated vCPUs*.

To prevent active vCPUs from being overloaded as a result of overly-aggressive consolidation (Figure 4), *Gleaner* monitors the workload in a VM and collects the following time measurements: *computation length*, denoted by l_{comp}, is the length of computation on a vCPU between two consecutive idle periods; *computation granularity*, g_{comp}, is the length of computation in a thread between two consecutive synchronization points; and *length of idle periods*, l_{idle}, is the length of an idle period of a vCPU. Note that *Gleaner* dynamically adjusts the number of active vCPUs in a VM, and measurements need not be collected on deactivated vCPUs. To characterize the workload, *Gleaner* computes averages for these values. In the remainder of the paper, l_{comp}, g_{comp}, and l_{idle} refer to the corresponding average values. For example, for the workload shown on the top of Figure 4(a), l_{comp} is $2t$; both g_{comp} and l_{idle} are t.

Gleaner consolidates workload threads cautiously and gradually. It periodically updates the above measurements, and reduces the active vCPUs one at a time when the following two conditions are satisfied: (1) $l_{comp} \leq \rho \times (N-1) \times l_{idle}$ and (2) $g_{comp} \leq min(\eta \times l_{idle}, min\_time\_slice)$. Here, N is the number of active vCPUs in the VM, and the *load factor*, ρ, and *granularity factor*, η, are tunable values between 0 and 1.

The first condition is to ensure that there is enough idle time on the $N-1$ vCPUs to accommodate the computation on the vCPU to be deactivated. The second con-

dition is to ensure that the computation periods on the to-be-deactivated vCPU are small enough, such that they can be relatively evenly distributed to other active vCPUs and fit into the available idle periods. It also ensures the low overhead of moving threads. *Gleaner* only consolidates workload threads with active computation periods shorter than the minimum timeslice *min_time_slice*, which is selected by the OS to be long enough (e.g., a few milliseconds) to tolerate the overhead of rescheduling threads. For workloads with periods of active computation longer than *min_time_slice*, blocking synchronization usually cannot significantly degrade performance; but cache locality may be an important performance factor. Therefore, instead of consolidating these threads with the imbalanced scheduling technique, *Gleaner* applies resource retention to suppress short idle periods.

Figure 4 uses two examples to explain the necessity to enforce the above conditions. For simplicity, we assume η and ρ are 1 in the examples. In 4(a), the workload running on three vCPUs (as shown on the top) is to be consolidated onto two vCPUs (as shown at the bottom). It meets both conditions. The consolidation does not degrade performance. In 4(b), the workload meets the first condition ($l_{comp} = 2t$ and $l_{idle} = t$), but not the second. Thus, were thread 1 to be migrated to vCPU B, vCPU B would be overloaded, reducing application throughput.

During the consolidation, *Gleaner* keeps monitoring the vCPU utilization of the workload. When it observes a vCPU utilization decrease, which indicates the execution of the workload has slowed down, it stops the consolidation and restores the vCPU that was last deactivated. Then, it adjusts *load factor* and *granularity factor*. Specifically, if the consolidation increases the utilization of a vCPU to 100%, indicating the performance degradation may be caused by the specific vCPU being overloaded (similar to the situation in Figure 4(b)), *granularity factor* is then reduced to $0.9 \times min(\eta, g_{comp}/l_{idle})$; otherwise, *load factor* is reduced to $0.9 \times min(\rho, l_{comp}/((N-1) \times l_{idle}))$. These adjustments are to prevent more consolidation in the future that may degrade performance.

Gleaner maintains the current set of active vCPUs as long as there is not much variation of vCPU utilization and computation granularity. However, a dramatic change in vCPU utilization or computation granularity may indicate a workload change. *Gleaner* must respond to such changes by adjusting the number of active vCPUs to better satisify the resource demand of the workload. Therefore, *Gleaner* activates all the deactivated vCPU in the VM, and distributes the workload on all the vCPUs. Then, it gradually reduces active vCPUs when the above two conditions are met, until it finds a good setting. In our current implementation, a change of vCPU utilization larger than 20%, per-vCPU utilization exceeding 90%, and g_{comp} increased by more than 2x or decreased by more than 50% are each considered as indicators of major change.

