USENIX Association

Proceedings of the Third Virtual Machine Research and Technology Symposium

San Jose, CA, USA
May 6–7, 2004

Towards Scalable Multiprocessor Virtual Machines

Volkmar Uhlig Joshua LeVasseur Espen Skoglund Uwe Dannowski

System Architecture Group
Universität Karlsruhe

contact@l4ka.org

Abstract

A multiprocessor virtual machine benefits its guest operating system in supporting scalable job throughput and request latency—useful properties in server consolidation where servers require several of the system processors for steady state or to handle load bursts.

Typical operating systems, optimized for multiprocessor systems in their use of spin-locks for critical sections, can defeat flexible virtual machine scheduling due to lock-holder preemption and misbalanced load. The virtual machine must assist the guest operating system to avoid lock-holder preemption and to schedule jobs with knowledge of asymmetric processor allocation. We want to support a virtual machine environment with flexible scheduling policies, while maximizing guest performance.

This paper presents solutions to avoid lock-holder preemption for both fully virtualized and paravirtualized environments. Experiments show that we can nearly eliminate the effects of lock-holder preemption. Furthermore, the paper presents a scheduler feedback mechanism that despite the presence of asymmetric processor allocation achieves optimal and fair load balancing in the guest operating system.

1 Introduction

A recent trend in server consolidation has been to provide virtual machines that can be safely multiplexed on a single physical machine [3, 7, 24]. Coupling a virtual machine environment with a multiprocessor system furthers the trend of untrusted server consolidation.

A multiprocessor system offers many advantages for a virtualized environment. The hypervisor, the controlling agent of the virtual machine environment, can distribute the physical processors to guest operating systems (OS) to support arbitrary policies, and reassign the processors in response to varying load conditions. The allocation policy may support concurrent execution of guests, such that they only ever access a fraction of the physical processors, or alternatively time-multiplex guests across a set of physical processors to, e.g., accommodate for spikes in guest OS workloads. It can also map guest operating systems to virtual processors (which can exceed the number of physical processors), and migrate between physical processors without notifying the guest operating systems. This allows for, e.g., migration to other machine configurations or hot-swapping of CPUs without adequate support from the guest operating system. It is important to recognize that allowing arbitrary allocation policies offers much more flexibility than schemes where one can only configure a virtual machine to either have an arbitrary share of a single processor [7, 24], or have uniform shares over multiple physical processors [10, 24].

Isolating commodity operating systems within virtual machines can defeat the assumptions of the guest operating system. Where the guest operating system expects constant resource configurations, critical timing behavior, and unrestrained access to the platform, the virtual machine provides illusionary access as it sees fit. Several methods exist to attempt to satisfy (a subset of) the assumptions of the guest operating system. The solutions may focus on the issues of instruction set emulation, such as trapping on system instructions [22], or they may focus on the behavior of the guest operating system algorithms, such as dynamic allocation of physical memory [25].

This paper presents solutions to two problems that arise with scheduling of virtual machines which provide a multi-processor environment for guest operating systems. Both problems limit scalability and performance. First, guest operating systems often use spin-locks as a means to offer exclusive access to code or data. Such spin-locks are, by design, only held for a short period of time, but if a virtual machine is preempted while holding the lock this assumption no longer holds. The crux of the problem is that the same virtual machine may still be running on a different processor, waiting for the lock to be released, thus wasting huge amounts of processor

cycles (often several milliseconds).

The second problem is due to the ability of virtual processors to offer the illusion of varying speed. Today's operating systems cannot react well to multi-processor systems where otherwise identical CPUs have asymmetric and varying clock speeds. The problem manifests itself in suboptimal scheduling of CPU intensive workloads and burst-load handling.

To address the first problem, we have devised two techniques for avoiding preemption of lock holders, one requiring modifications of the locking primitives in the guest OS, and one in which the hypervisor attempts to determine when it is safe to preempt the virtual machine (i.e., without modifying the guest OS). Initial results suggest that our lock holder preemption avoidance schemes can increase high-load web server performance by up to 28% compared to an approach where the virtualization layer does not take guest OS spin-locks into account.

To handle asymmetric CPU speeds we propose a technique called *time ballooning* where the hypervisor coerces the guest OS to adapt scheduling metrics to processor speed. The coercion may manifest as the introduction of ghost processes into the scheduling queues of the guest OS, or as balloon processes which donate their cycles back to the virtualization layer when scheduled by the guest OS. By artificially increasing the load on a virtual CPU we pressure the guest OS into migrating processes to other virtual CPUs with more available resources.

The remainder of the paper is structured as follows. Section 2 elaborates on the problem of lock-holder preemption. Sections 3 and 4 describe our solutions with lock-holder preemption avoidance and time ballooning, followed by experimental results in Section 5. The implications of our solution and future work are discussed in Section 6. Section 7 presents related work, and finally Section 8 concludes.

2 The Case for Lock-holder Preemption Avoidance

Many of the commodity operating systems used in server consolidation have optimized support for multiple processors. A primary function of the multiprocessor support is to guarantee atomic and consistent state changes within the kernel's data structures. Typical kernels use memory barriers to ensure in-order memory updates, and they craft critical sections protected by locks to enforce atomic updates. The critical sections may be associated with a region of code, or as with more fine grained locking, they may be associated with a particular piece of data.

When the number of processors in a system increases, more processors will be competing for access to the critical sections. To achieve multiprocessor scalability it is important that the time a processor spends in a critical section is short. Otherwise, the processors trying to acquire the lock for the critical section can experience long waiting times. Designing a system for short lock-holding times makes it feasible to poll for a lock to be released (i.e., using spin-locks). Short lock-holding times may also obviate the need to implement more expensive locking primitives to enforce fair lock access, since the kernel may achieve such fairness statistically.

A very different approach to achieve multi-processor scalability in operating systems has been to avoid locking altogether by using non-blocking synchronization primitives. Although an operating system kernel can in theory be made lock free using atomic *compare-and-swap* instructions supported by many hardware architectures, it has been shown that special hardware support is needed to make lock free kernels feasible [11]. Such special hardware support has been used to implement lock-free versions of Synthesis [19] and the Cache-kernel [6], but is not applicable to commodity operating systems in general, both because of the hardware requirements and the tremendous task of rewriting large parts of the kernel internal data structures and algorithms. Some form of locking therefore seems unavoidable.

When running a commodity operating system in a virtual machine, the virtual machine environment may violate some of the premises underlying the guest operating system's spin-locks. The virtual machine can preempt the guest kernel, and thus preempt a lock holder, which can result in an extension of the lock holding time. For example, in Linux, the typical lock holding time is under 20 microseconds (see Figure 2), which a preemption can easily extend by several time slices, often in the order of tens of milliseconds.

