Spin Detection Hardware for Improved Management of Multithreaded Systems

Tong Li, Alvin R. Lebeck, *Senior Member*, *IEEE*, and Daniel J. Sorin, *Member*, *IEEE*

Abstract—Spinning is a synchronization mechanism commonly used in applications and operating systems. Excessive spinning, however, often indicates performance or correctness (e.g., livelock) problems. Detecting if applications and operating systems are spinning is essential for achieving high performance, especially in consolidated servers running virtual machines. Prior research has used source or binary instrumentation to detect spinning. However, these approaches place a significant burden on programmers and may even be infeasible in certain situations. In this paper, we propose efficient hardware to detect spinning in unmodified applications and operating systems. Based on this hardware, we develop 1) scheduling and power policies that adaptively manage resources for spinning threads, 2) system support that helps detect when a multithreaded program is livelocked, and 3) hardware performance counters that accurately reflect system performance. Using full-system simulation with SPEC OMP, SPLASH-2, and Wisconsin commercial workloads, we demonstrate that our mechanisms effectively improve the management of multithreaded systems.

Index Terms—Deadlock, livelock, multiprocessor, multithreaded system, performance counter, scheduling, spinning, synchronization, virtualization.

1 Introduction

In an ideal world, programs would be perfectly tuned and bug-free. In reality, however, this is often not true. One particular situation is when programs use spinning (also known as busy-waiting) synchronization. When a thread is spinning, it wastes resources such as the processor and power. Excessive spinning often indicates performance or correctness (e.g., livelock) problems.

If we could detect whether a thread is spinning, there would be many benefits. For example, the operating system could de-schedule long-spinning threads to allow other threads to run. Such a scheme would be particularly beneficial to virtual machine systems, given the current trend of server consolidation [1], [2], [3], [4]. When detecting long-spinning applications or guest operating systems, the virtual machine monitor (VMM) could de-schedule an entire virtual machine (VM) such that other VMs could run. Another benefit of detecting spinning is that, if a group of threads are spinning simultaneously, we could analyze if a livelock exists and provide debugging support to programmers. We could also design more accurate hardware performance counters that factor out spinning instructions, because these instructions often cause inflated performance statistics, such as instructions-per-cycle (IPC).

Prior research has studied ways to detect spinning. Examples include the Denali [4] and Xen [1] virtual machine systems, which detect spinning in OS idle loops by modifying the OS source code. For the Denali system, Whitaker et al. [4] showed that descheduling a VM spinning in the idle loop prevented a 66 percent throughput degradation for their applications. The VMWare ESX Server performs binary translation to detect idle-loop spinning in certain operating systems [5]. Intel recognizes the difficulty of identifying spinning and suggests visually inspecting synchronization code [6] with the aid of the VTune Performance Analyzer [6]. These approaches, however, place a significant burden on programmers. They either suit only specific spinning scenarios or require source modification, which is difficult and sometimes even infeasible.

To overcome the limitations of existing approaches, we design hardware to detect spinning. Our hardware is small (less than 1 KB) and off the critical path. It detects spinning in unmodified applications and operating systems with no performance overhead. These design features are similar to what Intel is trying to achieve with their hardware support for virtualization [2].

Our hardware enables us to realize the benefits of spin detection. First, we study how to improve OS scheduling and power management by not allocating resources to spinning threads. Our evaluation shows that this design can improve application performance by 16.4 percent even in simple settings. Second, we extend the spin detector to detect potential livelocks. Intuitively, if all threads of a program are simultaneously spinning, then the program is livelocked. Finally, we develop accurate performance counters that factor out spinning instructions.

We make three contributions in this paper:

- We define the conditions of spinning. These conditions enable general spin detection without the need of application semantic knowledge.

- *T. Li is with Intel Labs, Mail Stop JF2-65, 2111 NE 25th Ave., Hillsboro, OR 97124. E-mail: tong.n.li@intel.com.*
- *A.R. Lebeck is with the Department of Computer Science, Duke University, Box 90129, Durham, NC 27708-0129. E-mail: alvy@cs.duke.edu.*
- *D.J. Sorin is with the Department of Electrical and Computer Engineering, Duke University, Box 90291, Durham, NC 27708-0291. E-mail: sorin@ee.duke.edu.*

Manuscript received 8 July 2005; revised 13 Feb. 2006; accepted 4 Mar. 2006; published online 26 Apr. 2006.
Recommended for acceptance by A. Gonzalez.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0325-0705.

TABLE 1
Spin Loop Examples in SPARC Assembly

0x2a134:	cmp	%i4, -0x1	0xfbf118:	cmp	%o0, 0x0
0x2a138:	be	0x2a150	0xfbf11c:	bne,a,pt	%icc, 0xfbf118
0x2a13c:	nop		0xfbf120:	ld	[%o1], %o0
0x2a140:	ld	[%i5], %i3		(b)	
0x2a144:	cmp	%i3, -0x1			
0x2a148:	bne,a	0x2a134	0x1002c990:	brnz,a,pt	%o0, 0x1002c990
0x2a14c:	ld	[%i5], %i4	0x1002c994:	ldub	[%i0 + 0xa0], %o0
	(a)			(c)	

(a) Spinning in OpenMP, (b) spinning in DB2 controller, and (c) spinning in Solaris dispatcher.

- We develop efficient hardware for detecting if a thread is spinning. Our hardware releases the burden on programmers and supports unmodified applications and operating systems.
- We extend our spin detection hardware to enable 1) efficient scheduling and power management, 2) hardware support for detecting livelock, and 3) accurate performance counters.

The remainder of this paper is organized as follows: In Section 2, we discuss the origins of spinning in multithreaded systems (including SMT and multiprocessors), our motivation for hardware spin detection, and the conditions of spinning. We present the spin detection hardware in Section 3. We show our simulation methodology and evaluation of the hardware in Section 4. We then study the three applications of our spin detection hardware in Sections 5, 6, and 7. We discuss related work in Section 8 and conclude in Section 9.

2 Spinning in Multithreaded Systems

In this section, we discuss why spinning exists in multithreaded systems (Section 2.1), why we design hardware to detect spinning (Section 2.2), and the conditions that enable hardware spin detection (Section 2.3). To avoid confusion between *threads* (software constructs) and *thread contexts* (hardware for running threads in some systems), we refer to the latter as processors without loss of generality.

2.1 Origins of Spinning

Spinning is a waiting mechanism with which the waiting thread continuously checks for the occurrence of a synchronization event. Blocking is an alternative mechanism that allows the waiting thread to be suspended and its resources to be reused. Both spinning and blocking are widely used in multithreaded systems, including SMT and multiprocessors.

Locks, barriers, and flags are the three most prevalent synchronization abstractions and their implementations often consist of spinning. In the area of high-performance computing, OpenMP is the de facto standard for shared-memory parallel programming. OpenMP locks and barriers use spinning as a means for thread synchronization. Table 1a shows the SPARC assembly code of a spin loop in `omp_set_lock`, a lock acquire function frequently used by programs in the SPEC OpenMP benchmark suite (SPEC OMP V3.0) [7]. Spinning also occurs in various commercial server applications, such as database servers. Table 1b shows an example of flag spinning in the IBM DB2 database system controller process `db2sysc`. Besides user-space applications, spinning is also widely used within operating system kernels. Spin locks and flags are prevalent in all existing operating systems that support SMPs. Table 1c shows a spin loop in the Solaris 8 kernel dispatcher. In many operating systems, the idle loop is also a spin loop in which the operating system waits for work.