6 Gleaner Implementation

A convenient place to implement *Gleaner* is the guest OS. Resource retention techniques can be implemented by modifying the idle driver, and imbalanced scheduling can be achieved by modifying the guest OS scheduler. However, these changes involve intensive modifications to the guest OS. Thus, in this section, we introduce a few techniques to implement *Gleaner* at the user level of the guest OS, to avoid intrusive kernel implementation and enable the adoption into proprietary operating systems.

At user level, idling operations in support of resource retention can be implemented by a *yielding thread* on each active vCPU. A *yielding thread* is a user-level thread that calls the *sched_yield()* system call in a loop. If there are not other threads ready to run on the vCPU, the *sched_yield()* call will return immediately. Otherwise, the *sched_yield()* call relinquishes the vCPU to other threads. Thus, the yielding thread keeps the vCPU active and does not impede the execution of application threads. On systems where the semantics of *sched_yield()* is not fully implemented (e.g., some versions of Linux kernels), the yielding threads should be assigned with the lowest priority possible in the guest OS (e.g., SCHED_IDLE scheduler class in Linux) to avoid hindering workload threads.

While a yielding thread can keep the vCPU active, to support resource retention well, it must also suspend the vCPU at appropriate times. To determine *when* the vCPU should be suspended, *Gleaner* monitors the time spent in *sched_yield()* calls to determine whether the calls return immediately. If a *sched_yield()* call spends much more time than that needed by returning immediately without relinquishing the vCPU, the finish of the call denotes the beginning of an idling operation. For time-outs, *Gleaner* accumulates the time spent in consecutive *sched_yield()* calls that return immediately and compares the time against the time-out value to determine whether a time-out should be triggered. *Gleaner* sets the time-out value of the idling operation based on the cost incurred by context transitions if hardware resources are released to lower layers in the software stack, and calls *nanosleep(1)* to effect the actual vCPU suspension.

At user level, imbalanced scheduling can be achieved by changing the CPU affinity of application threads, but an easier yet more scalable way is to leverage the resource container support on the guest OS, e.g., cgroup support on Linux or zone support on Solaris. *Gleaner* creates a resource container for workload threads. It dynamically adjusts the vCPUs assigned to the container

based on the policies described in Section 5. The workload threads will be accordingly redistributed by the guest OS scheduler on the assigned vCPUs upon every adjustment. For the vCPUs that are not assigned to the container, the *yielding threads* on them are put into sleep; thus these vCPUs will be suspended by the VMM to reclaim the physical resources.

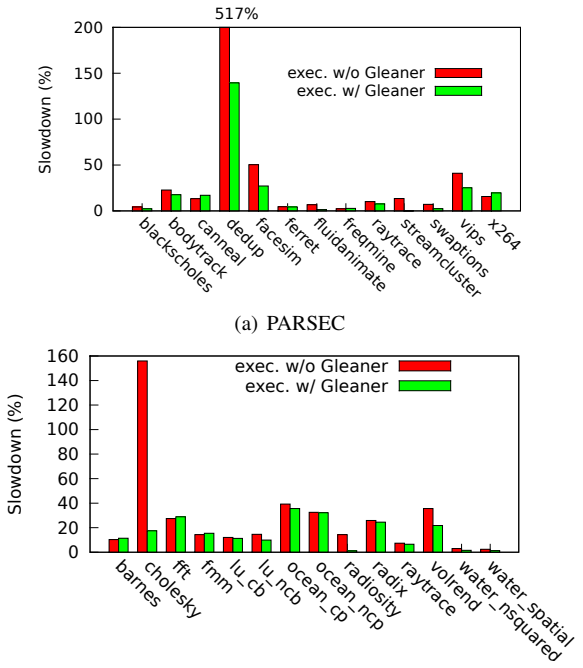
To get the measurements required by *imbalanced scheduling* (l_{comp}, g_{comp}, and l_{idle}) on active vCPUs, *Gleaner* periodically collects vCPU times spent by the yielding threads and workload threads (denoted by T_{yield} and T_{work}, respectively), as well as their number of context switches (S_{yield} and S_{work}, respectively). Then, l_{comp} is T_{work}/S_{yield}, g_{comp} is T_{work}/S_{work}, and l_{idle} is T_{idle}/S_{yield}.