Consequently, the main effect of lock preemption is the potential for wasting a guest operating system's time slice. If a guest kernel spins on a preempted lock, it could live out the remainder of its time slice spinning and accomplishing no work. Thus spin-times are amplified. A concomitant effect is the violation of the original statistical fairness properties of the lock.

The side effects of lock holder preemption could be avoided with coscheduling [21]. In coscheduling (or gang scheduling), all virtual processors of a virtual machine are simultaneously scheduled on physical processors, for an equal time slice. The virtual machine could preempt lock holders without side effects, since the coschedule guarantees that another processor will not spin on a preempted lock. However, coscheduling introduces several problems for scalability and flexibil-

	Povray	Kbuild	Apache 2
total	0.04%	15.3%	39.2%
average	3.0μs	1.8μs	2.2μs
max	103μs	293μs	473μs
std.dev.	5.3μs	6.7μs	7.7μs

Table 1. Lock-holding times for various Linux workloads. Hold times are measured while at least one kernel lock is being held by the CPU.

ity. Coscheduling activates virtual processors whether or not they will accomplish useful work, easily leading to underutilized physical processors. Further, coscheduling precludes the use of other scheduling algorithms, such as multiplexing multiple virtual processors on the same physical processor (e.g., in response to fault recovery of a failed processor, or load balancing).

Alternative lock wait techniques to spinning, such as reschedule or spin-then-reschedule, have successfully been applied to user applications [16], but these techniques are generally not applicable to traditional operating systems code because the kernel does not always enjoy the option of preempting its current job (e.g., if within a low-level interrupt handler). To conclude, we see that spin-locks are and will be used in commodity operating systems, and preempting lock-holders *may* as such pose a significant performance problem.

To determine whether the frequency of lock-holder preemption really merits consideration, we instrumented Linux 2.4.20 with a lock tracing facility to inspect locking statistics. Table 1 shows the results for three workloads with varying characteristics from the application spectrum. We measured the time for when a CPU holds at least one lock, on a machine with four 700MHz Intel Xeon processors. With CPU-intensive applications (povray, a ray-tracing application) we found locks being held for an average of 3.0μs and a maximum of 103μs. With an I/O-intensive workload like the Apache 2 web server under stress these numbers were 2.2μs and 473μs respectively.

From our tracing experiments we observe that the probability of preempting a virtual CPU while holding a lock lies between 0.04% for CPU-intensive workloads and 39% for I/O-intensive workloads. These numbers indicate that scheduling a virtual machine running an I/O-intensive workload without regard for guest OS spin-locks can severely impact the performance of the virtual machine. Some scheme for dealing with lock-holder preemption is therefore deemed necessary.

There are two approaches to deal with lock-holder preemption. The first approach is to detect contention on a lock and to donate the wasted spinning time to the lock holder. Also known as *helping locks*, this approach requires substantial infrastructure to donate CPU time between virtual CPUs (provided donation is possible at all) [14]. The second approach is to avoid preempting lock-holders altogether. Instead, soon to become lock-holders are preempted before acquiring a lock, or preemption is delayed until after the last lock has been released.

Depending on the level of virtual machine awareness in the guest operating systems, different methods of lock-holder preemption avoidance can be used. We discuss these methods in the following section.

3 Lock-holder Preemption Avoidance

Lock holder preemption avoidance can be achieved by either modifying the guest operating system to give hints to the virtual machine layer (intrusive), or have the virtual machine layer detect when the guest operating system is not holding a lock (non-intrusive). The former approach is well suited for systems where the virtualized architecture is not identical with the underlying hardware; also called *paravirtualization* [28]. The latter approach is well suited for fully virtualized systems.

3.1 Intrusive Lock-holder Preemption Avoidance

For the intrusive approach we use a similar scheme as implemented in Symunix II [8] and described by Kontothanassis et. al. [17]; the main difference being that the guest OS itself is here the application giving scheduling hints to the lower layer (the hypervisor).

Intrusive lock-holder preemption avoidance in our system is achieved by augmenting Linux (our guest OS) with a delayed preemption mechanism. Before acquiring a lock, the guest OS indicates that it should not be preempted for the next n microseconds. After releasing the lock, the guest OS indicates its willingness to be preempted again. The virtualization layer will not preempt the lock-holder if it has indicated that it wishes no preemption, but rather set a flag and delay the preemption by n microseconds. When the guest OS releases the lock, it is required to check the flag, and if set, immediately yield its processing time back to the virtualization layer. Failure to give back the processing time will be caught by the virtualization layer after n microseconds, and the guest operating system will be penalized by a subsequent reduction in its processing time and the possibility of untimely preemption.

The value to choose for n depends on how long the guest OS expects to be holding a lock and will as such rely heavily upon the operating system used and the

	Povray	Kbuild	Apache 2
Locked:			
total	0.04%	15.3%	39.2%
average	3.0μs	1.8μs	2.2μs
max	103μs	293μs	473μs
std.dev.	5.3μs	6.7μs	7.7μs
Unsafe:			
total	0.09%	26.6%	98.9%
average	6.9μs	17.8μs	1.4ms
max	1.4ms	2.0ms	47.6ms
std.dev.	28.7μs	52.4μs	7.5ms

Table 2. Lock-hold and unsafe times

workload being run. We have run a number of lock-intensive workloads on a version 2.4.20 Linux kernel and found that more than 98% of the times the Linux kernel holds one or more locks, the locks are held for less than 20μs. These numbers suggest that setting n any higher than 20μs will not substantially decrease the probability of preempting lock holders in the Linux kernel.

3.2 Non-intrusive Lock-holder Preemption Avoidance

It is not always the case that one has the possibility of modifying the guest operating system, in particular if the kernel is only distributed in binary form. We therefore need non-intrusive means to detect and prevent lock-holders from being preempted. Utilizing the fact that the operating system will release all kernel locks before returning to user-level, the virtualization layer can monitor all switches between user-level and kernel-level,[1] and determine whether it is safe to preempt the virtual machine without preempting lock-holders. This gives us a first definition of *safe* and *unsafe* preemption states:

safe state — Virtual machine is currently executing at user-level. No kernel locks will be held.

unsafe state — Virtual machine is currently executing at kernel-level. Kernel locks *may* be held.

The safe state can be further refined by monitoring for when the guest OS executes the equivalent of the IA-32 HLT instruction to enter a low-latency power saving mode (while in the idle loop). Since the operating system can be assumed to hold no global kernel locks while suspended, it is safe to treat the HLT instruction as a switch to safe state. A switch back into unsafe state will occur next time the virtual CPU is rescheduled (e.g., due to an interrupt).

[1]The mechanism used for monitoring depends heavily on the hardware architecture emulated and the virtualization approach used. One can for instance catch privileged instructions, or insert monitoring code where the hypervisor adds or removes protection for the guest OS kernel memory.

