Hardware Support for Spin Management in Overcommitted Virtual Machines

Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin, Madison
{pwells, kchak, sohi}@cs.wisc.edu

ABSTRACT

Multiprocessor operating systems (OSs) pose several unique and conflicting challenges to System Virtual Machines (System VMs). For example, most existing system VMs resort to gang scheduling a guest OS's virtual processors (VCPUs) to avoid OS synchronization overhead. However, gang scheduling is infeasible for some application domains, and is inflexible in other domains.

In an overcommitted environment, an individual guest OS has more VCPUs than available physical processors (PCPUs), precluding the use of gang scheduling. In such an environment, we demonstrate a more than two-fold increase in runtime when transparently virtualizing a chip-multiprocessor's cores. To combat this problem, we propose a hardware technique to detect several cases when a VCPU is not performing useful work, and suggest preempting that VCPU to run a different, more productive VCPU. Our technique can dramatically reduce cycles wasted on OS synchronization, without requiring any semantic information from the software.

We then present a case study, typical of server consolidation, to demonstrate the potential of more flexible scheduling policies enabled by our technique. We propose one such policy that logically partitions the CMP cores between guest VMs. This policy increases throughput by 10–25% for consolidated server workloads due to improved cache locality and core utilization, and substantially improves performance isolation in private caches.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Performance attributes, C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)

General Terms: Performance, Design

Keywords: Virtual machines, synchronization overhead, chip multiprocessors

1. INTRODUCTION

Virtualization — the art of separating resources from the interface used to access those resources — has been an important technique for many decades. Specifically, a *System Virtual Machine* (System VM) allows an operating system (OS) to run on a different host architecture than the one for which it was originally developed, including a host running multiple guest OSs. Smith and Nair provide a provide a good overview of many VM topics [20].

A *Virtual Machine Monitor* (VMM) is a layer which lies between the guest OS(s) and the hardware on which it runs, implementing the system VM by translating the interface exposed by hardware into the interface expected by the guest OS. Successful system virtualization requires a VMM to support virtual processors, memory, and peripheral devices for correctness, as well as manage resource allocation and maintain performance isolation.

Much prior system VM research has focused on the complexities of virtualizing memory, processors, and peripheral devices [2, 5, 7, 12, 25, 26]. However, there is little published work that has tackled the unique challenges of supporting an unmodified, modern, multiprocessor guest OS. With chip multiprocessor (CMP) systems quickly becoming ubiquitous even for desktops, system VMs must provide scalable support for multiprocessor OSs.

This work focuses on transparently virtualizing the processors (i.e., processing cores) of a CMP for multiprocessor OSs. There are several reasons why virtualizing these cores may be appealing, such as server consolidation, managing heterogeneous cores, thermal management, and fault tolerance. Traditionally, gang scheduling (or *co-scheduling* [17]) has been used with multiprocessor virtual machines to create a synchronization environment similar to a non-virtualized system [12, 28]. However, in many of the applications mentioned above, a single guest VM is *overcommitted*, i.e., the number of VCPUs exceeds the number of active physical CPUs or PCPUs, and gang scheduling is not feasible. In other applications, gang scheduling may not allow policies which optimize for throughput or performance isolation. Therefore, it is critical to allow the system VM to adopt more flexible scheduling policies.

In the absence of gang scheduling, several challenges arise due to certain assumptions made by the OS to accomplish the intricate task of synchronization between different CPUs. To ensure correct execution in such an environment, the VMM must not allow these (possibly now invalid) assumptions to affect correctness or cause significant performance loss. For example, a particular VCPU may send a software interrupt to another VCPU and simply busy wait for the response. If the receiving VCPU is not currently running on a PCPU (i.e., it is *paused*), the sender will spin without doing any useful work, possibly long enough for the operation to time out, panicking the kernel.

Uhlig, et al., also address managing kernel synchronization transparently to the software without using gang scheduling [26]. However, one key aspect that limits the scope of their solution is that individual guest VMs are never overcommitted. Therefore, it is always possible to execute all of an OS's VCPUs simultaneously, if necessary.

This paper makes three main contributions:

- First, we investigate the synchronization overhead of virtualizing a multiprocessor OS on a typical CMP in the absence of gang scheduling. We find that commercial workloads running on Solaris have significant synchronization overhead resulting from mutex locks and software-generated interrupts (CPU cross-calls), which can result in a 2.5 times increase in runtime. (Section 2)
- Second, we propose a novel hardware mechanism to manage the synchronization overhead without any software modification. Based on a simple, yet effective, heuristic, we detect when a VCPU is excessively spinning and preempt that VCPU in order to run a more productive VCPU. Compared to other methods, our solution does not require any semantic information from the software, leads to a more unified solution to detect other cases when a VCPU is not doing any useful work (e.g., kernel idle loop), and can be easily implemented in the hardware. We demonstrate that this technique dramatically reduces the synchronization overhead, and also out-performs another recently proposed VMM technique [26] to manage OS synchronization. (Sections 3 and 5)
- Third, we examine a case study, server consolidation, and propose one example of a flexible scheduling policy — enabled by our technique to manage synchronization overhead — where we concurrently execute only a subset of each guest VM's VCPUs. This allows us to logically partition the PCPUs among the guest VMs, significantly improving both cache performance and PCPU utilization compared to gang scheduling, leading to overall performance improvements of 10–25%. We also show that this policy can dramatically improve performance isolation. (Section 6)

2. CMP VIRTUALIZATION

Virtualizing processing cores in a CMP simply means that the processors seen by the OS (virtual processors or VCPUs) can differ from the physical processing cores (PCPUs) that are actually executing code. The number of VCPUs can be more, less, or the same as the number of PCPUs.

System virtualization can be done in a manner entirely transparent to the guest OS (i.e., *pure virtualization*, as in VMWare [28]), or in cooperation with the guest OS, (i.e., *para virtualization*, as in Xen [7], Denali [30], and the IBM Power5 Hypervisor [5]). Both techniques have several advantages and disadvantages. Primarily, pure virtualization can support unmodified OSs, but can suffer performance loss when the OS makes assumptions about the hardware that are invalid for virtualized resources [7]. Para virtualization eliminates these assumptions, but sometimes requires significant modification to the guest OSs and the hardware/software interface.