7 Experiments

We evaluated our prototype implementation of *Gleaner* on a Dell™ PowerEdge™ R720 server with 64GB of DRAM and two 2.40GHz Intel® Xeon® E5-2665 processors, each of which has 8 cores. VMs were created with 16 virtual CPUs and 16GB of memory. The VMM is KVM [13], with EPT support and PLE support enabled. Both the host OS and the guest OS are Ubuntu version 12.04 with the Linux kernel version updated to 3.9.4. To prevent the performance degradation caused by CPU power management to latency sensitive applications, we disabled the C states deeper than C1 (including C1E) [28]. Please note that the change of these settings does not favor *Gleaner* and *Gleaner* can improve application performance by larger percentages when the C states are enabled.

We selected the benchmarks in PARSEC 3.0 and SPLASH 2X suites, SysBench [18], and matmul. We compiled the PARSEC and SPLASH2X benchmarks using gcc with the default settings of the *gcc-pthreads* configuration in PARSEC 3.0. We used the *parsecmgmt* tool in the PARSEC package to run them with native input and with the minimum number of threads set to 16 in the "-n" option. We used the OLTP test mode of SysBench to benchmark a MySQL database's performance on VMs with OLTP workloads.

We performed two sets of experiments. In the first set of experiments, we launch one VM on the system, and run the benchmarks in the VM. These experiments test the effectiveness of *Gleaner* on improving application performance in virtualized environment with dedicated physical resources. At the same time, we aim to confirm that the performance improvement is achieved without increasing energy consumption. In the second set of experiments, we launch two or more VMs and run one benchmark in each of the VMs. The VMs contend for hardware resources and test the capability of *Gleaner*



(a) PARSEC

(b) SPLASH2X

Figure 5: Virtualization slowdowns of PARSEC and SPLASH2X benchmarks on 16 cores with and without *Gleaner*.

to improve the performance of synchronization-intensive applications and overall system throughput on an oversubscribed system.

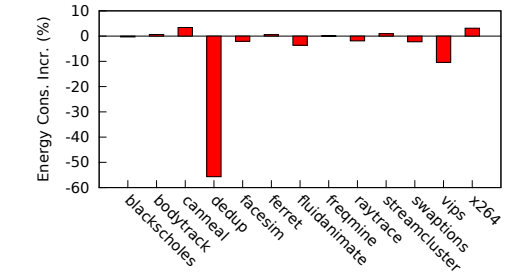
7.1 Single VM Experiments

We first execute the benchmarks in a single virtual machine running alone. We execute each benchmark in the following three scenarios: in the host OS, in the guest OS without Gleaner enabled, and in the guest OS with Gleaner enabled. In Figure 5, we report the slowdowns of the benchmark in the latter scenarios relative to its execution in the first scenario.

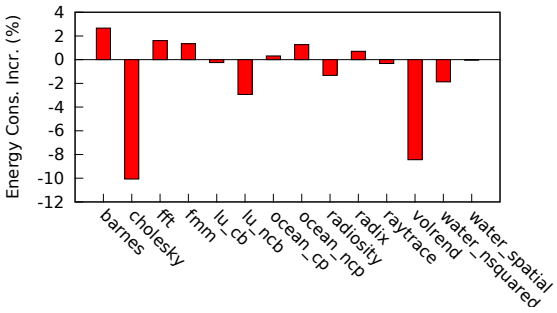
As shown in the figure, *Gleaner* is especially effective for the benchmarks suffering from the high blocking synchronization overhead. For example, virtualization slows down PARSEC's dedup and facesim benchmarks by 517% and 51%, respectively. With *Gleaner*, the slowdowns were reduced to 138% and 27%. The SPLASH2X benchmarks cholesky and volrend slow down by 155% and 35% without *Gleaner* but only by 19% and 21%, respectively, with it enabled.