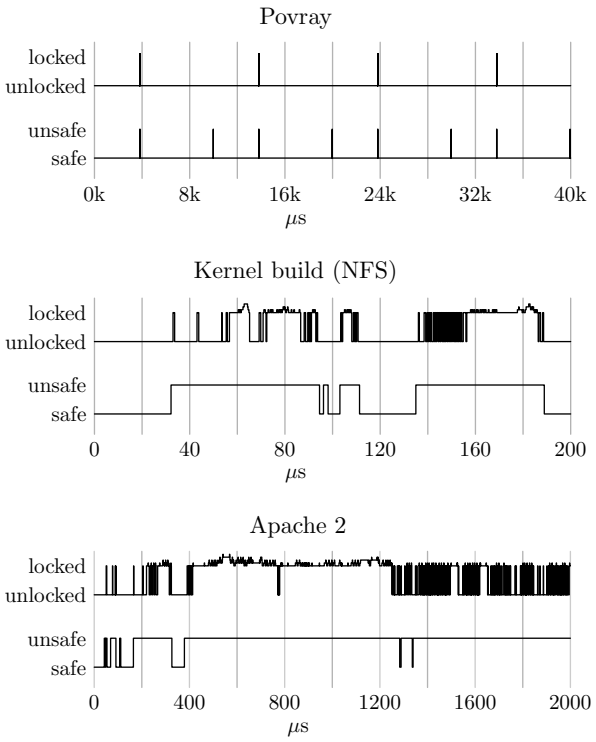


Figure 1. Locked and unsafe times for three different locking scenarios. Povray spends most of the time executing at user-level. Linux kernel build over NFS spends a considerable amount of time at user-level, and moderately stresses the VFS and network layer of the kernel. Apache 2 utilizes the sendfile system call which offloads large amounts of work to the kernel itself.

With the safe/unsafe scheme it is still possible that the virtual machine will be preempted while a *user-level* application is holding a spin-lock. We ignore this fact, however, because user-level applications using spin-locks or spin-based synchronization barriers are generally aware of the hardware they are running on, and must use some form of coscheduling to achieve proper performance. Section 6.4 deals with such workloads in more detail.

In order to substantiate the accuracy of the safe/unsafe model for approximating lock-holding times, we augmented the lock tracing facility described in Section 2 with events for entering and leaving safe states. Our measurements are summarized in Table 2. Figure 1 shows more detailed excerpts of the traces for three different workload classes (spikes on top of the locking states indicate nested locks). The povray work-

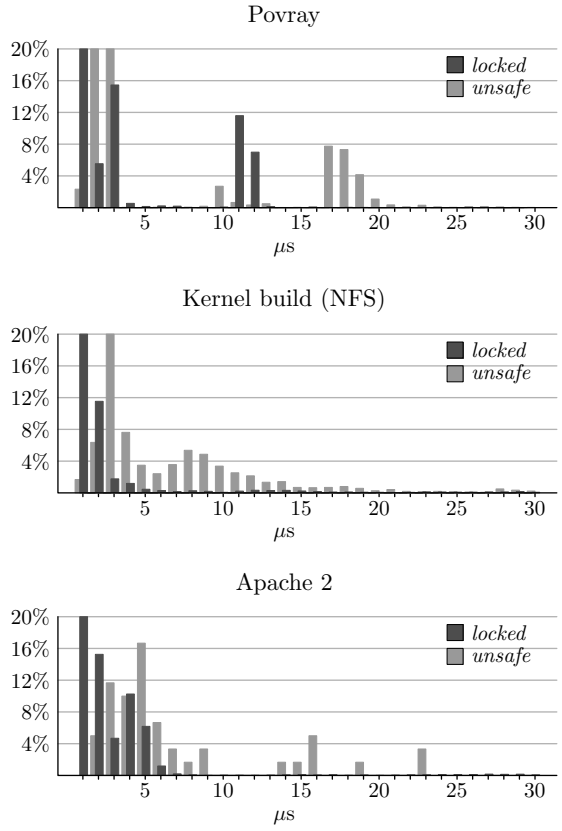


Figure 2. Lock-hold and unsafe state time distributions for three different locking scenarios. Histograms are for readability reasons truncated to a probability of 20%.

load executes almost entirely at user-level and experiences short unsafe periods only at fixed periods. The kernel-build workload performs a parallel build of the Linux kernel on an NFS-mounted file system. It moderately stresses the VFS subsystem and network layer while spending a fair amount of time at user-level. The figure shows a typical 200μs sample period. Lastly, the Apache 2 workload continually serves files to clients and only sporadically enters safe states (only 1.1% of the execution time is spent in a safe state). The large amount of unsafe time can be attributed to Apache's use of the sendfile system call to offload all file-to-network transfer into the kernel (i.e., avoid copying into user-level).

For the Povray and kernel-build workloads we observe that the unsafe times reasonably approximate their lock holding times. Figure 2 shows that the unsafe times for these workloads are generally less than 10μs longer than the lock-holding times. For the Apache 2 workload, however, the unsafe times are on average three orders of magnitude longer than the locking times. We observe an average unsafe time of more than 1.4ms. This is in spite

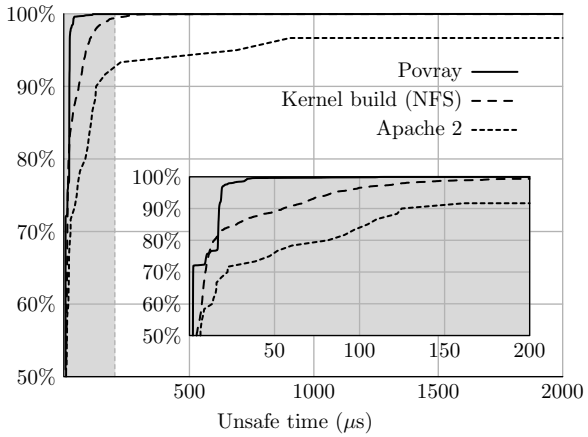


Figure 3. Cumulative probability of unsafe state times for three different workloads. The two graphs show the probabilities of both long and short unsafe state times.

of locking times staying around 2μs, and can, as mentioned above, be attributed to the Apache server offloading file-transfer work to the kernel by using the sendfile system call.

Now, observing that the standard deviation for the unsafe times in the Apache 2 workload is rather high (7.5ms), we might be tempted to attribute the high disparity between average lock-holding times and unsafe times to a number of off-laying measurements. Looking at Figure 3, however, we see that the Apache 2 workload has substantially longer unsafe times even for the lower end of the axis. For example, the Apache 2 workload only has about 91% of the unsafe times below 200μs, while the povray workload has close to 100% of its unsafe times below 20μs. These numbers suggest that the unsafe state approximation to lock-holding times is not good enough for workloads like Apache 2. We want a better approximation.

Having some knowledge of the guest OS internals, it is often possible to construct points in time, so called *safe-points*, when the virtual machine's guest OS is guaranteed to hold no spin-locks.