2.2 Motivation for Hardware Spin Detection

As we discussed in Section 1, detecting spinning has many benefits. One approach to detecting spinning is to instrument the spin code. However, instrumentation has two limitations.

First, program source code is often unavailable, making it difficult, if not impossible, to support proprietary and legacy code. Although binary instrumentation is possible, it places a burden on programmers and may introduce performance overhead.

Second, even when source code is available, identifying all spinning in a large code base places a significant burden on programmers, especially when the code does not use common libraries (e.g., spinning on a flag) [6]. Some code, such as the Solaris 8 idle loop, exhibits spinning only on certain execution paths, and thus is difficult to identify statically.

To free the burden from programmers and avoid perturbing application performance, we design hardware to dynamically detect if a thread is spinning. Our hardware supports unmodified applications and operating systems with no performance overhead. These features are similar to the design goals of the Intel Virtualization Technology [2]. Our hardware particularly suits virtual machine environments in which support for unmodified and legacy code is desired and detecting spinning is essential for high performance [8].

In Fig. 1, we compare software and hardware spin detection for two SPLASH-2 [9] benchmarks, `Radiosity`

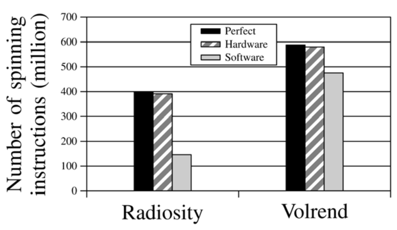


Fig. 1. Comparison between software and hardware spin detection.

and `Volrend`, on a simulated 8-processor SMP system (see Section 4 for our simulation methodology). The *total* bar represents the total number of spinning instructions committed in the benchmark's entire execution. We obtain this value by instrumenting all spin loops in the benchmark's program and the PARMACS library [10]. The *software* bar corresponds to spins detected by a software detection mechanism, which instruments spin loops only in the PARMACS library. The *hardware* bar corresponds to spins detected by our hardware (details in Section 3). As we can see, our hardware detects nearly all spins and the software mechanism misses many of them. The spins that the hardware mechanism misses are due to having finite hardware and, thus, having to use heuristics. The spins that the software mechanism misses are due to flag spinning in the benchmarks' programs (not the library). However, identifying flag spinning in application programs requires thorough understanding of the programs and, thus, can be a tremendous burden on programmers [6].

To design hardware for spin detection, we face the challenge that hardware has no knowledge of application semantics and, thus, must exploit features common in all forms of spinning. In the next section, we study the general conditions that enable dynamic spin detection in hardware.

2.3 General Conditions for Identifying Spinning

Intuitively, if a thread executes a static instruction and later executes it again (e.g., in another iteration of a loop) with the state of the system unchanged, then the thread is spinning between the two executions of that instruction. More precisely, a thread on processor P is spinning between time t_a and time t_b if its execution satisfies the following two conditions.

Spin Condition 1: The observable state of the thread for the period between t_a and t_b is the same at t_a and t_b.

For a given period of time in a thread's execution, the *observable state* of the thread includes the architectural registers and memory locations accessed by the thread during the given period. Since the observable state includes the program counter (PC), an important implication of this condition is that the thread executes the same static instruction (PC) at both t_a and t_b. Therefore, all instructions executed between t_a and t_b form a cycle in the control flow graph of the thread's program. We call this cycle a *spin loop* (although it can be more complex than a simple "loop") and each instruction executed in the spin loop a *spinning instruction*. If Spin Condition 1 is satisfied, all the architectural registers and memory locations accessed by the thread between t_a and t_b have the same values at t_a and t_b. Thus, the thread performs no useful work with respect to its computation on processor P. This condition, however, does not preclude the observable state from changing between t_a and t_b, as long as it changes back to its initial state by t_b.

Spin Condition 2: Any change made by the thread to its observable state between t_a and t_b is not observed by any thread outside processor P.

This condition captures the scenario in which changes made by the thread to its observable state between t_a and t_b may cause changes to the observable state of another thread running on a different processor. This scenario can happen if the observable states of the two threads overlap, e.g., they both access the same memory locations. If Spin Condition 2 is satisfied, then the execution of the thread between t_a and t_b does not affect the computation of any thread outside processor P.

The above two conditions provide the basis for spin detection and ensure that we never incorrectly detect spinning that does not exist. In some situations, these conditions may be restrictive. For example, when implemented with a two-phase waiting algorithm [11], [12], a thread may spin for a while and then block. During each spin loop iteration, the thread typically increments a counter until it exceeds some threshold. Thus, Spin Condition 1 is violated because the observable state of the thread is different at the start and end of each iteration. However, from the programmer's point of view, the thread does spin because all the state changes have no contribution to the intended computation. To detect such spinning, software mechanisms can be used in conjunction with our hardware. On the other hand, the time of such spinning in a two-phase waiting algorithm is often programmed to be short to optimize the overall synchronization performance. Thus, missing the detection of such short spinning does not have much impact on our scheduling, power management, and livelock detection mechanisms.

3 Dynamic Spin Detection

In this section, we present our hardware design. To detect spinning, our hardware needs to dynamically check the two conditions in Section 2.3. For Spin Condition 1, we observe that any control flow cycle must have an instruction that causes a backward transfer of control flow. We call this instruction a *backward control transfer* (BCT). A BCT can be a backward conditional taken branch, unconditional branch, or jump instruction. Although other instructions such as calls, call returns, traps, and trap returns can also cause backward transfers of control flow, practical spin loops rarely rely only on these instructions to implement control flow cycles. Thus, we do not include them as backward control transfers.

For Spin Condition 1, we check whether a processor commits a BCT at time t_a and later commits the same BCT (PC) again at time t_b. If so, between t_a and t_b, the processor has executed one iteration of a control flow cycle. To further check whether the observable state of the thread is the same at t_a and t_b, we divide it into observable memory state and observable register state, and we discuss these issues separately in Section 3.1 and Section 3.2. We also show that, with our approach, if Spin Condition 1 is satisfied, then Spin Condition 2 is satisfied as well. We explain how to detect nested spin loops in Section 3.3 and describe our hardware in detail in Section 3.4. We illustrate the operation of our hardware via an example in Section 3.5.

3.1 Observable Memory State

In this section, we consider the observable memory state part of Spin Condition 1. Given the execution of a thread between time t_a and time t_b, we assume that any *non-silent store* [13] executed by the thread can cause its observable memory state at t_b to be different from its initial observable memory state at t_a. A non-silent store is a store instruction that writes to a memory address with a value different from the existing value at that address. Our assumption is conservative because the location changed by a non-silent store may later change back to its initial value. However, it simplifies our hardware for checking Spin Condition 1. Any non-silent store committed between t_a and t_b indicates that the observable memory state may differ at t_a and t_b. To detect non-silent stores to cacheable memory locations, we use the ECC store verify approach of Lepak and Lipasti [13]. For stores to non-cacheable memory locations, such as I/O addresses, we conservatively assume they are all non-silent.