For benchmarks that are not sensitive to blocking synchronization overhead, such as PARSEC's ferret and freqmine and SPLASH2X's water_nsquared and water_spatial, *Gleaner* neither improves nor degrades their performance, indicating that its overhead is very low.

For other benchmarks, *Gleaner* may slightly reduce or



(a) PARSEC



(b) SPLASH2X

Figure 6: Energy consumption increases of PARSEC and SPLASH2X benchmarks on 16 cores with *Gleaner*. Negative bars indicate energy consumption reduction.

increase their execution times. For example, the execution times of bodytrack and lu_ncb reduce by 6% and 5%, respectively, and *Gleaner* slightly increases the execution times of canneal (3%) and x264 (5%).

Gleaner may increase the execution times because, implemented at user level, it may not be able to quickly detect the increasing concurrency levels at the beginning of some new execution phases, and thus it cannot promptly adjust the number of active vCPUs to maximize throughput. The problem could be addressed with some assistance from the guest OS. For example, when threads are woken up (i.e., the concurrency level increases), the guest OS could notify *Gleaner* to increase the number of vCPUs.

On average, without *Gleaner*, the execution times of the PARSEC benchmarks and SPLASH2X benchmarks slow down by 55% and 30% when virtualized, relative to native execution, but with the tool enabled, the average slowdowns were reduced to 20% and 17%, respectively.

Gleaner uses yielding threads to keep some vCPUs busy even when they do not have any threads to run. This potentially increases energy consumption; however, the energy consumed by yielding threads can be justified if a reduction in execution time results, which in turn can be translated to reduced energy consumption. Moreover, *Gleaner* consolidates application threads and adjusts the number of active vCPUs to suppress the energy consumed by yielding threads. This significantly reduces the energy consumed by yielding threads.

To test whether or not *Gleaner* increases energy consumption, we used the IPMI OEM utility to measure the energy consumption of the system during the execution of each benchmark, and compared the energy consumption in the last two scenarios. The energy consumption increases are as shown in Figure 6. The data show that, although *Gleaner* may slightly increase the energy consumption for some benchmarks, it reduces energy consumption for many, especially for the benchmarks suffering from blocking synchronization overheads. Energy consumption is reduced primarily by reducing execution times: the benchmarks with larger execution time reductions usually show larger energy consumption reductions. The reductions in energy consumption are not proportional to reductions in execution time because yielding threads increase the power consumption when *Gleaner* is enabled. On average, *Gleaner* reduces the energy consumption by 5% for PARSEC benchmarks and by 1% for SPLASH2X benchmarks.