One example of how safe-point injection can be achieved is through targeted device drivers installed in the guest OS, designed to execute in a lock free context. An example (compatible with Linux) is the use of a network protocol handler, added via a device driver. The virtualization layer could craft packets for the special protocol, hand them to the virtual NIC of Linux, from where they would propagate to the special protocol handler. When the guest OS invokes the protocol handler, it will hold no locks, and so the protocol handler is safe to yield the time slice if a preemption is pending.

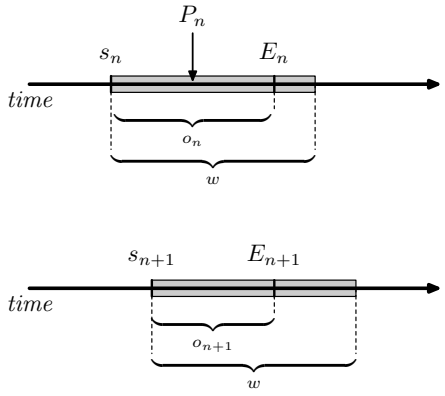


Figure 4. Virtual machine scheduling. A window of size w indicates when a VM may be preempted. E_n indicates the n^{th} end of time slice, P_n indicates the n^{th} actual preemption point, and s_n indicates the start of the n^{th} preemption window.

In an experimental Linux setup we measured as little as 8000 cycles for such a packet to travel from the virtual NIC to our protocol driver under high load. With time slice lengths in the millisecond range, this enables very precise injection of preemption points.

3.3 Locking-Aware Virtual Machine Scheduling

In the following section we describe a mechanism for efficient scheduling of multiple multi-processor virtual machines, leveraging the techniques described in the previous two sections.

Virtual CPUs can be modeled as threads in the virtualization layer which are then subject to scheduling, each one in turn receiving its time slice to execute. Our goal is to preempt a virtual machine (i.e., a thread in the virtualization layer) as close to the end of its time slice as possible, or before that if the virtual machine decides not to consume the whole time slice. In addition, we want to guarantee fairness so that a virtual machine will aggregately obtain its fair share of the CPU time.

Figure 4 illustrates the basis of our virtual machine scheduling algorithm. We define a preemption window, w, around the end of the virtual machine's time slice. For the n^{th} time slice, this preemption window starts at time s_n, E_n is the actual end of the allocated time slice, and P_n is the time when the hypervisor finds it safe to preempt the virtual machine (the last spin-lock was released or the VM entered a safe state). If no safe preemption point occurs before the end of the window, the hypervisor will enforce a preemption.

Our goal with the scheduling algorithm is to choose the window start, s_n, so that the preemption point, P_n, on average coincides with the actual end of time slice, E_n (i.e., $\sum_{i=0}^{n} E_i - P_i = 0$). In doing so we achieve fair access to desired processor time.

Now, assume that in the past the average distance between our start points and preemption points equaled an offset, o_n. In order to keep this property for our next time slice we must calculate the next offset, o_{n+1}, so that it takes into account the current distance between preemption point and window start ($P_n - s_n$). This is a simple calculation to perform, and the result is used to determine the next window start point: $s_{n+1} = E_{n+1} - o_{n+1}$. The consequence of our algorithm is that a preemption that occurs before the end of time slice will cause the preemption window to slide forwards, making premature preemptions less likely (see lower part of Figure 4). Conversely, a preemption after end of time slice will cause the preemption window to slide backwards, making premature preemptions more likely.

The scheduling algorithm ensures that any preemptions before or after the end of time slice will eventually be evened out so that we achieve fairness. However, since the algorithm keeps an infinite history of previous preemptions it will be slow to adapt to changes in the virtual machine workload. To mitigate this problem we can choose to only keep a history of the last k preemptions. The formula for calculating the next window offset then becomes:

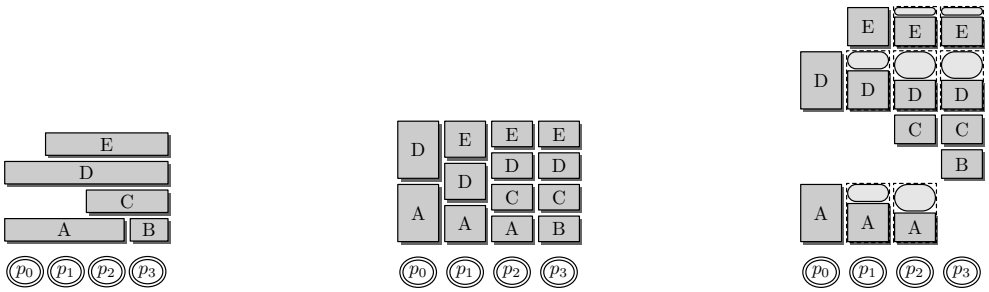
$$o_{n+1} = \frac{o_n(k-1) + (P_n - s_n)}{k}$$

A further improvement of the algorithm is to detect when preemptions have to be forced at the end of the preemption window—a result of no safe state encounters—and remedy the situation by injecting safe points into subsequent preemption windows.

There are two tunable variables in our VM scheduling algorithm. Changing the window length, w, will decrease or increase the accepted variance in time slice lengths, at the expense of having the virtual machines being more or less susceptible to lock-holder preemptions. Changing the history length, k, will dictate how quickly the hypervisor adapts to changes in a virtual machine's workload.

4 Time Ballooning

The scheduling algorithm of a multiprocessor OS distributes load to optimize for some system wide performance metric. The algorithm typically incorporates knowledge about various system parameters, such as processor speed, cache sizes, and cache line migration costs. It furthermore tries to perform a reasonable prediction about future workloads incorporating previous workload patterns.



(a) Five virtual machines running on four physical processors.

(b) Processing time for each processor is apportioned evenly among all virtual machines running on that processor.

(c) Balloon processes are started to even out the differences in processor execution speed as seen by the virtual machines. Rather than consuming CPU cycles, the balloon processes donate their CPU time to other virtual machines.

Figure 5. Time ballooning

By multiplexing multiple virtual processors on one physical CPU the virtualization layer modifies the fundamental system parameters which are used to derive the process distribution, e.g., linear time, identical processor speeds and assumptions about cache working sets due to scheduling order. Based on its wrong assumptions the guest OS's scheduling algorithm will distribute processes among physical processors in a sub-optimal way leading to over-commitment or under-utilization

To perform correct scheduling decisions the guest scheduler has to be made aware of the virtualization layer and incorporate the additional system parameters. Instead of distributing load equally between all processors it should distribute based on the percentage of physical resource allocation.