In addition, there are many cases where para-virtualization is inapplicable or unnecessary, such as when supporting legacy software. It is hard to predict which particular approach — pure or para — if not both, will survive, given the diverse application domain for virtualization. In this paper, we assume the use of pure virtualization.

2.1 Motivation

Many intriguing proposals which suggested changes to the hardware/software interface have failed to materialize, in part due to the necessity of maintaining backward compatibility. However, system VMs (particularly co-designed system VMs [21]) have the potential to eliminate the interface changes otherwise required for some proposals. We believe that there are several applications that can leverage pure virtualization of CMP cores, though the original proposals often assumed software support:

- **Hardware Scheduling on CMP Cores**: Virtualizing CMP cores allows hardware to schedule computation transparently to the software layers above. This technique can be applied to a chip with heterogeneous cores [14] or to exploit the synergy among different threads in homogeneous workloads as demonstrated by Computation Spreading [8] or Cohort Scheduling [15].
- **Dynamic Thermal Management:** Virtualization can be used to perform dynamic thermal management similar to *Heat-and-Run* [18] transparently to the OS. This may allow hardware designers to implement the best thermal management policy for a particular chip, without relying on OS vendors to provide optimized support.
- **Speculative/Redundant Multithreading:** A VMM may be aware of special requirements for a particular program or guest OS. It may then dynamically choose to execute redundant helper threads for reliability [11], or use multiple cores for speculative parallelization of single-threaded programs, such as the Multiscalar system [22].
- **Fault Tolerance**: Virtualized CMP cores can allow the chip to function, albeit at a lower performance, even if multiple cores sustain permanent faults. This can be done transparently to the OS if desired (at least while waiting for the OS to cleanly shut down the VCPU mapped to the faulty PCPU), and can alleviate the need to add spare cores precisely for this purpose.
- **Server Consolidation:** Consolidating many separate servers onto a single piece of hardware can provide a significant reduction in IT costs [5, 23, 28]. Current technologies limit the scheduling flexibility.

For many of these applications, virtualization will result in overcommitment of a single OS's VCPUs. That is, there will be prolonged periods when a given guest OS has more VCPUs than available PCPUs. This paper addresses the challenges of such a scenario.

2.2 Synchronization Overhead

Many of the challenges that arise when virtualizing an OS are similar whether running in a multiprocessor or uniprocessor virtual machine, including proper handling of memory and I/O devices. We focus on the main difference between uniprocessor and multiprocessor VMs: synchronization overheads.

Several prior proposals adopt gang scheduling (or *co-scheduling* [17]) for multiprocessor virtual machines. This technique simply ensures that either all VCPUs of a particular guest OS are executing concurrently or none are. Thus, gang scheduling attempts to create an execution environment similar to a non-virtualized system. Gang scheduling is used by VMWare's ESX server [27] and Cellular Disco [12], among others. OS synchronization overheads in this case are similar to a non-virtualized machine. However, gang scheduling is not always possible or appropriate, particularly in an over-committed environment (either temporally or permanently).

In the absence of gang scheduling, synchronization overhead becomes a significant impediment to the performance of a multiprocessor virtual machine. Below we discuss the two major components of this overhead: mutex locks and CPU cross-calls (software-generated interrupts).

2.2.1 Mutex Locks

A multiprocessor OS which is unaware of any virtualization underneath assumes all its VCPUs are executing all the time. However, this assumption does not hold for many of the applications in Section 2.1. This leads to synchronization problems which can result in deadlock, kernel panic, and/or severe performance loss. The problems with this false assumption manifests itself through the use of mutex spin locks to protect critical sections in the OS kernel.[1]

For example, a Solaris kernel thread attempting to acquire an adaptive mutex lock will sometime block (i.e., preempt) the thread when the lock is already held, but for performance reasons, will spin when the owner of the lock is running on another VCPU (assuming the lock will not be held long). If this lock-holding VCPU is not currently executing on a PCPU, however, the thread will spin unnecessarily. Other, non-adaptive locks are not preemptible at all. Linux 2.6.10, which we also use for this study, does not use adaptive locks, and always spins, though it sometimes uses semaphores for preemptible synchronization.[2] As we will see in Section 2.2.3, this unnecessary spinning can easily increase runtime by a factor of two or more.

2.2.2 Cross-Calls

A multiprocessor OS often employs cross-calls (using synchronous software interrupts) to communicate between different CPUs. An example of this is TLB shootdowns [19] where one CPU wishes to change a virtual-to-physical page mapping (e.g., to invalidate an entry or update the protection bits) that may be cached in other CPUs' TLBs. Another example, prevalent in the Solaris kernel, is the use of cross-calls to preempt one or more remote CPU(s) to run a higher priority thread. Linux does not use cross-calls for this type of scheduling action. When a cross-call is invoked, the software busy waits until the hardware notifies it that the interrupt has actually been delivered to the remote CPU.

For SPARC, the remote CPU sends a NACK when it already has an outstanding software interrupt, in which case, the sender must continue to retry until the request is ACKed. Other architectures, such as x86, also require software to ensure that these interrupts are not lost.

[1]Spin locks can affect user code as well, but we focus on non-scientific applications, which typically do not make the same assumptions as the OS and do not spin.

[2]The Linux 2.6.16 kernel added support for adaptive mutex locks using a similar policy as Solaris, but not in time for us to rebuild our workloads for this study.

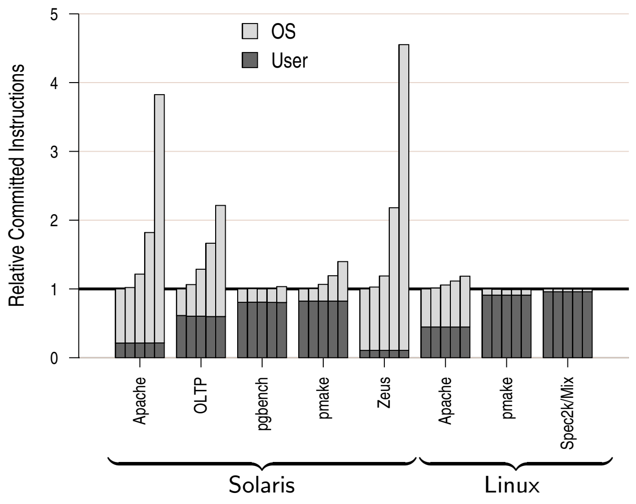

Figure 1: Relative Instruction Count. Five bars for each benchmark (from left to right) represent results for 10μs, 20μs, 50μs, 100μs and 200μs timeslices. Results are normalized to the 10μs timeslice (lower is better).