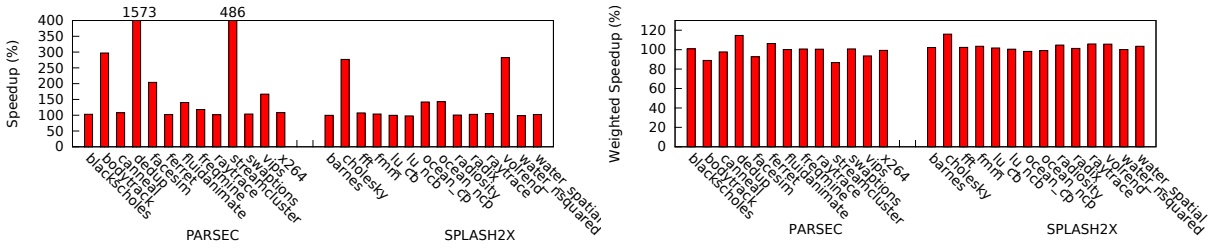
7.2 Experiments with Co-Running VMs

In this subsection, we present the experimental results when the system is oversubscribed. We launch two virtual machines, one with the PARSEC or SPLASH2X benchmark under test and one with matmul running repeatedly. As shown in Figure 1(b), synchronization-intensive benchmarks suffer much higher slowdowns than they do on a VM with dedicated hardware. With the first part of the experiments, we show that *Gleaner* can effectively speed up these applications on oversubscribed systems. Then, in the second part of the experiments, we show that *Gleaner* can improve system throughput by reducing the overhead caused by vCPU switches.

7.2.1 Reducing Application Execution Times

On an oversubscribed system, *Gleaner* improves the performance of synchronization-intensive applications by preventing hardware resources from being taken by other VMs if the resources are to be used soon. In the experiments, we use the application performance in the VM without *Gleaner* enabled as a baseline. In Figure 7(a), we show the speedups of the PARSEC and SPLASH2X benchmarks when *Gleaner* is enabled, relative to the baseline. In addition to the performance of individual applications, we also want to investigate how the idling operations affect system throughput. Thus, we use *Weighted-Speedup* to measure the system throughput, which is the average speedups of the benchmark and matmul, relative to their performance when *Gleaner* is disabled. The system throughput when *Gleaner* is disabled is always 1. Thus, in Figure 7(b), we only show the throughput of the system when *Gleaner* is enabled.

For the benchmarks that suffers significantly from



(a) speedup (b) throughput

Figure 7: Speedup and system throughput of PARSEC and SPLASH2X benchmarks on a 16-vCPU VM with *Gleaner* on the oversubscribed system.

the overhead of blocking synchronization, e.g., dedup, streamcluster, bodytrack, cholesky, and volrend, *Gleaner* can dramatically improve their performance. It reduces the execution times by several factors (up to 16x for dedup). For benchmarks that are not sensitive to the overhead of blocking synchronization, e.g. blackscholes, ferret, and freqmine, *Gleaner* does not reduce their performance. For all the selected benchmarks, the average speedup is 203% when *Gleaner* is enabled.

The improvement of application performance does not come without cost. Idling operations may reduce efficiency because they prevent hardware resources from being utilized by other vCPUs. But they may also improve efficiency by reducing excessive costly vCPU switches. Therefore, we observed that *Gleaner* improves throughput for some workloads and reduces throughput for others. Generally, the system has similar throughputs when *Gleaner* is enabled (the average throughput is 2% higher than when *Gleaner* is disabled).

We notice that if the performance degradation of the benchmark is due to frequent barrier synchronization or condition variable synchronization, *Gleaner* usually reduces system throughput (e.g., bodytrack, streamcluster, and vips). This is because *Gleaner* usually cannot consolidate the threads of the application. The performance of such applications is more sensitive to the delay of its computation than other applications. The delay caused by *imbalanced scheduling* may degrade the application performance. For example, delaying the computation of one thread and making it the last one reaching a barrier will effectively delay all the other threads waiting at the barrier. In contrast, for the threads contending for a mutex, if the computation of a thread is delayed and it reaches a synchronization point late, the delay will not block other threads. When *imbalanced scheduling* is not used to reduce the length of idle periods, *Gleaner* cannot effectively minimize the cost of the idling operations handling these idle periods, which in turn reduces system throughput.