Using ideas similar to memory ballooning in the VMware ESX Server [25], we propose a new mechanism, *time ballooning*, that enables the hypervisor to partake in the load balancing decisions without requiring modification of the scheduling code itself. A balloon module is loaded into the guest OS as a pseudo-device driver. Periodically, the balloon module polls the virtualization layer to determine how much processing time the virtual machine is allotted on the different CPUs. It then generates virtual workload making the guest OS believe that it is busy executing a process during the time of no physical CPU allocation. The virtual workload levels out imbalances of physical processor allocation (see Figure 5 (c)). The time balloon is used to correct the scheduler's assumption that all processors have the same processing speed, leading to the anticipated distribution.

The method for generating virtual workload depends on the specific load balancing scheme of the guest OS and cannot be generalized. We investigated two Linux scheduling algorithms, the latest, better-scalable O(1) scheduler and the original, sampling-based work stealing algorithm. Since our focus was on non-intrusive solutions, we explicitly avoided modifications of the guest OS' scheduling code that may have resulted in a more efficient solution.

4.1 Linux O(1) Scheduler

Linux's O(1) scheduler uses processor local run queues. Linux bases load balancing decisions on two parameters, the run queue length, using the minimum of two sampling points, and cache working-set estimates based on last execution time. When the balancing algorithm finds a length difference of more than 25% between the current and the busiest run queue, or when the current processor falls idle, it starts migrating processes until the imbalance falls below the 25% threshold.

To reflect differences in virtual machine CPU resource allocations, the balloon module has to manipulate the run queue length. This can be achieved non-intrusively by generating low priority virtual processes with a fixed CPU affinity. In the balloon module we calculate the optimal load distribution using the total number of ready-to-run processes and the allocated CPU share for each virtual CPU. Based on the calculated optimal distribution we add balloon processes to each run queue until all have equal length to the longest run queue, i.e., the virtual CPU with the largest physical processor allocation.

For a guest OS with n virtual CPUs, a total of p running processes, and a CPU specific processor share s_{cpu}, the number of balloon processes b on a particular virtual

processor is

$$b_{cpu} = \left\lceil \frac{\max(s) - s_{cpu}}{\sum_{i=1}^{n} s_i} \cdot p \right\rceil$$

Rounding up to the next full balloon results in at least one balloon process on all but those virtual CPUs with the largest s_{cpu} and thereby avoids aggressive rebalancing towards slower virtual CPUs that fall idle.

4.2 Linux Work-Stealing Scheduler

The second scheduling algorithm, the work-stealing algorithm, migrates processes under two conditions. The first condition responds to newly runnable processes. When a processor P_1 must schedule a newly woken task, it can suggest to idle processor P_2 to acquire the task. The migration completes only when P_2 chooses to execute the task. In the second migration condition, which takes place during general scheduling events such as end-of-timeslice, a processor can choose to steal any process from the centralized task list which is not hard-bound to another processor.

Linux bases load balancing decisions purely on the characteristics of a process, calculated as a "goodness" factor. The decision to steal a process is independent of the status of other processors, and thus doesn't consider factors such as the number of processes associated with other processors.

To influence load distribution, a balloon module has to give the impression that while the virtual processor is preempted it is not idle (i.e., it is executing a virtual process), to avoid migration attempts from other processors. The module could add balloon threads to the task list, bound to a particular processor, and which yield the virtual machine to the hypervisor when scheduled. But the module is unable to guarantee that Linux will schedule the balloon threads at appropriate times. The likelihood of scheduling balloon tasks can be increased by adjusting their priorities.

An alternative to balloon processes is possible. The work-stealing algorithm stores the inventory of running tasks in a central list, and thus if these tasks possess the property of cache affinity, then their task structures are likely to possess a field to represent preferred processor affinity (as is the case for Linux). The balloon module can periodically calculate an ideal load distribution plan, and update the tasks' preferred processor. Thus, as processors perform scheduling decisions, they'll find jobs in the task list biased according to feedback from the hypervisor.

5 Evaluation

5.1 Virtualization Architecture

For our experiments we used a paravirtualization approach running a modified Linux 2.4.21 kernel on top of a microkernel-based hypervisor [13, 23]. With our approach the management and device access parts of the hypervisor run unprivileged in user-space; interaction with the virtual machine manager, including device access, takes place via the microkernel's inter-process communication (IPC) facility.

We modified the Linux kernel to utilize the hypervisor's virtual memory management and scheduling primitives. All device access was wrapped into virtual device drivers that communicate with the real device drivers in the hypervisor.

The hypervisor supports all core device classes: hard disk, Gigabit Ethernet, and text console. Furthermore, it manages memory and time allocation for all virtual machines. To reduce device virtualization overhead we export optimized device interfaces using shared memory communication and IPC.

Currently, the hypervisor partitions memory and processors statically, i.e., no paging of virtual machines is taking place and virtual processors do not migrate between physical CPUs.

5.2 Lock Modeling

Our paravirtualization approach permitted us to reimplement the Linux kernel locks to study the benefits of intrusive lock-holder preemption avoidance. We implemented delayed preemption locks, pessimistic yield locks, and optimistic yield locks (with brief spinning).

The delayed preemption locks were constructed to inhibit preemption whenever at least one lock was held. Each virtual CPU (VCPU) maintained a count of acquired locks. Upon acquiring its first lock, a VCPU enabled its delayed preemption flag to prevent preemption by the hypervisor. The flag was cleared only after all locks were released. Setting and clearing the flag was a low cost operation, and only involved manipulating a bit in a page shared between the hypervisor and the VCPU. If the hypervisor signaled that it actually delayed a preemption, via another bit in the shared page, then the Linux kernel would voluntarily release its time slice immediately after releasing its last lock.

The yield locks were pessimistic and assumed that any spinning was due to a preempted lock, thereby immediately yielding the time slice if unable to acquire the lock. We also used an optimistic yield lock, which first spun on the lock for 20μs (as suggested from lock holding times in Figure 2), and then yielded the time slice with the assumption that the lock was preempted.

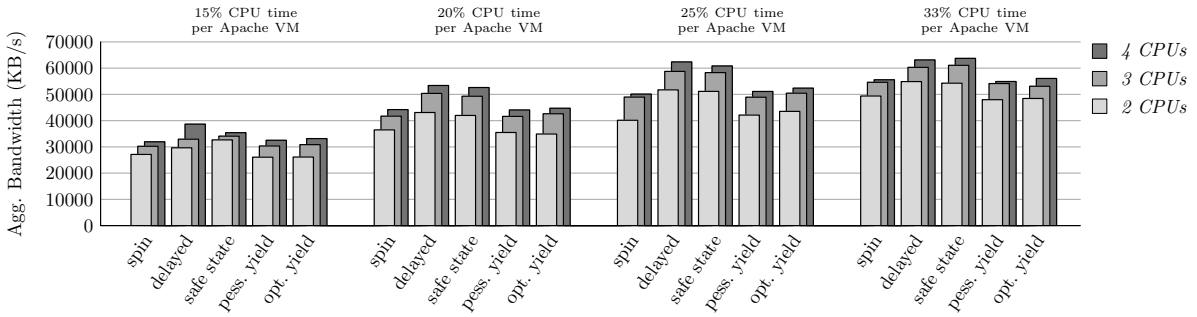


Figure 6. Bandwidth measurement for Apache 2 benchmark. Two virtual machines are configured to run on 2, 3 and 4 CPUs. For each CPU configuration the virtual machines are further configured to consume 15%, 20%, 25% or 33% of the total CPU time. A third virtual machine with a CPU intensive workload consumes the remaining CPU time. The bars show the aggregate bandwidth of the two VMs running the web servers.