Fast execution of cross-calls becomes challenging in an overcommitted virtual machine where all the VCPUs of a VM are not running concurrently. Since OSs typically do not use nested locking, the forward progress of the lock-holder is always guaranteed in a simple mutex lock. But in a cross-call, the forward progress of the initiating VCPU, which is often holding a lock (such as page table lock or run-queue lock), is dependent upon the recipient VCPUs. For example, when multiple VCPUs send interrupts to a paused VCPU before they can be handled, one or more of the senders will continue to spin until its interrupt can be delivered. In other words, when any responding VCPU is paused with a pending software interrupt, it will thwart the forward progress of the interrupt initiator, which in turn will cause a cascading affect on other VCPUs attempting to acquire the lock held by the initiator.

Modifying the interface between the OS and hardware to allow (and guarantee delivery of) an unlimited number of outstanding interrupts might alleviate this problem. However, for the purpose of this paper, we do not investigate such interface modifications.

2.2.3 Experimental Results

To expose the challenges of virtualizing a multiprocessor OS in an overcommitted environment, we consider the case of a single guest VM with 24 VCPUs running on an eight core (8 PCPU) CMP system. In other words, there are hundreds of software threads that the OS schedules among its 24 VCPUs, the VMM then selects 8 VCPUs to execute on the PCPUs at any given time. To maintain cache and TLB affinity, we assume that three VCPUs are statically mapped to a given PCPU, though only one is executing at a time (maintaining load balance in a real system would mean that this mapping could change dynamically). To maintain forward progress for all VCPUs, each PCPU preempts the running

VCPU after a given *timeslice* so that the other VCPUs can run; preemption occurs independently for each PCPU. The workload setup and target CMP will be described in more detail in Section 4.

Figure 1 shows the breakdown of instructions executed by the user code and the OS, respectively, for various VCPU timeslices, while performing the same amount of work (i.e., the same number of workload *transactions*). While the aggregate user instructions executed remain very stable across different timeslices, we notice a dramatic increase in OS instructions for larger timeslices. For example, OLTP executes 2.2 times more instructions with a timeslice of 200μs compared to 10μs, almost entirely due to OS instructions. Longer timeslices, such as 1ms, can cause a kernel panic in several Solaris workloads. For Apache on Solaris, the runtime (not shown) of the 200μs timeslice is more then 2.5 times that of the 10μs case. We also see that the workloads running on Linux show significantly less overhead due to additional OS instructions compared to their Solaris counterparts, largely due to the absence of frequent cross-calls. However, Apache on Linux also shows excessive spinning when using longer timeslices (data not shown).

While a timeslice as low as 10μs can mitigate the OS spin problem, it can also destroy cache locality (in addition to incurring other overheads). For example, OLTP, despite a more than two-fold increase in instructions due to spinning, has 20% *fewer* L1 misses with a timeslice of 200μs than 10μs (data not shown for brevity). Pmake on Linux, which has no synchronization overhead, sees a 35% reduction in L1 misses. Clearly we want a better solution which will allow us to give VCPUs a longer timeslice when they are doing useful work. Traditional gang scheduling *can* allow longer timeslices, but we are seeking solutions with more flexibility and feasibility across a range of VM applications.

3. MANAGING OS SYNCHRONIZATION

3.1 Current Solutions

The problems associated with preempting a lock holder are well understood in the context of user programs [4, 31, 32]. In any parallel application (a multiprocessor OS being a complex and interesting example), there are two generic approaches to mitigate the synchronization problem among threads which are not concurrently executing: a) avoid preempting a thread holding a lock or b) pro-actively preempt a thread excessively spinning to acquire a lock in favor of executing a more productive thread.

In an optimized parallel application, the length of critical sections is typically short. Thus, avoiding preemption of the lock holder can often yield good performance for such applications. However, this approach requires precise information about the lock holder. While this information may be available from a user application, it is more subtle and largely impossible to ascertain from an unmodified multiprocessor OS running in a virtual machine.

However, one can use more conservative information to derive a software-transparent solution as shown by Uhlig, et al., [26]. They make the observation that a VCPU executing in user mode is not holding a kernel lock, and can be safely preempted. These preemptible locations are referred to as *safe points*. A VCPU executing in the OS may be holding a kernel lock, and thus, the VMM will try *not* to preempt it at that point.[3] However, this scheme fails to efficiently handle the frequent cross-calls seen in the applications running Solaris. We discuss the reasons in Section 5.

The second approach to reduce the synchronization overhead is to detect when a VCPU is excessively spinning on a lock and preempt the VCPU at that point. This is similar in concept to *helping locks* proposed by Hohmuth, et al. [13]. Most current solutions for identifying spin locks and loops (including the OS idle loop) involve OS-intrusive modifications or kernel PC annotations. Several existing projects use this method (often just for the idle loop), including IBM's Power5 Hypervisor[5], Cellular Disco [12], and one of the solutions offered by [26]. Instead, we propose a simple yet effective heuristic to dynamically determine spin loops in hardware without requiring any software modification.

3.2 Detecting Spins in Hardware

We make the observation that a program executing in a spin loop has a distinctive execution pattern. Typically, while waiting for certain events and not making any forward progress, it makes very few, if any, modifications to the program state. We can infer this lack of program state modifications from the absence of store instructions that change values in memory. Consequently, this execution pattern can also be easily recognized by observing few *unique* stores committed by the program in a given interval, where the uniqueness of a store is determined by having an address or value different from other stores.

An important exception to the above observation arises when a memory location is register allocated. For example, while searching an array structure, a program may not execute any store instructions since the array index variable is likely to be register allocated. Therefore, during long search operations, predominantly found in user code, a program may not execute any store instructions. To avoid such false positive spin detections, we also check for unique load instructions (uniqueness determined by the load address only) when executing user code.

Thus, we detect a kernel spin when the number of unique stores executed within N committed instructions is less than some pre-defined threshold. On the other hand, a user spin will be detected when both unique stores and loads are less than that threshold. In our experiments, we find that for a period of N=1024 committed instructions, a threshold value of eight is effective to detect all known spin loops with few false positives. We also find that this technique leads to a more unified solution by automatically detecting other cases where a VCPU is not doing useful work, such as the OS idle loop and spins in user code.