On the other hand, if the performance degradation of a benchmark is due to frequent mutex synchronization (e.g., dedup, ferret, and cholesky), *Gleaner* usually can

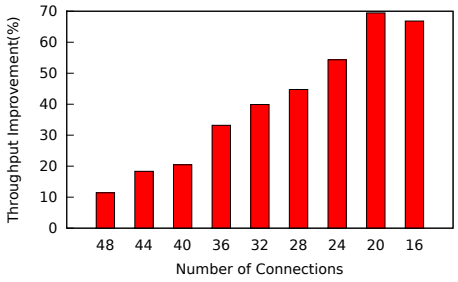


Figure 8: Throughput improvements of MySQL servers driven by the OLTP workload generated by SysBench.

increase the system throughput, albeit the gains are less than 15%. The reason for the modest gains is that, when the VM of these benchmarks run with the VM of matmul, the vCPU switches are not frequent, because the vCPU scheduling policy of KVM enforces a minimum time slice for the vCPUs running matmul. Thus, there is limited potential for *Gleaner* to improve performance.

We replaced matmul with freqsleep and repeated the above experiments to compare the performance of the benchmarks under these two settings. The benchmarks showed similar performance after replacing matmul with freqsleep. Even for dedup, which suffered the largest performance variation when *Gleaner* was not enabled (its execution time was 7x longer co-running with matmul than with freqsleep), after *Gleaner* was enabled its execution co-running with matmul was only 21% longer than with freqsleep. For the PARSEC and SPLASH2X benchmarks, with *Gleaner* enabled, the average execution time was 15% smaller when freqsleep replaces matmul, while the number is 3X with *Gleaner* disabled.

7.2.2 Improving System Throughput

With some commercial workloads and scientific workloads, frequent vCPU switches can significantly degrade system throughput. For such workloads, *Gleaner* can effectively reduce vCPU switches and improve system throughput. To demonstrate this capability of *Gleaner*, we select dedup and the SysBench OLTP benchmark, which was designed for MySQL server benchmarking.

First, we use SysBench to generate the OLTP work-

load to drive the MySQL database servers running in two VMs. The workload consists of a mixture of back-to-back transactions on a table with 1 million records as specified with SysBench OLTP "advanced transactional" test mode. In the experiment, we change the number of connections between SysBench and MySQL to vary the workload. Since each connection is backed by a MySQL server thread, when we reduce the number of connections, the number of server threads on each vCPU is also reduced; thus the chance for a vCPU becoming idle increases. Figure 8 shows that the throughput of MySQL servers is improved by increasingly larger percentages (upto 69%) by enabling *Gleaner* on the VMs running MySQL, when the number of connections is decreased from 48 to 16, with a peak at 20.

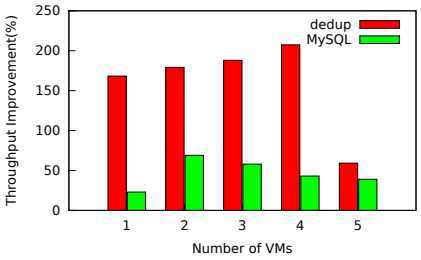


Figure 9: Throughput improvements of dedup and MySQL varying the number of VMs.

Finally, we vary the number of VMs running on the physical machine and test how the throughput improvements change with dedup and MySQL (the number of connections to each MySQL instance is 20). When the the number of VMs is increased to 5, the application performance is significantly degraded relative to that with one VM. For example, the average response time of the MySQL servers is increased by about 3X. Thus, we did not further increase the number of VMs. As shown in Figure 9, for all the above settings, *Gleaner* can substantially improve throughput. On average, it improves the throughput by 160% for dedup and 45% for MySQL database server.

We expect that the improvements increase with the number of VMs, because the contention for hardware resources increases with more VMs. As shown in Figure 9, dedup shows this trend before the number of VMs is smaller than 5. When the number of VMs is increased to 5, *Gleaner* cannot improve the throughput as much as it does with fewer VMs due to memory overcommitment. For the OLTP workload generated by SysBench, we observe that the throughput improvement gradually reduces when the number of VMs increases from 2 to 5. This is because when more VMs share the same physical cores, each VM has fewer active vCPUs, which in turn alleviates the mutex contention in MySQL servers, making the server threads less likely to be blocked.