To model the performance of lock-holder preemption avoidance with a fully virtualized virtual machine, and thus an unmodified guest OS, we had to create estimates using our paravirtualization approach. Given that paravirtualization can outperform a fully virtualized VM [3], our results only estimate the real benefits of lock preemption avoidance in a fully virtualized VM. Our safe-state implementation precisely models the behavior we describe in Section 3.2. We model safe-state detection by using the hypervisor's delayed preemption flag. When the guest OS executes kernel code we enable the delayed preemption flag, thus compelling the hypervisor to avoid preemption at the end of normal time slice. After finishing kernel activity (upon resuming user code or entering the idle loop), we clear the preemption flag. If Linux kernel code exceeds a one millisecond preemption grace period, it is preempted.

We observed spin-lock performance by using standard Linux spin-locks. The spin-lock data apply to the case of using spin-locks for paravirtualization, and they approximate the case of a fully virtualized VM.

5.3 Execution Environment

Experiments were performed with a Dell PowerEdge 6400, configured with four Pentium III 700 MHz Xeon processors, and an Intel Gigabit Ethernet card (using an 82540 controller). Memory was statically allocated, with 256 MB for the hypervisor, and 256 MB per virtual machine.

The guest operating system was a minimal Debian 3.0, based on a modified Linux 2.4.21 kernel. Most Debian services were disabled.

The hard disk was unused. Instead, all Linux instances utilized a 64 MB RAM disk. Linux 2.4.21 intelligently integrates the RAM disk with the buffer cache which makes this setup comparable to a hot buffer cache scenario.

5.4 Synthesized Web Benchmark

The synthesized web benchmark was crafted to tax the Linux network and VFS subsystems, in effect, to emulate a web server under stress. It used Apache 2, for its support of the Linux sendfile system call. The sendfile method not only offloads application processing to the kernel, but it also supports network device hardware acceleration for outbound checksum calculations and packet linearization.

The benchmark utilized two virtual machines, each hosting a web server to serve static files. The two virtual machines competed for the network device, and each had an equal amount of processor time. A third virtual machine provided an adjustable processor load, to absorb processor time.

5.5 Lock-Holder Preemption Avoidance Data

For studying the effect of our lock-holder preemption schemes, we instrumented Linux's synchronization primitives, and measured the throughput of the synthetic web benchmark described in the previous section.

To capture lock scaling in terms of number of processors, the Linux instances were configured to activate two, three, or four processors of our test system. Further, for each processor configuration we configured the virtual machines hosting the web servers to consume 15%, 20%, 25% or 33% of the total CPU time. Figure 6 summarizes our results, showing the aggregate bandwidth of both web servers for the different virtual machine and locking scheme configurations.

The results often reveal a substantial performance difference between the various locking techniques. In order

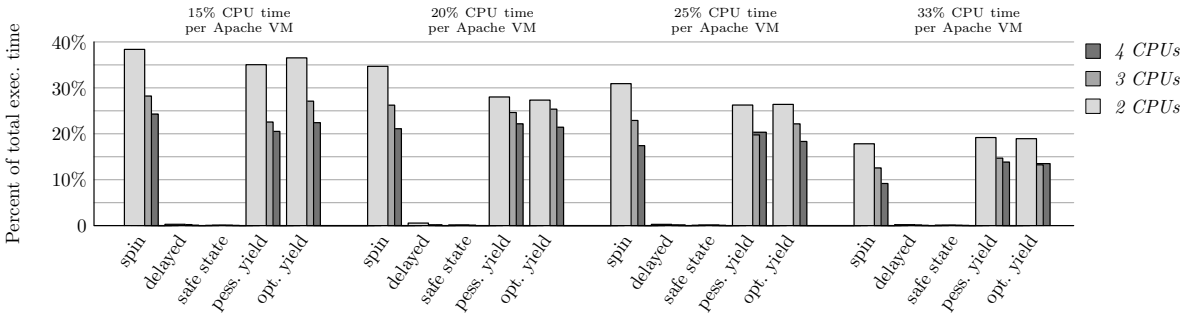


Figure 7. Extended lock-holding times for various virtual machine configurations (same configurations as in Figure 6). The bars show the extended lock-holding time for one of the web server VMs, per processor, expressed as a percentage of the run time. An extended lock-hold is one which exceeds 1 ms.

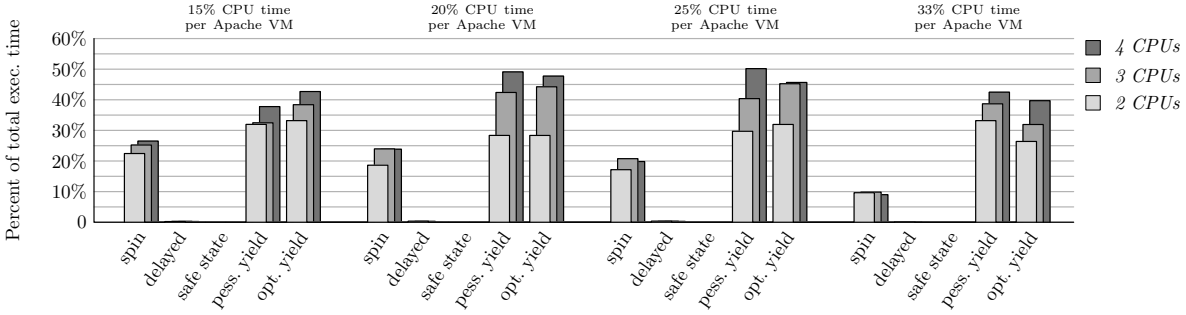


Figure 8. Extended lock-wait times for various virtual machine configurations (same configurations as in Figure 6). The bars show the extended time spent waiting for locks on one of the web server VMs, per processor, expressed as a percentage of the run time. An extended lock-wait is one which exceeds 1 ms.

to explain these differences, we measured lock holding times in each experiment. More precisely, we measured the amount of wall-clock time that a lock was held by the virtual machine running one of the web servers. Figure 7 shows the extended lock-holding time as a percentage of the virtual machine's real execution time during the benchmark. To distinguish between normal locking activity and lock-preemption activity, we show only extended lock-holding time. A lock-hold time is considered extended if it exceeds one millisecond.