3.3 Spin Detection Buffer: Implementation

We propose a simple hardware structure, the *Spin Detection Buffer* (SDB), to implement spin detection functionality. We employ two fully associative, eight entry content-addressable memory (CAM) structures to hold the unique stores and loads, respectively. During a given period (of

[3]To determine safe preemption points while inside the OS, they also investigate injecting additional safe points into OS execution by installing a fake device driver and sending interrupts from the VMM to this driver. Since they assume the OS is not holding a lock while executing this driver, they can safely preempt the VCPU at that point. Our evaluation in Section 5 does not implement this additional complexity.

1024 instructions in our simulations), each committed store (and load when in user mode) searches the appropriate CAM to determine if its address/value is unique. A unique load or store then inserts its address/value into the appropriate CAM array. Once either array becomes full, subsequent instructions need not search the CAM.

At the end of the period of committed instructions, we simply check the number of entries in each array. If there are less than eight valid entries in the store array and the VCPU is executing in the OS, the SDB indicates a spin. If there are less than eight entries in both arrays and the VCPU is executing user code, the SDB again indicates a spin. Otherwise, the arrays are flushed and it is assumed that the VCPU is making forward progress. If a user/OS mode change occurs within the period, forward progress is assumed regardless of the number of entries in the arrays.

Updates to these structures are not on the critical path, since they are performed after instructions commit. Because these arrays do not have a strict correctness requirement, other optimizations could be performed to avoid using a CAM, if desired.

4. EXPERIMENTAL EVALUATION

In this section, we describe our evaluation methodology and the target CMP system in more detail, including the hardware virtualization support we assume.

4.1 Methodology

We evaluate several multithreaded applications running under both Solaris 9 and Linux 2.6.10. For this study, we use Virtutech Simics [16], an execution driven, full-system simulator which functionally models a *SunFire 6800* server in sufficient detail to boot unmodified operating systems and run unmodified commercial workloads. This system models UltraSPARC IIICu CPUs, which implement the SPARC V9 ISA. We use Simics as a functional simulator only, and model cache latencies and bandwidth using a detailed memory hierarchy timing simulator. For the study in Sections 5, we are considering only one guest OS, and have no need to virtualize I/O, memory, or privileged instructions. In Section 6, we assume the use of a software VMM that virtualizes I/O, memory, and privileged instructions, but we do not model the overhead of this software VMM.

A brief description of each workload is provided in Table 1. Two workloads, Apache web server and pmake, are evaluated on both OSs, while other commercial workloads on Solaris only, and Spec2000Mix on Linux. All workloads are run for several simulated seconds to warm up the workload and OS disk cache before a checkpoint is taken that is used by our timing simulations. Workloads are warmed up on a (simulated) system with as many PCPUs as VCPUs.

Due to inherent variability in these workloads (primarily from interrupt processing and OS scheduling decisions) as described by Alameldeen, et al. [3], we add a small random variation to the main memory latency, and run several trials of each benchmark per experiment. We present average results, and include the 95% confidence interval on the graphs when it is significant and appropriate.

4.2 Target CMP System

The relevant configuration parameters for our target CMP system are shown in Table 2. The private L2 caches maintain coherence using an invalidate-based MOSI directory protocol. The directory maintains shadow tags of the private L2s, and is located with the appropriate L3 bank. The L3 is shared and strictly exclusive with the L2s. We use a point-to-point crossbar-like interconnect which maintains FIFO ordering among source-destination pairs. CMP cores (PCPUs) are modeled as simple single-issue, single stage, blocking cores to speed simulations.

Apache (Solaris & Linux)	We use the Surge client [6] to drive the open-source Apache web server, version 2.0.48. We do not use any think time in the Surge client to reduce OS idle time. The timing runs are for 15,000 transactions on Solaris and 30,000 transactions on Linux.
pmake (Solaris & Linux)	Parallel compile using GNU make and the Sun Forte Developer 7 C compiler (on Solaris) or gcc-3.3.4 (on Linux). We do not include serial phases. Timing runs are for 1.5 billion user instructions.
Zeus (Solaris)	We use the Surge client to drive the commercial Zeus web server, configured similarly to the Apache web server. The timing runs are for 7500 transactions.
OLTP (Solaris)	OLTP uses the IBM DB2 database to run queries from TPC-C. The database is scaled down from TPC-C specification to about 800MB and runs 192 concurrent user threads with no think time. The timing runs are for 750 transactions.
pgbench (Solaris)	Pgbench runs TPC-B like queries on the PostgreSQL database [1].We run for 3000 transactions.
Spec2000 (Linux)	We concurrently run all Spec2000 benchmarks, except the FORTRAN90 FP benchmarks. Instead, we run two copies of gcc, mcf, and art to keep all 24 VCPUs busy. We run 1 billion user instructions.

Table 1: Workloads used for this study

Processor cores	8 single-issue, single-stage, 1GHz
Priv. L1 instr cache	16kB, 2-way, 1-cycle, coherent
Priv. L1 data cache	16kB, 2-way, 1-cycle, write-back
Private L2 unified cache	512kB, 4-way assoc, 15 cycle load-to-use, 4 banks, 4-stage pipelined, inclusive
On-chip shared L3 cache	8MB, 16-way, 55-cycle load to use, 8 banks, pipelined, exclusive
On-chip interconnect	Point-to-point links, avg 10-cycle latency
Main Memory	365 cycle load-to-use, 40GB/sec

Table 2: Microarchitecture parameters

4.3 Hardware Support for Virtualization

We evaluate a VMM assuming very low level software (e.g., microcode) with hardware support. However, most of the issues and many of the techniques we describe are similarly applicable to a fully software VMM. Though we do not model the overhead of microcode, the primary VMM cost which affects processor virtualization is the process of migrating VCPUs on and off the PCPUs, which we do model faithfully.

As described in Section 2.2, the VMM will often need to switch the VCPU running on a particular CMP core (e.g., a particular PCPU) by saving and restoring the state of that VCPU to other backing storage (such as a cache). A VCPU's *architected state*, which consists of its memory and register values, must be preserved by the VMM. The memory state can simply be communicated as needed via the on-chip coherence network that is already required to support shared memory multiprocessing. Registers must be saved and restored, similar to an OS saving the state of a process when it is context-switched.