8 Related Work

While there are a number of studies identifying performance overheads of virtualized execution [1, 6, 7, 8, 14, 15, 16, 17, 25, 26], most of them focus on the overhead incurred by I/O operations and spinlock synchronization. To reduce the performance degradation caused by spinlock synchronization in virtual machines, a few approaches have been proposed, including vCPU scheduling approaches [5, 6, 12, 20, 23, 24, 26], hardware approaches [21, 29], and improved spinlock design [19]. Flex also addresses the fairness issue of scheduling multicore VMs [20]. None of these studies identify and address the performance degradations caused by blocking synchronizations in multicore virtualized environments (the BWW problem).

The blocked-waiter wakeup problem was also described in a technical report [22] and a workaround was proposed to run a user-level idle daemon to avoid halting vCPUs, which is similar to yielding threads in our solution. The workaround helps improve performance for VMs with dedicated hardware (at the cost of increased energy consumption). But it causes performance degradation when the system is oversubscribed, similar to that caused by the LHP problem. This paper systematically analyzes the issues with the BWW problem and provides an efficient and universal solution.

The BWW problem, as well as the LHP problem, is caused by the lack of coordination between the vCPU scheduler in the VMM and the task scheduler in the guest OS. Thus, one solution is to enforce the collaboration between the schedulers using techniques similar to scheduler activations [3]. However, this approach requires intensive modifications to both the VMM and the guest OS. Another approach is to minimize vCPU scheduling by assigning one runnable vCPU to each pCPU and making other vCPUs offline to computation in guest OSes [23]. This approach avoids vCPU preemption and can effectively address the LHP problem. Making some vCPUs offline may be able to reduce the idle time on the online vCPUs and reduce the chance for them to become idle. But it may not be effective for the BWW problem, because every vCPU state transition from busy to idle still incurs a switch between vCPUs and the number of online vCPUs in each individual VM may not be adjusted in a way to reduce idle time.

Retaining idle resources for anticipated usages is a common scheduling technique in system designs [9, 10, 11]. It avoids the high overhead associated with resource reallocation and state switches. At a high level, the resource retention techniques in *Gleaner* share a similar idea with these designs.

A preliminary version of this paper appeared in HotCloud'13 [4].

9 Conclusion and Future Work

This paper identifies an understudied problem for running multithreaded applications in virtualized multicore environments. Namely, the costs incurred by blocking synchronization in virtualized environments can exact a significant performance penalty when scaling multicore applications to take advantage of larger and larger core counts. This paper proposes and designs *Gleaner* as a solution, which combines resource retention approaches with idling operations and consolidation scheduling. Extensive experiments show that *Gleaner* can significantly improve application performance and system throughput in virtualized environments. It can also reduce application performance variations when multiple VMs share the same physical resources.

As future work, we want to test and extend our approach to reduce the overhead due to vCPU state transitions caused by operations other than blocking synchronization. For example, SSD access latencies are typically tens of microseconds in Amazon EC2 instances. The state transitions of vCPUs when they are waiting for I/O can double the I/O latency and reduce I/O throughput at the same time. It seems that *Gleaner* could be adapted to reduce this extra cost incurred by vCPU state transitions.

Acknowledgments. We thank the anonymous reviewers for their constructive comments, and Dr. Haibo Chen for his helpful suggestions as the shepherd for this paper. This research was supported in part by the Intel Science and Technology Center for Cloud Computing.

References

[1] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *ACM ASPLOS 2006*, pp. 2–13.

[2] AMAZON, 2014. http://aws.amazon.com/ec2/instance-types/instance-details/.

[3] 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. In *ACM SOSP 1991*, pp. 95–109.

[4] DING, X., GIBBONS, P. B., AND KOZUCH, M. A. A hidden cost of virtualization when scaling multicore applications. In *HotCloud* (2013).

[5] DRUMMONDS. Co-scheduling SMP VMs in VMware ESX server, 2008. http://communities.vmware.com/docs/DOC-4960.