We also measured the total time spent acquiring locks in each of the experiments (i.e., the lock spin-time or wait-time). These lock-acquire times are presented in Figure 8. Again, the time is measured relative to the real execution time of the benchmark. The data include only wait-times which exceeded one millisecond, to distinguish between normal behavior and lock-wait activity due to preempted locks.

5.6 Time Ballooning

Figure 9 presents the result of an experiment designed to show the effectiveness of the O(1) time ballooning algorithm with asymmetric, static processor allocation. We ran two virtual machines, A and B, each configured to run on two physical processors. The goal was to fairly distribute physical processor time between all processes. Both VMs were running a Linux 2.4.21 kernel with the O(1) scheduler patch and our balloon module. Virtual machine A was configured with 30% of the processing time and ran ten CPU intensive tasks. Virtual machine B was configured with the remaining 70% of the processing time, but only ran one CPU intensive task for a short period of time; the other CPU was idle.

Before VM B started using its processing time, VM A used all processing time on both CPUs. Once VM B started processing (at about 7 seconds into the experiment), virtual machine A's processing share on CPU 0 immediately dropped to 30%. Shortly thereafter, VM A detected the imbalance in processing time, and attempted to mitigate the problem by inserting balloon

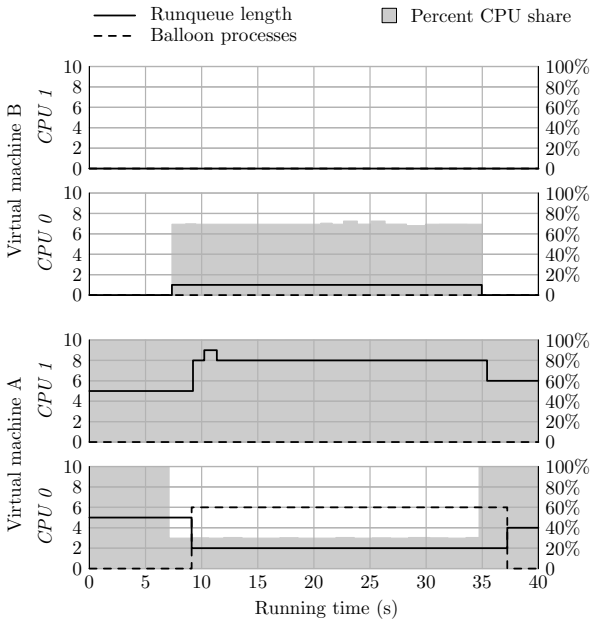


Figure 9. Time ballooning induced load balancing.

processes on the slower CPU. This in turn caused Linux to start migrating processes to the other CPU.

When virtual machine B ceased using its processing time (at about 35 seconds into the experiment), virtual machine A again got the full CPU share, causing the balloon processes to be removed, and Linux to perform another load-balancing.

6 Discussion and Future Work

6.1 Lock-holder Preemption Behavior

The web server benchmark provided complicated system dynamics. The processor utilization at almost every data point fell below the allocated processor share (we used a 5 ms time slice), suggesting that the workload was I/O-bound. On the other hand, the bandwidth is (non-linearly) proportional to the amount of processor share, suggesting a processor-bound workload. The sequencing of virtual machine scheduling, compared to packet arrival and transmit periods, probably causes this behavior. When a guest OS has access to the physical processor it won't necessarily be saturated with network events, since the network card may be sending and receiving packets for a competing virtual machine. Other factors can also regulate web server performance. For example, when a virtual machine sees little concurrent access to both the network device and the processor, the role of kernel buffers and queues becomes important. The buffers and queues must hold data until a resource is available, but buffer overruns lead to work underflow, and thus under-utilization of the time slice. The sequencing of events may also lead to high packet processing latency, causing TCP/IP packet retransmits, which increases the load on the limited resources.

During the experiments involving our lock-holder preemption avoidance techniques we set the size of the preemption window (the value of w in Section 3.3) to 1 ms. Since more than 96% of the unsafe times for an Apache 2 workload fall below the 1 ms boundary (see Figure 3), the non-intrusive technique did not suffer from excessive amounts of lock-holder preemptions. As such, the web-server throughput achieved with the non-intrusive technique was on par with the intrusive technique.

The hypervisor uses a proportional share stride-scheduling algorithm [26, 27], which ensures that every virtual machine receives its allocated share of the processor. However, the scheduler doesn't impose a ceiling on a virtual machine's processor utilization. If one virtual machine under-utilizes its processor share for a given unit of time, then another virtual machine can pilfer the unused resource, thus increasing its share. With I/O bound workloads, we witness processor stealing (because we have time slice under-utilization). In essence, the scheduling algorithm permits virtual machines to rearrange their time slices. The concept of time-stealing also applies to yield-style locks. When a virtual processor yields its time slice, rather than wait on a lock-acquire, it may avoid many milliseconds of spinning. Yielding the time slice, though, enables the hypervisor to schedule another virtual machine sooner, and thus return to scheduling the yielding virtual machine earlier, at which point the lock should be released from the original lock holder. Time slice yielding, as an alternative to spinning on preempted locks, will improve the overall efficiency of the system, as it avoids processor waste. And via the rearrangement of time slices, yield-style locks may improve their processor/packet sequencing, and improve performance.

Increasing parallelism by adding more processors increases the likelihood of lock contention. Thus preemption of a lock holder can lead to spinning by more processors. As Figure 8 shows, the lock-wait time increases with the number of processors.

In the graphs which represent lock-holding time (Figure 7) and lock-waiting time (Figure 8), the mechanisms which avoid lock-holder preemption (delayed-preemption locks and safe-state detection) have under 1% time involved with locking activity. In contradistinction, the spin-locks lead to severe locking times and spinning times due to preempted locks. The yield-style locks are susceptible to lock preemption, as they only

focus on recovering from lock spinning, and also suffer from long lock holding times. And due to releasing their time slices for use by other virtual machines, yield-style locks suffer from long lock-wait times as well. The tactic of avoiding spin-time by yielding doesn't help the benchmark results, and often leads to performance worse than spin-locks. Spin-locks may spin for a long time, but they have a chance of acquiring the lock before the end of time slice, and thus to continue using the time slice.

6.2 Time Ballooning

Our time ballooning mechanism is targeted towards processor allocations with infrequent changes. Highly dynamic processor reconfigurations introduce load-dependent properties, and are a topic of future work. To support dynamic load balancing in response to adjustments of the processor allocations requires attention to several variables, such as the rates at which the guest OS can rebalance and migrate tasks, allocation sampling rates, sampling overhead, cache migration costs, sample history window size, allocation prediction, and attenuation of thrashing. Likewise, response to burst loads is another dynamic situation for future work. For example, to achieve optimal use of the available physical processor resources, web servers distributed across asymmetric processors may require load balancing of incoming wake-up requests, or load balancing of the web server processes after they wake/spawn.