The UltraSPARC IIICu architecture we model has a large number of architected registers. Including windowed, alternate global, floating-point, privileged, ASI-mapped, and TLB control registers, this comprises 277 64-bit registers, or 2.2kB (nearly half of this is SPARC register windows).

SPARC V9 uses a software managed TLB. We only transfer the *locked* TLB entries, since the OS cannot tolerate a miss for these, but will handle misses to unlocked entries. Since SPARC V9 uses context IDs to tag TLB entries, we allow multiple VCPUs of the same VM to share the (unlocked) TLB entries, though we flush the TLB when switching between VCPUs of different VMs (in Section 6).

4.4 Implementation

For VMMs that may require frequent VCPU switching, hardware support is useful to reduce the latency of this switching. We propose using a simple mechanism with minimal additional hardware and no special-purpose storage by simply storing the VCPU state in the memory subsystem. A portion of the physical address space is set aside for this storage, and microcode is used to load and store the state values. When the VMM chooses to pause a VCPU, it interrupts the corresponding PCPU, which flushes its pipeline and executes the microcode to save the current VCPU state and also restore the state for the next VCPU (as specified by the VMM). The latency of this operation is then determined by the available memory read (and write) ports and the cache bandwidth. While this solution incurs some cache overhead, it is a small fraction of the multi-megabyte caches found on current chips. If the VMM chooses to run the paused VCPU on another CMP core, the VCPU state can be transparently migrated using the on-chip coherence protocol.

Recently, both Intel and AMD have introduced hardware virtualization support with their Virtualization Technology (VT) and SVM project, respectively. Though the microarchitectural details are not yet clear, both support hardware VCPU switching and migration at the direction of a software VMM. We expect the actual mechanisms to be similar to those we evaluate.

5. RESULTS

This section examines the ability of several techniques to mitigate the synchronization overhead apparent from Section 2.2.3. We again expose the challenges of an overcommitted VM by using a single guest OS with 24 VCPUs running on an 8 PCPU system.

5.1 Implementing Safe Points

Though conceptually appealing, implementing safe point aware preemption is not straightforward. To avoid starvation of other VMs, Uhlig., et al. [26], suggest using a 1ms *grace period*, after which point a VCPU will be preempted even if it has not reached a safe point. When the number of VCPUs exceeds the number of PCPUs, this grace period is also necessary to avoid deadlock in a single VM.

In Solaris, which frequently sends software interrupts to implement cross-calls, it is critical to schedule the interrupted VCPU as soon as possible to ensure forward progress of the sender. But when several running VCPUs are either sending cross-calls or attempting to acquire a lock held by one of the senders, most executing VCPUs start spinning in the kernel (and are thus considered unsafe to preempt). Often no PCPU is available to run the interrupted VCPUs until the grace periods for these spinning VCPUs expire. There are a range of possible strategies for dealing with this problem, including reducing the grace period when paused VCPUs have outstanding interrupts, or delaying scheduling the interrupted VCPU until it can be scheduled on its currently assigned PCPU (to maintain affinity). We find that the best overall policy is to maintain the 1ms grace period, but run the interrupted VCPU on the next available PCPU, regardless of cache affinity. While this results in frequent VCPU migrations, and requires a centralized control point, it greatly reduces the synchronization overhead compared to the other alternatives.

Uhlig, et al., do not observe this seemingly pathological cross-call behavior with their evaluation methodology, since a) this mode of synchronization is fairly uncommon in Linux, and b) they do not consider overcommitted VMs, and can thus concurrently schedule all VCPUs of a given guest VM when necessary.

5.2 Comparison of Synchronization Management Techniques

In Figure 2, we present the relative instruction count with different policies for managing synchronization overheads. From left to right, the bars for each benchmark represent timeslices of 100μs and 10μs (without any additional synchronization management), safe point-aware preemption (with a 100μs timeslice and 1ms grace period) and our SDB scheme with a 100μs timeslice, respectively. The results are normalized to the 100μs timeslice. Several observations can be made from this figure. First, we find that, except for Zeus and pgbench, safe point-aware preemption is quite effective in reducing the instruction overhead due to synchronization. For these two benchmarks, on the other hand, the safe point-aware preemption scheme was often unable to find preemptible VCPUs during concurrent cross-calls (as discussed in Section 5.1), leading to a dramatic increase in synchronization overhead. Second, we find that our SDB is the most effective technique at reducing the instruction overheads in almost all cases. For example, Zeus and OLTP execute 3.15X and 1.87X fewer instructions compared to the 100μs timeslice, respectively.

While all three techniques for spin management (10μs timeslice, safe-points, and SDB) are effective at reducing extra committed instructions compared to the 100μs timeslice, they have different effects on cache performance. Figure 3 shows the total memory stall time due to L1 misses, relative to the 100μs timeslice. This is total memory stall, for a

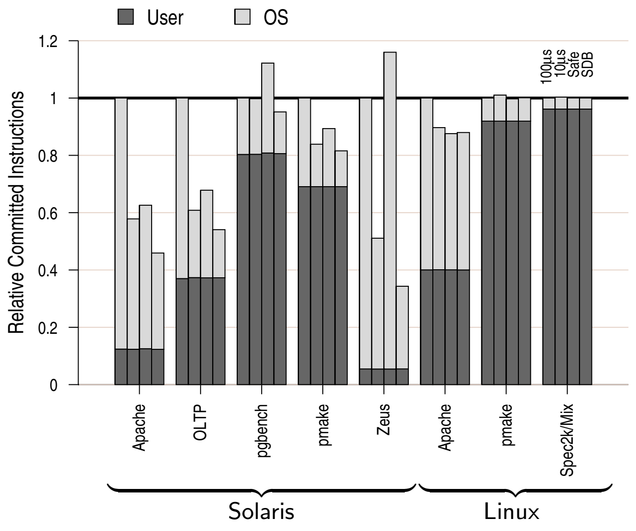

Figure 2: Relative instruction counts of synchronization management techniques. Four bars for each benchmark (from left to right) represents results for a 100μs timeslice, 10μs timeslice, safe point aware preemption and SDB. Results are normalized to the 100μs timeslice (lower is better).