We assume an architecture in which processors share memory but not registers, i.e., instructions on one processor cannot write registers on a different processor.[1] Thus, for Spin Condition 2, a thread can only change the observable state of another thread by modifying a shared memory location. Given a thread's execution between t_a and t_b, if all stores are silent, then the thread makes no change to shared memory locations, and thus Spin Condition 2 is satisfied. Therefore, by detecting non-silent stores, we can check if the observable memory state part of Spin Condition 1 and the entire Spin Condition 2 are both satisfied.

3.2 Observable Register State

In this section, we describe how to check the observable register state part of Spin Condition 1. Given the execution of a thread between time t_a and time t_b, the observable register state of the thread is the same at t_a and t_b if and only if the entire register state of the processor is the same. The register state of a processor consists of all the architectural registers, including control registers, but excluding performance counters and registers mapped to the I/O space.[2] Therefore, for Spin Condition 1, we can save the processor register state at t_a and check if it is the same at t_b. This approach is logically simple, but the challenge is to implement it efficiently.

To efficiently check if the observable register state remains the same, we note that the only registers that can change between t_a and t_b are those written by the thread. Thus, checking if the observable register state of a thread remains the same is equivalent to checking if the registers written by the thread are the same. In practice, spin loops are often small and each loop iteration only writes a small number of architectural registers. In all the workloads we study, the maximum number of architectural registers written in a spin loop iteration is only 17. Therefore, by maintaining and comparing only the registers written by the thread, we can greatly reduce hardware costs. The difficulty, however, is that the processor does not know a priori at t_a which registers the thread will write during its execution between t_a and t_b. Nevertheless, our implementation can discover these registers as the execution progresses.

We use a *Register Update Buffer* (RUB) to dynamically track the architectural registers written by a thread in each iteration of a potential spin loop. *An invariant in our algorithm is that the RUB always keeps the architectural registers whose current values are no longer the same as when the loop iteration began.* The RUB is empty when the processor starts a potential spin loop iteration, i.e., when it commits a BCT. For each instruction it then commits, the processor checks if the instruction's architectural (i.e., logical) destination register is already in the RUB. If not, the processor has discovered a new register written by the thread. It then compares the new value of this register (i.e., the value being committed) to its old value (i.e., the value before being overwritten by the commit). If not equal, the processor adds the register number and its *old* value to the RUB. If the register is already in the RUB, then the processor compares its new value to its current value in the RUB. If equal, it deletes this register from the RUB; otherwise, no action is necessary. With this algorithm, when the processor reaches the end of the iteration, i.e., when it commits a new dynamic instance of the same static BCT, if the RUB is empty, then the observable register state of the thread is the same as when the iteration began. We will present full details of the RUB implementation in Section 3.4.

3.3 Nested Loops

Identifying a control flow cycle (even without trying to detect spinning) involves saving the PC of a BCT and checking if the thread commits the same PC again. At a first glance, it seems sufficient to save only the PC of the most recently committed BCT. This is, however, not true in the presence of nested loops. Table 2 shows the SPARC assembly code for an example nested spin loop in the lock acquire routine of the PARMACS library. When a thread fails to acquire a lock, it spins until succeeding both the test and test&set.

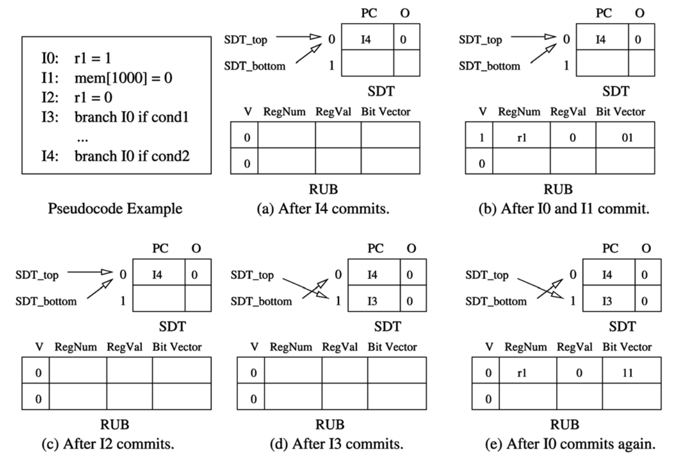
To identify a nested spin loop, we need to keep multiple PCs, corresponding to the BCT of each level of the nested loop. Since the depth of nesting is typically small in realistic spin loops, we only need to keep the PCs for a handful of the most recent BCTs. We add a *Spin Detection Table* (SDT) after the commit stage of the processor pipeline. For each BCT it commits, the processor searches the SDT for a matching PC. If found, a control flow cycle is identified. If not found, the processor inserts the PC of this BCT at the top of the SDT, and pushes down all the existing SDT entries, thereby evicting the bottom entry if the SDT is full. To avoid physically moving the SDT entries, we manage the SDT as a circular array and use two counters, `SDT_top` and `SDT_bottom`, to maintain the indices of the top and bottom SDT entries. For all our workloads, experiments show that a 16-entry SDT provides the same results as an unbounded SDT.

1. We treat I/O devices as processors that share I/O space memory with the host processor to which they are connected.

2. Not all architectures require performance counters to be implemented and, when they are implemented, including them (e.g., cycle counters) in processor register state would cause the state to always change. Since no practical spin loop involves performance counters, we do not include them in processor register state. We do not include memory-mapped registers in processor register state because they are treated as part of the memory state.

TABLE 2
Nested Spin Loop in PARMACS's Lock Acquire Routine

	0x12cac	cmp	%l1, 0x0	
TTS:	0x12cb0	bl,a	0x12cd4	// test and test&set
	0x12cb4	ld	[%i0], %g2	// load lock status
	. . . (never here) . . .			
	0x12cd4	cmp	%g2, 0x0	// test lock
BCT 1:	0x12cd8	bne	0x12cb0	// fail test, retry TTS
	0x12cdc	cmp	%l1, 0x0	// delay slot
	0x12ce0	mov	%i0, %o0	
	0x12ce4	call	test&set	// test&set lock
	0x12ce8	mov	0xff, %ol	
	0x12cec	cmp	%o0, 0xff	
BCT 2:	0x12cf0	be	0x12cb0	// fail test&set, retry TTS
	0x12cf4	cmp	%l1, 0x0	// delay slot
	0x12cf8	ret		
	0x12cfc	restore		
	. . . (never here) . . .			
test&set:	0x134f8	ret		
	0x134fc	ldstub	[%o0], %o0	// atomic load-store 0xff



```
I0:  r1 = 1
I1:  mem[1000] = 0
I2:  r1 = 0
I3:  branch I0 if cond1
     ...
I4:  branch I0 if cond2
```

Fig. 2. Operation of two-entry SDT and two-entry RUB. (a) Assume that I3 is not taken and I4 is taken. Thus, I4 is inserted into the SDT after it commits. (b) After I0 commits, r1 and its old value 0 are inserted into the RUB. The bit vector is set to 01 to indicate that the RUB entry corresponds to SDT entry 0. Assume that I1 is a silent store, so it has no effect. (c) I2 changes r1 back to its original value 0, so its RUB entry is deleted. (d) Assume that I3 is taken this time. Set `SDT_top` = (`SDT_top`-1+`SDT_size`) mod `SDT_size`, and insert I3 at the top of the SDT. (e) I0 commits again. Insert r1 and its old value 0 into the RUB. The bit vector is 11, indicating that the RUB entry corresponds to SDT entries 0 and 1.