The time ballooning mechanism is designed to be installed in unmodified guest operating systems via device drivers. Where it is possible to modify the guest operating system, one can construct feedback scheduling algorithms optimized for the OS and workload and virtualization environment.

6.3 Dealing with Lock-Holder Preemption

An alternative to avoiding preemption of lock holders in the first place is to deal with the effects of the preemption: Have the hypervisor detect extensive spinning and schedule other, more useful work instead.

Techniques to detect spinning include instruction pointer sampling and trapping on instructions that are used in the back-off code of spin-locks. Descheduling a virtual CPU immediately after a failed acquire operation is expected to show the same behavior as pessimistic yield locks. To reduce the yield time, one could look for a lock release operation following a failed lock acquire. Lock release operations can be detected by write-protecting pages containing a lock, using debug registers to observe write accesses, or, with additional hardware support, through extensions of the cache snooping mechanism.

However, preempting a virtual CPU due to a remote release operation on a monitored lock may preempt another lock holder. Most likely, release operations on a contended lock occur with much higher frequency than preemptions due to regular VM scheduling. Instead of offering a solution, chances are that this approach could amplify the problem. The potential costs and complexity of detecting spinning on a lock and executing another virtual CPU instead could outweigh the benefits.

6.4 Coscheduling Dependent Workloads

In this paper we have discussed application workloads that do not possess strong cross-processor scheduling requirements. However, some workloads in the parallel application domain do rely on spin-based synchronization barriers or application spin-locks, and thus necessitate some form of coscheduling in order to perform efficiently. Coscheduling can only be achieved on physical resources (processors and time), and the coscheduling requirements in the guest OS must therefore be communicated to the virtualization layer so that they can be processed on a physical processor basis.

Making the virtualization layer aware of an application's coscheduling requirements and devising algorithms for fulfilling these requirements is future work.

7 Related Work

The problems that may result from preempting parts of a parallel application while holding a lock are well-known [16, 20, 30] and have been addressed by several researchers [1, 4, 8, 17, 29]. Proposed solutions include variants of scheduler-aware synchronization mechanisms and require kernel extensions to share scheduling information between the kernel's scheduler and the applications. This translates well to our para-virtualization approach where the guest OS kernel provides information to the hypervisor. To our knowledge no prior work has applied this technique in the context of virtual machines and their guest OSs.

Only very few virtual machine environments offer multiprocessor virtual machines, and they either avoid lock-holder preemption completely through strict use of gang scheduling [5, 10], use gang scheduling whenever they see fit [24], or they choose flexible scheduling but don't address lock-holder preemption [15].

Load balancing across non-uniformly clocked, but otherwise identical CPUs is a standard technique in cluster systems. However, we know of no commodity operating system for tightly coupled multiprocessors that would explicitly support such a configuration. The few existing virtual multiprocessor VM environments either

implicitly enforce equal virtual CPU speeds (through gang scheduling) or do not consider speed imbalances. As such, we know of no prior art where a virtual machine environment would coerce its guest operating system to adapt to changing, or at least statically different, virtual CPU speeds. Our solution was inspired by the ballooning technique for reclaiming memory from a guest OS [25].

Performance isolation is a quite well-researched principle [2, 3, 12, 18] in the server consolidation field. For CPU time our hypervisor enforces resource isolation using a proportional share stride-scheduling algorithm [26, 27].

8 Conclusion

Virtual machine based server consolidation on top of multiprocessor systems promises great flexibility with respect to application workloads, while providing strong performance isolation guarantees among the consolidated servers. However, coupling virtual machine environments with multiprocessor systems raises a number of problems that we have addressed in this paper.

First, our schemes for avoiding preemption of lock-holders in the guest operating systems prevents excessive spinning times on kernel locks, resulting in noticeable performance improvements for workloads exhibiting high locking activity. Our lock-holder preemption avoidance techniques are applicable to both paravirtualized and fully virtualized systems.

Second, the allocation of physical processing time to virtual machines can result in virtual machines experiencing asymmetric and varying CPU speeds. Our time ballooning technique solves the problem by creating artificial load on slower CPUs, causing the guest OS to migrate processes to the CPUs with more processing time.

Combined, our solutions enable scalable multiprocessor performance with flexible virtual processor scheduling.

Acknowledgements

We would like to thank Åge Kvalnes for his insightful feedback. Further thanks to the anonymous reviewers for their valuable comments. The work presented in this paper was in part funded through a grant by Intel Corporation.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 22–25 1999.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 19–22 2003.

[4] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, 1990.

[5] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proc. of the 16th Symposium on Operating Systems Principles*, Saint Malo, France, October 5–8 1997.

[6] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proc. of the 1st Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 14–17 1994.

[7] Connectix. *Virtual PC for Windows, Version 5.1, User Guide*, second edition, July 2002.

[8] Jan Edler, Jim Lipkis, and Edith Schonberg. Process management for highly parallel UNIX systems. In *Proc. of the USENIX Workshop on Unix and Supercomputers*, Pittsburg, PA, September 26–27 1988.

[9] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[10] Kingshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262, 2000.

[11] Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 28–31 1996.

[12] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proc. of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, September 23–26 2003.

[13] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based systems. In *Proc. of the 16th ACM Symposium on Operating System Principles*, Saint Malo, France, October 5–8 1997.

[14] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for Real-Time systems. In *Proc. of the 2001 USENIX Annual Technical Conference*, June 25–30 2001.

[15] IBM. *z/VM Performance, Version 4 Release 4.0*, fourth edition, August 2003.

[16] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, October 13–16 1991.

[17] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, 1997.

[18] Åge Kvalnes, Dag Johansen, Robbert van Renesse, and Audun Arnesen. Vortex: an event-driven multiprocessor operating system supporting performance isolation. Technical Report 2003-45, University of Tromsø, 2003.

[19] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, New York, NY, 1991.

[20] Maged M. Michael and Michael L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proc. of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1–5 1997.

[21] John. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. of the 3rd International Conference on Distributed Computing Systems*, Ft. Lauderdale, FL, October 18–22 1982.

[22] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proc. of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 25–30 2001.

[23] System Architecture Group. The L4Ka::Pistachio microkernel. White paper, Universität Karlsruhe, May 1 2003.

[24] VMware Inc. VMware ESX Server online documentation. http://vmware.com/, 2003.

[25] Carl A. Waldsburger. Memory resource management in VMware ESX Server. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 9–11 2002.

[26] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 14–17 1994.

[27] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, MIT Laboratory for Computer Science, 1995.

[28] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 9–11 2002.

[29] Robert W. Wisniewski, Leonidas I. Kontothanassis, and Michael L. Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 19–21 1995.

[30] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.