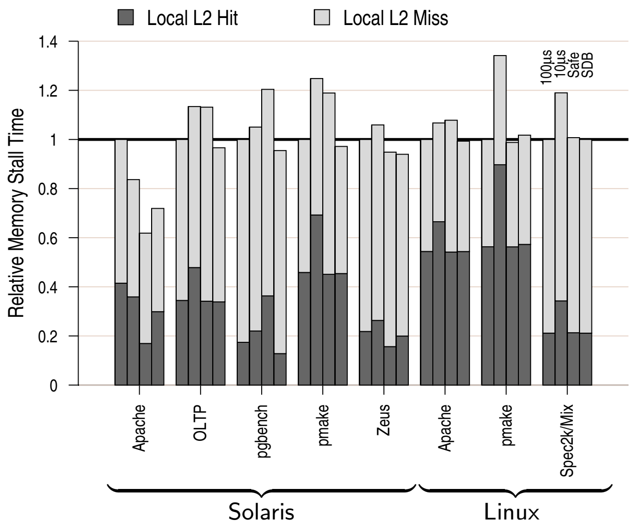

Figure 3: Relative memory stall time of synchronization management techniques. Four bars for each benchmark (from left to right) represents results for a 100μs timeslice, 10μs timeslice, safe point aware preemption and SDB, respectively. Results are normalized to the 100μs timeslice (lower is better).

given amount of work, regardless of the number of instructions. Stall time is broken up into stalls occurring at the local L2, and other stalls (from remote L2s, the on-chip L3, or main memory). Similar to the study of scheduling policies for cache affinity by Torrellas, et al. [24], we see that for most cases, the 10μs timeslice has a much larger memory stall time than the 100μs timeslice. Apache is the only exception because its spin loops incur L2 coherence traffic.

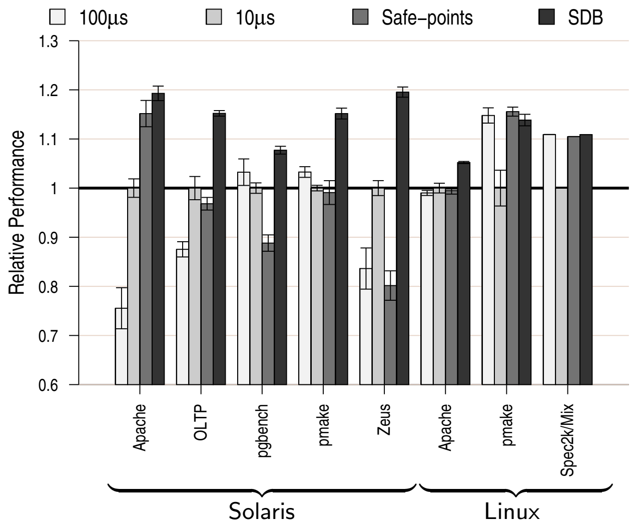

Figure 4: Relative performance of of synchronization management techniques (higher is better).

The safe-points scheme and the SDB typically have similar stall time from the local L2, but the safe-points scheme often has a larger L2 miss component. Since the safe point-aware preemption scheme schedules VCPUs eagerly across many PCPUs when they have pending software interrupts (in order to reduce spin instructions as much as possible), it can hurt cache affinity.

We also show the relative performance of these four configuration in Figure 4 (normalized to the 10μs timeslice). For application where there is little spinning, the 100μs timeslice outperforms the 10μs timeslice due to better cache behavior. For application where spins are a problem, the 10μs timeslice is better due to substantial reduction in committed instructions. For benchmarks where the safe-points scheme is effective at mitigating spin and does not incur extra L2 misses, it performs similarly to the SDB. On the other hand, safe-points performs poorly for many benchmarks which exhibit a significant spin overhead. In every case except pmake on Linux and SpecMix, the SDB outperforms the other schemes by a significant margin.

While it is clear that the SDB is effective in reducing the synchronization overheads, it is difficult to precisely determine how accurate it is. After carefully browsing the kernel source and observing program execution, we are unaware of any case where it fails to detect a spin (false negative). It also generates very few false positives in our workloads. For Spec2000Mix, which is a multi-programmed workload (single-threaded SPEC benchmarks) without any spin loops in user code, it triggers about 130 false positives in the course of over one billion committed instructions.

6. SERVER CONSOLIDATION STUDY

In this section, we present a case study to demonstrate the effectiveness of using our spin detection techniques for CMP core virtualization on a simple server consolidation setup. Server consolidation is an emerging application which aims to simplify the operation and reduce the cost of an organization's multiple servers by using virtualization to run multiple services on the same physical server [5, 10, 23, 28].

While gang scheduling is not possible for many applications described in Section 2.1, it is often a viable option for server consolidation. However, gang scheduling is inflexible, and in this section, we show that pursuing other policies can improve both throughput and performance isolation.

6.1 Traditional Gang Scheduling

Consider the case of two services, currently set up and optimally configured and tuned to run on different 8-processor servers that are being consolidated onto one 8-processor server. Each original server can run as a guest VM on the new hardware platform with minimal configuration changes (one goal of consolidation), if each guest VM is configured to see eight VCPUs. The VCPUs of each VM are then gang scheduled, so that PCPUs alternate between the two guest VMs, allowing each to run for a given *timeslice*.

While it may be possible to share read-only code and data between multiple VMs running the same OS version [28], we do not allow this in our experiments. It is thus desirable to choose a longer timeslice to avoid thrashing in the caches and TLBs [24]. However, most applications cannot sustain an arbitrarily long delay in responses (for requests to a VM that is not scheduled). For example, low response times for Multiplayer Online Game (MOG) servers are critical in order to provide a satisfactory user experience [9]. Therefore, timeslices in the range of less than one to several milliseconds are generally used.

An alternative is to set up these guest VMs to only see four VCPUs each, but this has two drawbacks. First, it requires modifying the previously tuned configuration of each server, and second, it does not allow one guest VM to utilize all of the PCPUs to handle a burst in demand.

When many VCPUs of a guest VM are idle, gang scheduling leads to inefficient use of the processors since idle VCPUs of one VM are scheduled even when non-idle VCPUs of another VM could run. To combat this problem, some gang schedulers use techniques to identify the idle loop of the guest OSs [12]. However, even with precise information about which VCPUs of VM A are idle, it is not straightforward to allow only some of VM B's VCPUs to execute, since they will be seriously affected by kernel synchronization unless *all* non-idle VCPUs of VM B can be co-scheduled.

6.2 Logical Partitioning

Our proposed hardware technique to manage the synchronization overhead can allow much more flexibility in choosing a scheduling policy. A VMM designer can then choose a policy that best serves the specific needs (e.g., one that provides maximum performance isolation in a untrusted environment while maintaining low response latency).