3.4 Spin Detection Hardware

In this section, we describe in detail the components of our hardware:

- *A 16-entry SDT*: Each 33-bit entry contains a PC field, corresponding to a potential control flow cycle, and a `RUB_overflow` (`O`) bit (see below). Whenever the processor commits a non-silent store, it clears the SDT by setting both `SDT_top` and `SDT_bottom` to invalid.
- *A 64-entry RUB*: Each 57-bit entry contains four fields: register number (`RegNum`), value of the register (`RegVal`), valid (`V`) bit, and SDT bit vector. For each SDT entry, the RUB maintains the corresponding observable register state of the running thread. Our experiments show that a 64-entry RUB provides the same results as an unbounded RUB for our workloads.

The total size of these structures is less than 1 KB. Spin detection hardware is after the Commit stage of the pipeline and thus its *latency* is off the critical path and does not directly affect microprocessor performance. To guarantee that spin detection hardware is not a *throughput* bottleneck that requires lengthening the clock cycle, we pipeline it.

In each RUB entry, a one in the ith least significant bit of the SDT bit vector indicates that register `RegNum` corresponds to SDT entry i. For each BCT the processor commits, if its PC matches the PC of an SDT entry, say k, then the processor does a wired-OR over the kth bit of all the bit vectors in the RUB. If the result is one, then there exists at least one entry in the RUB that corresponds to this SDT entry. This means that the observable register state of the thread differs and, thus, the thread was not spinning. If the result is zero, then the observable register state is the same and, thus, the thread was spinning.

To manage RUB space, we use a free map (a 64-bit vector) to track free RUB entries, i.e., entries with zero in the valid (V) bit. Initially, all valid bits are zero. Any insertion or deletion consults the free map. When the processor inserts a new entry into the RUB (according to the conditions in Section 3.2), it initializes the entry's valid bit to one, and the SDT bit vector to contain ones in the bits corresponding to all the valid SDT entries and zeros elsewhere.

When the processor inserts an entry into a full RUB, it locates all SDT entries pointed to by this entry's bit vector. For each of these SDT entries, the processor sets the `RUB_overflow` (`O`) bit in the entry to indicate that the RUB is not large enough to hold its corresponding observable register state. The processor then considers (conservatively) that the observable register state for these SDT entries has changed and will not detect spinning for them. In this case, all of the registers maintained in the RUB for these SDT entries will no longer be useful. Thus, the processor clears the bits pointing to these entries from the bit vector of each RUB entry. If a bit vector becomes all-zero after the clear, the processor adds its RUB entry to the free map. Similarly, when the processor evicts an entry from the SDT, it clears the entry's corresponding bit in the bit vectors of all RUB entries. If a bit vector becomes all-zero, the processor frees the corresponding RUB entry.

The spin detection hardware resides in the commit stage of the processor pipeline. Since spin detection is not latency-sensitive, the RUB pipelines its accesses to the register file via a single dedicated read port. The RUB hardware is similar to a processor issue queue in that it performs associative searches for matching registers. Since our hardware is small and off the processor critical path, it has no impact on processor cycle times.

3.5 Example Operation

Fig. 2 illustrates the operation of a two-entry SDT and two-entry RUB for a nested spin loop. Initially, the SDT and RUB are empty and all registers are zero. The processor starts at

TABLE 3
Target System Parameters

Processor	clock frequency	2 GHz
	reorder buffer	128 entries
	pipeline width	4
	pipeline stages	11
	direct branch predictor	YAGS, 1024-entry choice PHT, 256-entry direction caches
	indirect branch predictor	cascaded, 64-entry leaky BTB filter and second stage table
	integer registers	160 (logical) + 64 (rename)
	floating-point registers	64 (logical) + 128 (rename)
Memory System	L1 cache (I and D)	128 KB, 4-way set associative
	L2 cache	4 MB, 4-way set associative
	memory	2 GB, 64-byte blocks
	interconnection network	broadcast tree, 6.4 GB/sec link

instruction I0 and follows steps in Figs. 2a, 2b, 2c, 2d, and 2e. Finally, when I3 commits again, both the SDT and RUB are the same as in Fig. 2d. The processor finds that SDT entry 1 has the same PC as I3 and no entry in the RUB corresponds to it, thus concluding that the thread has been spinning since the last dynamic instance of I3.

4 Methodology and Evaluation of Spin Detection

4.1 Methodology

To evaluate spin detection and its applications, we simulate a symmetric multiprocessor as our target multithreaded system. We use a heavily modified version of the GEMS simulation infrastructure [14]. Our simulator is based on Simics/sun4u 1.4.4 [15], which models the SPARC V9 architecture in sufficient detail to boot unmodified Solaris 8. Each node in our system consists of a processor, two levels of cache, some portion of the shared memory, and a network interface. We model processor timing using TFSim [16], and we use a detailed memory hierarchy timing simulator that models a MOSI broadcast snooping cache coherence protocol and an interconnection network composed of a hierarchy of switches. Table 3 shows the parameters of our target processor and memory system.

We use SPEC OMP v3.0 [7] and SPLASH-2 [9] benchmarks and two workloads from the Wisconsin Commercial Workload Suite [17] to evaluate spin detection and its applications. For the SPEC OMP benchmarks, we use the medium versions and compile them using Sun Studio 9 with flags suggested by SPEC for Sun systems. We omit three benchmarks: `Galgel`, `Mgrid`, and `Wupwise`, because they take too much time to run on our simulator. Table 4a shows our settings for the OMP benchmarks. We use the training inputs, as opposed to the reference inputs, such that the simulations finish within a reasonable amount of time. The central part of most OMP benchmarks is a major loop whose body contains code executed by multiple threads in parallel (one iteration of the loop often contains a significant amount of computation). We set the number of OpenMP threads equal to the number of processors in the simulated system. For every benchmark except `Art` and `Equake`, we warm up the system for one iteration and then run one more iteration for detailed timing simulation. The code of `Art` does not have a major loop. Thus, we consider its entire program as one "iteration" and run it for a 36×36 windowed image until completion. For `Equake`, we run it for two iterations, since it contains relatively little computation in one iteration of its major loop compared to the other benchmarks.

Table 4b shows the problem sizes that we use for the SPLASH-2 benchmarks. For each benchmark, we first run it with the warmup problem size to prime the system, and then run it again with the simulation problem size to perform detailed timing simulation. To avoid measuring thread forking, our timing simulation starts at the beginning of the parallel phase and continues until the benchmark finishes. Table 5 describes our two commercial workloads.