[6] FRIEBEL, T., AND BIEMUELLER, S. How to deal with lock holder preemption. *Xen Summit North America* (2008).

[7] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: bare-metal performance for I/O virtualization. In *ACM ASPLOS 2012*, pp. 411–422.

[8] HAN, J., AHN, J., KIM, C., KWON, Y., CHOI, Y.-R., AND HUH, J. The effect of multi-core on HPC applications in virtualized systems. In *Euro-Par 2010*, pp. 615–623.

[9] IRANI, S., SHUKLA, S., AND GUPTA, R. Algorithms for power savings. In *ACM-SIAM SODA 2003*, pp. 37–46.

[10] IYER, S., AND DRUSCHEL, P. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *ACM SOSP 2001*, pp. 117–130.

[11] KARLIN, A. R., LI, K., MANASSE, M. S., AND OWICKI, S. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *ACM SOSP 1991*, pp. 41–55.

[12] KIM, H., KIM, S., JEONG, J., LEE, J., AND MAENG, S. Demand-based coordinated scheduling for SMP VMs. In *ACM ASPLOS 2013*, pp. 369–380.

[13] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), pp. 225–230.

[14] LANDAU, A., BEN-YEHUDA, M., AND GORDON, A. SplitX: split guest/hypervisor execution on multi-core. In *USENIX WIOV 2011*, pp. 1–7.

[15] LANGE, J. R., PEDRETTI, K., DINDA, P., BRIDGES, P. G., BAE, C., SOLTERO, P., AND MERRITT, A. Minimal-overhead virtualization of a large scale supercomputer. In *ACM VEE 2011*, pp. 169–180.

[16] LUSZCZEK, P., MEEK, E., MOORE, S., TERPSTRA, D., WEAVER, V. M., AND DONGARRA, J. Evaluation of the HPC challenge benchmarks in virtualized environments. In *Euro-Par 2011*, pp. 436–445.

[17] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *ACM VEE 2005*, pp. 13–23.

[18] MYSQL AB. SysBench manual, 2010.

[19] OUYANG, J., AND LANGE, J. R. Preemptable ticket spinlocks: Improving consolidated performance in the cloud. In *ACM VEE 2013*, pp. 191–200.

[20] RAO, J., AND ZHOU, X. Towards fair and efficient smp virtual machine scheduling. In *ACM PPoPP 2014*, pp. 273–286.

[21] RIGHINI, M. Enabling Intel® virtualization technology features and benefits. Tech. rep., Intel, 2010.

[22] SONG, X., CHEN, H., ZANG, B., SONG, X., CHEN, H., AND ZANG, B. Characterizing the performance and scalability of many-core applications on virtualized platforms. Tech. Rep. FDUPPITR-2010-002, Parallel Processing Institute, Fudan University, 2010.

[23] SONG, X., SHI, J., CHEN, H., AND ZANG, B. Schedule processes, not VCPUs. In *APSys 2013*, pp. 1:1–1:7.

[24] SUKWONG, O., AND KIM, H. S. Is co-scheduling too expensive for SMP VMs? In *EuroSys 2011* (2011), ACM, pp. 257–272.

[25] TICKOO, O., IYER, R., ILLIKKAL, R., AND NEWELL, D. Modeling virtual machine performance: Challenges and approaches. *SIGMETRICS Perform. Eval. Rev. 37*, 3 (Jan. 2010), 55–60.

[26] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards scalable multiprocessor virtual machines. In *VM 2004*, pp. 43–56.

[27] VMWARE. Host power management in VMware vSphere 5, 2010. http://www.vmware.com/files/pdf/hpm-perf-vsphere5.pdf.

[28] VMWARE, 2013. http://www.vmware.com/resources/techresources/10205.

[29] WELLS, P. M., CHAKRABORTY, K., AND SOHI, G. S. Hardware support for spin management in overcommitted virtual machines. In *PACT 2006*, ACM, pp. 124–133.