As an alternative to gang scheduling, we propose to use *Logical Partitioning*, to demonstrate the potential of a more flexible policy. In this scheme, we distribute PCPUs among guest VMs, and only execute a subset of each VM's VCPUs at a time. Since guest VMs are allocated fewer PCPUs than VCPUs (for an extended period of time, if not permanently), they are overcommitted. Especially when running shared memory applications where different VCPUs from the same VM share instructions and data, logical partitioning can lead to better cache locality by improving constructive interference in the private structures of a PCPU (within a logical partition) while eliminating destructive interference from the other VM. Simple extensions to this policy, for example to allow dynamic load balancing, might make it appropriate for a wider range of application domains, and are not the focus of this case study. Instead, we show the effectiveness of this policy at increasing throughout and improving performance isolation, when compared to gang scheduling for one particular setup. Some systems, such as the Power5 [5], have an interface to avoid synchronization overheads, and could already take advantage of this policy.

6.3 Throughput Comparison

For this comparison, we have set up eight server-consolidation workloads running Solaris on the 8-PCPU CMP system described in Section 4. Each workload combines two 8-VCPU guest VMs running benchmarks (on Solaris) from Section 4. Each guest VM is configured with its own I/O devices and physical memory space, but VMs dynamically share the processors, caches, and TLBs. We are assuming the use of a software VMM, similar to VMWare, which virtualizes I/O, memory, and privileged instructions. Since we are only investigating processor virtualization, we do not model the overhead of this software VMM. The two guest OSs are allocated enough physical memory so that the VMM does not need to swap *real* memory.

One set of the these workloads combines two guest VMs, where each guest VM is warmed up and fully utilizes all 8 VCPUs (i.e., it achieves near 0% idle time when run by itself on an 8-processor CMP). The other set combines guest VMs where each guest is warmed up, but only utilizes approximately 50% of the 8 VCPUs when run by itself. The full utilization case represents a worst-case load for the consolidated server, while the half-utilization represents the expected common case. To cover workloads from different domains, we evaluate gang scheduling with two different timeslices: 10ms (which is used by many OSs as a scheduling timeslice) and 100μs (to ensure low response time in environments like MOG).

In Figure 6 we show the performance of the consolidated workloads for three experiments: a) gang scheduling with a 100μs timeslice, b) gang scheduling with a timeslice of 10ms, and c) partitioning the PCPU resources evenly between the guest VMs (with a 100μs timeslice). Performance is relative to gang scheduling at 100μs. Speedup, for two experiments $ExpA$ and $ExpB$ is simply the average speedup of the two VMs and is calculated as:

$$Speedup = \frac{1}{2}\left(\frac{UserIPC_{VM0}^{ExpB}}{UserIPC_{VM0}^{ExpA}} + \frac{UserIPC_{VM1}^{ExpB}}{UserIPC_{VM1}^{ExpA}}\right)$$

where $UserIPC$ is the number of committed *user* instructions over the *total* number of cycles. Based on our own experiments (e.g., Figure 1) and recent literature [29], we find that user instructions committed is a good proxy for throughput metrics in commercial workloads. The graph breaks down the speedup component from each VM.

We also show the relative L2 misses incurred in each of these experiments in Figure 5. Partitioning the PCPU resources, along with private caches and TLBs, results in 18–29% fewer private L2 misses than gang scheduling with the 100μs timeslice, causing an overall speedup of 10–25% as evident from the graph. The best speedups are from the lower utilization workloads due to better core management. Longer (10ms) timeslices for gang scheduling improve throughput, as expected. When the VMs are fully utilizing the processors, both runtime and L2 misses of gang

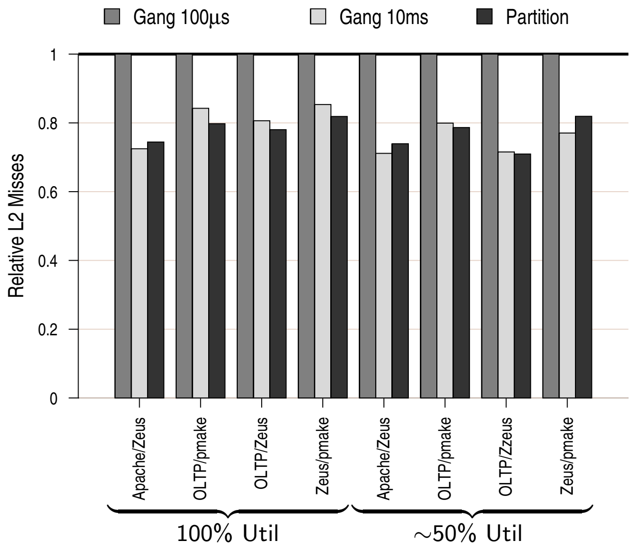

Figure 5: Relative L2 misses of different scheduling policies. Results are normalized to gang scheduling with a 100μs timeslice (lower is better).

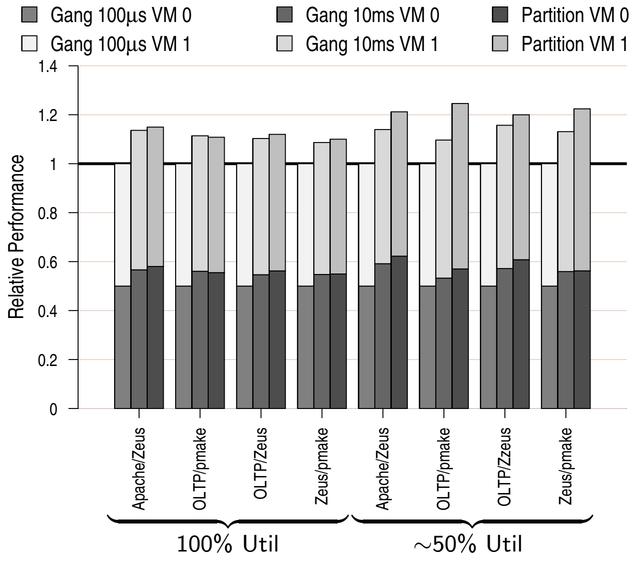

Figure 6: Relative performance of different scheduling policies. Results are normalized to gang scheduling with a 100μs timeslice (higher is better).

scheduling are similar to partitioning. However, much longer timeslices (100ms or more) do not continue to improve gang scheduling for most benchmarks. A timeslice of 1ms, used in other systems [26], performs in-between 100μs and 10ms.