TABLE 4
Configurations of the Scientific Workloads

Benchmark	Warmup	Simulation
AMMP	1 iteration	1 iteration
Applu	1 iteration	1 iteration
Apsi	1 iteration	1 iteration
Art	none	36×36 image
Equake	1 iteration	2 iterations
Fma3d	1 iteration	1 iteration
Gafort	1 iteration	1 iteration
Swim	1 iteration	1 iteration

(a)

Benchmark	Warmup	Simulation
Barnes	512 particles	16K particles
FMM	512 particles	16K particles
Ocean	66×66 grid	258×258 grid
Radiosity	room	room
Raytrace	car	car
Volrend	head-scaleddown4	head
Water-Nsquared	512 molecules	512 molecules
Water-Spatial	512 molecules	512 molecules

(b)

(a) SPEC OMP settings and (b) SPLASH-2 problem sizes.

TABLE 5
Configurations of the Commercial Workloads

Java server: SPECjbb2000 is a Java benchmark that models a 3-tier system with driver threads. We use Sun's HotSpot 1.4.0 Server JVM. The number of threads and warehouses is 1.5 times the number of processors. We warm up for 100,000 transactions and run for 50,000.
Static web server: We use Apache 2.0.39 for SPARC/Solaris 8, configured to use pthread locks and minimal logging. We use SURGE to generate web requests. Our experiments use a repository of 20,000 files (totally about 500 MB) and 10 simulated users per processor. For a system of N processors, we warm up for $100{,}000 \times N$ requests and run for 5,000.

4.2 Evaluation

4.2.1 Microbenchmarks

We first evaluate our spin detector with two microbenchmarks to show that it can accurately detect spinning in both lock-intensive applications and the operating system kernel. The first microbenchmark, called `contended lock`, implements N threads in an N-processor system. All threads vie for a single spin lock, which provides mutual exclusion for a counter that each thread tries to increment until it reaches 100,000. The second microbenchmark, called `null`, models five million cycles of an idle system in which only the operating system runs and no user threads exist. We run both microbenchmarks on a system of N processors, where N equals 1, 2, 4, 8, and 16. In Fig. 3, we plot the total and useful (i.e., nonspinning) instructions committed. The results show that we detect significant amounts of spinning. As N increases, the total instructions increase, but the useful instructions stay constant. In `contended lock`, only one thread holds the lock at a time and all others spin until the lock is free. In `null`, spinning comes from the Solaris idle thread, which executes a surprisingly complex nested loop to test for runnable threads. The useful instructions in `null` are so few that their corresponding bars are hardly seen in the figure. These two microbenchmarks demonstrate that our spin detector can accurately detect both simple spinning due to spin locks and complex flag spinning that involves hundreds of instructions in one spin loop iteration.

4.2.2 Scientific and Commercial Workloads

In Fig. 4, we show the total number of committed instructions versus useful (nonspinning) instructions for a subset of the SPEC OMP and SPLASH-2 benchmarks and the two commercial workloads. For SPEC OMP and SPLASH-2, we choose two benchmarks: one that has the least fraction of spinning (left figure) and one that has the most (right figure) among all the benchmarks in the same suite. The spinning in these benchmarks is due to spin locks, flags, and barriers. We see that the number of total instructions increases as the number of processors increases, while the number of useful instructions stays almost constant; the same is also true for the benchmarks that are not shown. This demonstrates that our spin detector accurately detects spinning instructions.

The commercial workloads do not have much spinning at the user level. As we see in Fig. 4, the number of useful instructions does not stay constant as the number of processors increases for the two commercial workloads. This is because these workloads make significant use of the OS. The amount of work that the OS performs changes as the number of processors varies (so does the number of threads in the system), thus causing the number of useful instructions to change. Nevertheless, the differences between the total and useful instructions in Fig. 4 indicate that our spin detector detects spinning. Further inspection of our data shows that most of this spinning comes from flags and spin locks in the Solaris kernel dispatcher.

The evaluation in this section has demonstrated the accuracy of our spin detection hardware. As we showed in Section 2.2, our hardware design releases the burden that software approaches place on programmers. Since it requires no program modification, our hardware can support both legacy and proprietary software. In the next three sections, we show how to apply our hardware to improving performance and resource management of multithreaded systems.

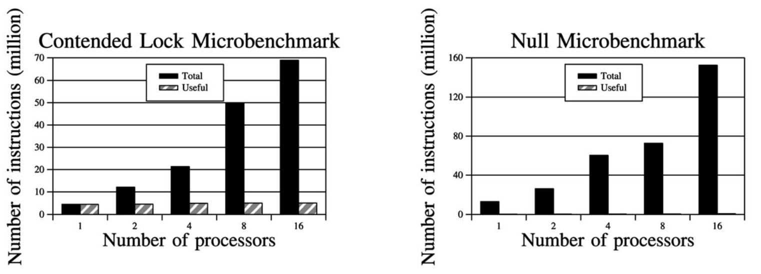


Fig. 3. Total versus useful instructions for the microbenchmarks.

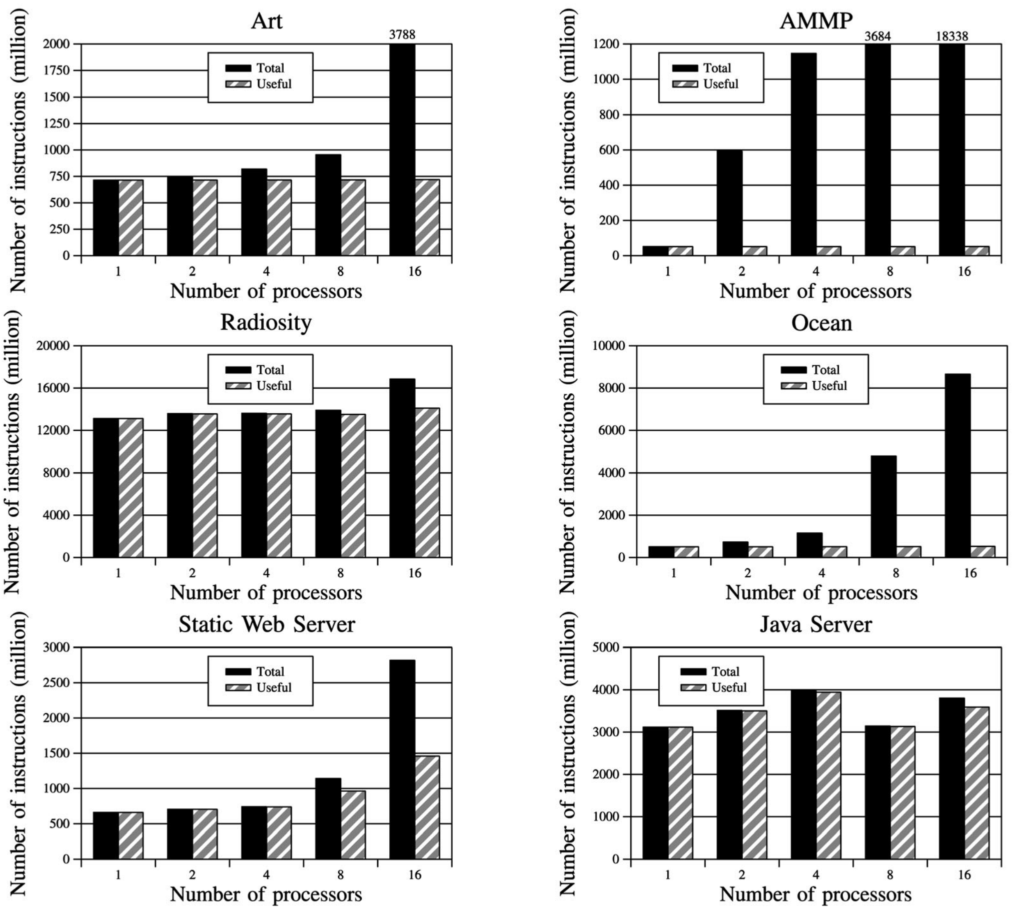


Fig. 4. Total versus useful instructions for the scientific and commercial workloads.

5 Scheduling and Power Management

We now show how spin detection helps enable efficient OS scheduling and power management.

5.1 Scheduling

We extend our spin detector to implement a *spin-then-yield* synchronization mechanism to improve thread scheduling. When the hardware detects that a thread has spun for N iterations (i.e., the processor has committed the same BCT N times), where N is a tunable parameter, it raises an interrupt to the processor. The OS services this interrupt by invoking a handler routine, which blocks the spinning thread and moves it to the end of the run queue, thus allowing another thread to execute. The spinning thread resumes execution when it returns to the head of the run queue and the OS dispatcher selects it to run again.

The spin-then-yield mechanism is similar to the software-level two-phase waiting algorithm in which the thread spins for a certain number of iterations and then invokes the `yield` system call to relinquish the processor; the difference is that our approach can automatically convert spinning to two-phase waiting, thus simplifying the job of programmers.

Since we do not have access to the Solaris source code, we cannot directly implement the OS support for spin-then-yield in our simulator. Instead, we emulate it as follows.

When the hardware detects that a thread has spun for N iterations, we force the thread to jump to a routine, called `force_to_yield`, by setting the program counter of the thread's processor to the start of this routine. Within `force_to_yield`, the thread first saves its current PC, and then invokes the `yield` system call, thus blocking itself. When the `yield` system call returns (i.e., the blocked thread returns to the head of the run queue and the OS dispatcher selects it to run again), the thread jumps back to the saved PC, thus resuming the interrupted spin loop. The spin detection hardware also resets its counter for the spin loop iterations. In our implementation, we hardwire the `force_to_yield` routine to start at virtual address 0x10000 and span only a few bytes. This address range is unused by all Solaris programs and, thus, it does not affect the code or data of any program.

We evaluate the performance effects of spin-then-yield on a 2-processor system running simultaneously two SPEC OMP benchmarks, `Art` and `Gafort`. Each benchmark uses two OpenMP threads. We vary the number of iterations N that a thread is allowed to spin before blocking from 5,000 to 25,000 with increments of 5,000. For each value of N, we run the workload until both benchmarks finish one iteration of their major loop (see Section 4), and measure performance as the reciprocal of the execution time. In Fig. 5, for each value of N, we plot the system performance normalized to the performance of a system that does not implement spin-then-yield, i.e., N is infinity. We see that spin-then-yield

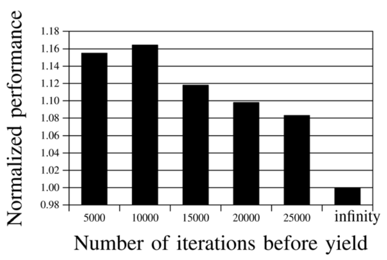


Fig. 5. Results for spin-then-yield.

greatly improves performance. The system obtains the highest performance when N is 10,000 (16.4 percent speedup compared to not using spin-then-yield); beyond 10,000, the benefits of spin-then-yield drop gradually.

We can design similar techniques to improve scheduling in virtual machine systems without modifying guest operating systems. When our hardware detects spinning, the VMM can choose to deschedule the spinning virtual CPU that is waiting on a synchronization object (e.g., a lock) and select another virtual CPU within that VM to run. When the preempted virtual CPU is rescheduled at a future time, it may then successfully acquire the lock in contention if it has been released by another virtual CPU in the interim.

5.2 Power Management

An extreme form of scheduling is to turn off power when a processor detects that a thread is spinning. Alternatively, we can perform dynamic voltage scaling: the processor can switch to a low power mode when a thread is spinning, and switch back to the normal mode when the thread stops spinning. This scheme achieves similar effects to Intel's HALT instruction [6] and ARM's wait/signal instruction pair [18]; the difference is that it requires no software modification.

To support detecting when a thread stops spinning, we assume a shared-memory multiprocessor system with an invalidation-based cache coherence protocol; however, our design can be easily adapted to other types of system. A thread can exit a spin loop only if it observes a change to its observable state, and this is possible only if some other thread or I/O device modifies a cache block accessed within the spin loop. To track cache blocks accessed within a spin loop, we use an eight-entry *load address cache* (LAC). If a processor detects that its current thread has spun for one iteration (i.e., it commits a BCT the second time) and if the BCT is at the top of the SDT, the processor sets a `try_suspend` bit to one, and then starts recording the block addresses of subsequent loads into the LAC. If the processor detects a second iteration of the spin loop, it switches to a low power mode and resets the `try_suspend` bit. The cache controller then probes the LAC for every invalidation it receives. On a match, the spinning thread would potentially stop spinning. Thus, the processor switches back to the normal power mode and clears the LAC. The LAC could overflow before the processor switched to the low power mode. If so, any invalidation will cause the processor to switch back to its normal power mode. The processor also switches back to the normal power node and clears the LAC on every interrupt. The operation of the LAC is similar to the load-locked store-conditional primitives in Alpha processors that provide support for atomic instruction sequences [19].

A subtle complication may arise when a thread spins for only a short period of time. Due to deep pipelines and out-of-order execution, the processor can receive an invalidation that will cause its thread to stop spinning well before it switches to the low power mode. Thus, after the cache controller starts probing the LAC, it will never see the relevant invalidation. Therefore, the processor may never switch back to the normal power mode.

To solve this problem, we make the following additions to our design. For each load that the processor issues, it records the issue time in the load's reorder buffer entry (or load queue entry depending on the architecture). The processor also updates a timestamp when it receives any invalidation from another processor. When a load commits, the processor compares the load issue time and the timestamp, and sets a `recv_invalidate` bit to one if the load was issued earlier. Upon reaching the BCT for the third time, the processor switches to the low power mode only if `recv_invalidate` is zero. Otherwise, it clears the LAC and `recv_invalidate`, and stays in the normal power mode. Note that this implementation is conservative in that any invalidation can prevent the processor from switching to the low power mode. A more precise implementation would monitor only the addresses observable from within the spin loop. However, our simple implementation is sufficient to serve as a proof of concept.

Since our simulator does not yet have a power model, we evaluate our design by computing the fraction of time that processors can stay in the low power mode. For a 16-processor system, we compute the *total system time* by aggregating the total time that a workload spends on each processor, and the *total suspension time* by aggregating the time that each processor stays in the low power mode during the entire execution of the workload. Due to space limitations, we show results for a representative subset of our workloads. We obtain the total suspension time divided by the total system time as follows: `Fma3d` 52 percent, `Water-Spatial` 39 percent, static Web server 57 percent, and Java server 13 percent. These results indicate that, by detecting spinning, we could achieve significant power savings during the executions of these workloads.