6.4 Performance Isolation

For this study, we designed a set of experiments which pair a VM running one of the workloads described in Section 4 with a VM running one of two microbenchmarks. The first, labeled *Comp* is computationally intensive with a small cache footprint. The second, labeled *Stream* is extremely cache intensive: it streams through a large array, effectively replacing the entire private L2 caches (but only a fraction of the shared L3). We use a 1ms timeslice for gang scheduling in this experiment.

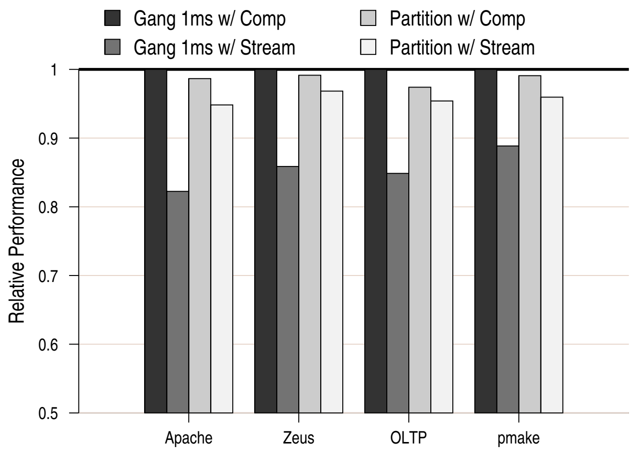

Figure 7: Performance isolation. Bars represent performance when a commercial workload in one VM is paired with a second VM running either a computationally-intensive (*Comp*), or cache-intensive (*Stream*), microbenchmark.

Figure 7 compares the performance of configurations which perform gang scheduling and partitioning PCPUs. We show the relative performance of the VM running the commercial workloads only (i.e., we are not measuring the performance of the microbenchmarks), relative to gang scheduling with the *Comp* microbenchmark. When combined with the *Comp* microbenchmark, the commercial workloads with partitioning are slightly slower than gang scheduling (up to 2.5%) as expected, since spreading the commercial workloads across all eight private L2s yields more aggregate cache space. However, when combined with the *Stream* microbenchmark, the commercial workloads' performance suffers substantially with gang scheduling (by 11–18%). The partitioning policy also leads to performance loss when combined with the *Stream* microbenchmark, largely due to the additional capacity and bandwidth pressure on the shared L3. However, the relative performance lost due to partitioning is several times lower than that of gang scheduling.

7. CONCLUSIONS

Traditionally, gang scheduling has been employed to run multiprocessor OSs in virtual machines. However, supporting emerging architecture and application trends, such as efficient server consolidation, dynamic thermal management and speculative or redundant multithreading, requires more flexible scheduling policies. In the absence of gang scheduling, which concurrently runs all VCPUs of a guest OS, kernel synchronization overhead becomes a significant impediment to the performance of these virtual machines.

We propose a simple hardware technique to detect when a VCPU is spinning, without requiring any software modification, and preempt that VCPU in favor of one which is making forward progress. Our results show that the synchronization overhead can be substantially reduced using this technique, thereby allowing more flexible scheduling policies than are possible with gang scheduling.

Finally, we perform a server consolidation case study to examine the benefits of more flexible scheduling enabled by our spin detection technique. We propose a policy to partition a subset of the physical cores, along with private resources like caches and TLBs, among guest VMs. We show that this partitioning achieves 10–25% speedup over gang scheduling by reducing private L2 cache misses and improving hardware resource utilization, and improves performance isolation of private caches.

Acknowledgments

This work was supported in part by National Science Foundation grants CCR-0311572 and EIA-0071924, and by donations from Intel Corporation. We also thank Saisanthosh Balakrishnan and the anonymous reviewers for their comments on this paper.

8. REFERENCES

[1] PostgreSQL. http://www.postgresql.org/.

[2] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Dec 2005.

[3] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[4] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, 1992.

[5] W. Armstrong, R. Arndt, D. Boutcher, R. Kovacs, D. Larson, K. Lucke, N. Nayar, and R. Swanberg. Advanced virtualization capabilities of POWER5 systems. *IBM J. Research and Development*, 49(4/5), 2005.

[6] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 International Conference on Measurement and Modeling of Computer Systems*, 1998.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.

[8] K. Chakraborty, P. Wells, and G. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (to appear)*, 2006.

[9] G. Deen, M. Hammer, J. Bethencourt, I. Eiron, J. Thomas, and J. Kaufman. Running Quake II on a grid. *IBM Systems Journal*, 45(1), 2006.

[10] R. Figueiredo, P. A. Dinda, and J. Fortes. Resource virtualization renaissance. *IEEE Computer*, 38(5):28–31, 2005.

[11] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[12] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th Symposium on Operating Systems Principles*, 1999.

[13] M. Hohmuth and H. Hartig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

[14] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st International Symposium on Computer Architecture*, 2004.

[15] J. R. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[16] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[17] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Distributed Computing Systems*, 1982.

[18] M. D. Powell, M. Gomaa, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[19] B. Rosenburg. Low-synchronization translation lookaside buffer consistency in large-scale shared-memory multiprocessors. In *Proceedings of the 12th Symposium on Operating Systems Principles*, 1989.

[20] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[21] J. E. Smith, S. S. Sastry, T. Heil, and T. M. Bezenek. Achieving high performance via co-designed virtual machines. In *International Workshop on Innovative Architecture*, 1999.

[22] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[23] Sun Microsystems, Inc. Sun enterprise 10000 server: Dynamic system domains. http://www.sun.com/servers/white-papers/domains.html. Viewed 6/23/2006.

[24] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 24(2):139–151, 1995.

[25] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5), 2005.

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

[27] VMWare. ESX Server - best practices using VMware virtual SMP. www.vmware.com/pdf/vsmp_best_practices.pdf. Viewed 6/23/2006.

[28] C. A. Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[29] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[30] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[31] R. W. Wisniewski, L. Kontothanassis, and M. L. Scott. Scalable spin locks for multiprogrammed systems. Technical Report TR454, University of Rochester, Rochester, NY, USA, 1993.

[32] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Trans. Parallel Distrib. Syst.*, 2(2):180–198, 1991.