There are two sources of overhead for our power management scheme. First, our power management hardware consumes a small but nonzero amount of power (i.e., much less than the power consumed by a spinning processor). Thus, it is beneficial only if we can detect a significant amount of spinning. Second, switching power modes introduces extra overhead (e.g., time) and can be beneficial only if the overhead is relatively small. Thus, we experimented to see whether the time that a processor stays in the low (or normal) power mode is long enough to outweigh a reasonable overhead of switching the power mode. We refer to the period during which a processor is in the normal mode as a *run*, and the period during which it is in the low power mode as a *gap*. In Fig. 6, we show the distributions of the run and gap lengths for each workload. For a given time interval on the x-axis, the y-axis shows the fraction of runs (or gaps) whose length is within that

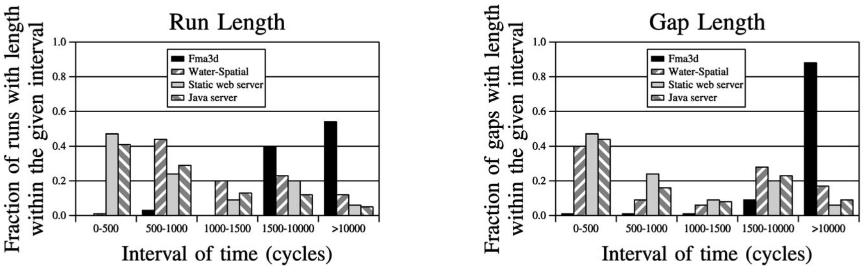


Fig. 6. Distribution of run and gap lengths.

interval. We observe that the workloads all have a significant portion (> 50 percent) of their runs and gaps longer than 500 cycles, which indicates that they can tolerate the overhead of sophisticated power management techniques.

6 Livelock Detection

In this section, we discuss how we extend our spin detector to support livelock detection. From the hardware's perspective, a program is *livelocked* if the program indefinitely executes instructions, but makes no forward progress in its computation. Our livelock definition encompasses certain situations that software may otherwise consider to be deadlock. For example, when multiple threads of a program wait on each other in a circular fashion, we consider the program livelocked, instead of deadlocked, because it continues to execute instructions.

6.1 Architectural Support for Livelock Detection

Recently, Li et al. [20] proposed Pulse, an operating system mechanism that dynamically detects deadlock (or livelock in our terminology). An assumption of Pulse is that all threads involved in a deadlock are blocked. The design of our spin detector motivates us to extend Pulse to detect deadlocks involving spinning threads. In this paper, we focus on livelocks/deadlocks that consist of only spinning threads and no blocked threads.

Intuitively, if all threads of a program are simultaneously spinning, then the program is livelocked. Nevertheless, a program could have a subset of its threads livelocked while the rest still moves forward. Thus, detecting livelock involves two parts: The OS specifies a group of threads of interest, and the hardware detects if each thread in the group is spinning. Simultaneous spinning of all threads, however, is not sufficient for claiming livelock; the OS also needs to guarantee that no thread outside the group nor any future I/O event could cause a spinning thread to stop spinning. Such guarantees may not be easy to obtain in general, but may be reasonable in some situations. For example, lock-induced livelocks often have well-defined thread groups competing for locks. A thread can stop spinning only if another thread in the same group releases a lock; no thread outside the group or I/O can affect the lock on which the thread spins.

Using the spin detector, our hardware checks if a group of threads spin simultaneously. Assume for now that the number of threads in the group (T) is less than or equal to the number of processors (P) in the system, and all T threads run concurrently on different processors. The problem with detecting simultaneous spinning at any *physical time* is the presence of in-flight invalidation requests that could cause some threads to stop spinning at a later time. To avoid this problem, we implement *logical time* by leveraging our invalidation-based broadcast snooping cache coherence protocol. For every processor that runs a thread in the group, we initialize its logical time (a 64-bit integer) to zero. Whenever a processor observes a cache invalidation request, it increments its logical time by one. Using this logical time design ensures that no in-flight invalidation requests exist when a group of threads are spinning simultaneously.

To support detection of simultaneous spinning, we extend each SDT entry with a time field. When a processor inserts a new entry into its SDT, it sets the entry's time to the processor's current logical time. When the processor detects a spin loop, the corresponding SDT entry's time and the current logical time together form the interval in which the thread was spinning. A processor sends this interval to the OS (e.g., via an interrupt) or a system service controller (like the one in Sun's Starfire [21]) whenever it detects spinning. The OS or the controller keeps for each thread the most recent interval it receives and checks periodically if all the T threads' intervals intersect. If so, they have been spinning simultaneously.

We now consider the case that T is greater than P. The challenge is that the T threads never all run at the same logical time and, thus, their spin intervals may never intersect.[3] To overcome this, we include the SDT and RUB as part of a thread's state that the OS saves and restores at a context switch. If the OS switches a thread out while it is spinning and switches it back in later, based on the restored SDT and RUB, the processor can determine that the thread has been spinning since the last time it was running until now (i.e., the spin interval includes the period during which the thread is not running). Thus, if the T threads are livelocked, eventually the system can detect that they are spinning simultaneously.

6.2 Evaluation

Since livelock does not occur in our workloads, we create a microbenchmark, called `circular lock`, to evaluate our livelock detection mechanism. The microbenchmark implements T threads, each of which holds a lock (Pthreads spin lock) while waiting for a lock that another thread holds, and all the locks form a circular dependence chain. We evaluate our mechanism using Simics/x86 1.6.11 running Linux kernel 2.6.3. We add a system call to Linux, which allows a user program to specify a group of threads (PIDs) for

3. In a multiprogrammed system, this can also be true even when $T \leq P$.

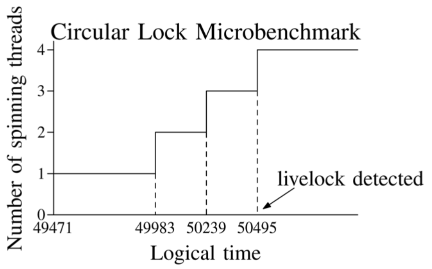


Fig. 7. Results for livelock detection.

livelock monitoring. We modify Linux's context switch code to save and restore a thread's SDT and RUB.

We run `circular lock` on a simulated 4-processor system with T varying from two to sixteen. Our results show that we can detect livelock for all of these cases. In Fig. 7, we plot the number of spinning threads in the system as a function of logical time for $T = 4$. We see that the number of spinning threads that our hardware detects increases as logical time progresses and, finally, the hardware detects livelock when all threads spin simultaneously at logical time 50,495. Moreover, by running `contended lock` (see Section 4.2), we verify that our mechanism does not generate false alarms of livelock when it does not exist.

We also evaluate the context switch overhead introduced by saving and restoring the SDT and RUB. Since we do not yet have a detailed timing model for the Simics/x86 simulator, we measure the number of dynamic instructions, instead of time, for each context switch. We measure from the start of the `switch_to` function in the Linux scheduler to the execution of the first instruction of the next process that the scheduler selects to run. Over 1,000 context switches, an average Linux context switch executes 98 instructions, while spin detection only increases it to 116 instructions.

7 Accurate Hardware Performance Counters

In this section, we extend our spin detector to design accurate hardware performance counters.

7.1 Useful IPC

Both hardware and software often rely on performance counters to make dynamic decisions. For example, to save power, the processor can dynamically resize its issue queue based on online calculation of IPC [22]. IPC is a widely used performance metric for single-threaded uniprocessors, as well as multithreaded systems (SMT and multiprocessors) [23], [24], [25], [26]. However, spinning can lead to arbitrarily inflated IPC numbers that mismatch actual system performance. For example, a processor spinning on a lock can achieve high IPC while doing no useful work. To accurately measure the performance of multithreaded systems, we propose *useful IPC* (uIPC), which counts only useful (non-spinning) instructions that processors commit per cycle.

We extend our spin detector to provide a new performance counter, called *PerfCtr-SpinInstrs*, which counts the number of spinning instructions committed by the processor. To compute uIPC, we subtract *PerfCtr-SpinInstrs* from the total number of instructions and divide it by the number of cycles. To count spinning instructions, we add a counter to each SDT entry. When the processor inserts a new entry into the SDT, it initializes the entry's counter to zero. When an instruction commits, all SDT entries increment their counters by one. When a processor detects spinning for an SDT entry, it adds that counter value to *PerfCtr-SpinInstrs*. To avoid double-counting of spins because of nested loops, when the processor detects spinning for an SDT entry, it subtracts the counter in the entry from all entries below it in the SDT. Finally, it resets the entry's counter to zero and pops all entries above it.

7.2 Evaluation

We now evaluate uIPC and verify that it tracks performance more accurately than IPC. The goal of our evaluation is not to make conclusions about the performance of any system or application, but rather to show how IPC can be misleading in the presence of spinning and how uIPC better tracks performance. Due to space limitations, we only show results for a representative subset of the workloads: `Fma3d`, `Water-Spatial`, static Web server, and Java server.

In Fig. 8, we plot performance (reciprocal of runtime), IPC, and uIPC as a function of the number of processors in the system for our workloads. All metrics are normalized to their 1-processor values. IPC and uIPC are sums of these quantities across all processors.

For `Fma3d` and `Water-Spatial`, our results show that uIPC is linearly correlated with performance, while IPC is not. When the number of processors increases, IPC shows a drastic increase, completely disproportionate to the improvement of performance. Thus, if we use performance counters that do not distinguish spinning instructions, we would falsely assume that performance improves greatly even though the actual improvement is only moderate.

For static Web server and Java server, we see that IPC and uIPC are almost the same from one to eight processors. This is because these workloads use mostly blocking synchronization in the user-level code and do not have much spinning. However, for 16 processors, IPC is higher than uIPC because much more spinning occurs in the Solaris kernel when the processor count increases to 16 (as we have seen in Fig. 4).

For static Web server, uIPC is not linear with performance. Moreover, from 8 to 16 processors, uIPC increases while actual performance drops. Our results show that when the number of processors increases from 4 to 8 and 16, this workload executes significantly more useful instructions in the OS kernel. This causes uIPC to increase even when performance decreases. The increased kernel instructions come from both the `httpd` threads of the Apache Web server and the `sched` process (PID 0) of the Solaris kernel. As the number of processors increases, the number of `httpd` threads in the system also increases, which causes both `httpd` and `sched` to execute more instructions in the kernel dispatcher.

To isolate the performance of `httpd`, in Fig. 9a, we replot our results by counting only `httpd` instructions. We make three observations from this analysis. First, IPC and uIPC are almost equal in all the systems. This indicates that most spinning instructions come from the `sched` process. Second, uIPC now increases (or decreases) as performance increases (or decreases). Thus, the `sched` process was the major cause for the uIPC-performance mismatch we

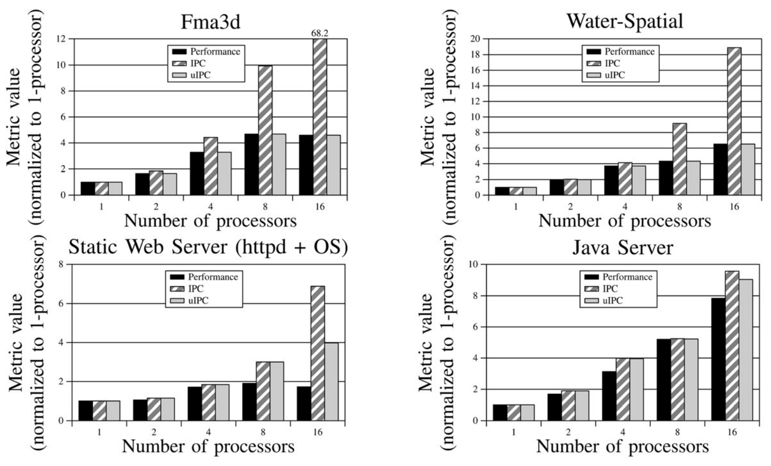


Fig. 8. Useful IPC results.

observed earlier. Finally, uIPC is, however, still not linearly correlated with performance. We hypothesized that this was due to the more kernel instructions that `httpd` executes in the 8-processor and 16-processor systems than in the other systems. To verify our hypothesis, in Fig. 9b, we replot the results for only the user-level instructions in `httpd`. We see that both uIPC and IPC now achieve nearly linear correlation with performance. In practice, performance analysts can do similar analysis as above. Whether to aggregate the performance counters of different threads depends on which aspects of the system (e.g., OS or application) the analysts want to study.

For the Java server workload, uIPC reflects performance directly, but not linearly. The nonlinear correlation is due to the increased number of kernel instructions as the number of processors increases, similar to what we see in the static Web server workload.

8 Related Work

We present related work on hardware synchronization support, livelock detection, and multithreaded system performance analysis.

8.1 Hardware Synchronization Support

Tullsen et al. [27] proposed hardware blocking locks to dynamically suspend threads that would otherwise spin. Keckler et al. [28] used register full/empty bits to enable threads waiting on shared data to stall rather than spin. McDowell et al. [29] proposed hardware-lock-then-block for synchronization, which is similar to our spin-then-yield mechanism. Intel IA-32 provides the HALT and PAUSE instructions to save power and improve spin loop performance [6]. ARM provides a wait/signal instruction pair to save power for spin locks [18]. The IBM POWER5 allows software to set lower scheduling priorities for spinning threads [30]. The Thrifty Barrier [31] allows software to choose a low power mode when entering a spin lock barrier. To benefit from these designs, legacy code requires recoding and recompilation, which places a burden on programmers and introduces costs that could outweigh their benefits [32]. In contrast, our mechanisms require no modification to applications.

8.2 Livelock Detection

Conventionally, people prove liveness properties for multithreaded programs using static techniques such as temporal logic [33], model checking [34], and reachability analysis [35]. Recent work by Engler and Ashcraft [36] described a static tool to detect deadlocks in large multithreaded systems. Dynamically, indirect approaches are often used to infer potential livelock situations. For example, discarding packets due to queue overflow could imply livelock [37]. Compared to the indirect approaches, our hardware support can provide more accurate information about livelock without the need of knowing program semantics. Recently, Li et al. [20] proposed Pulse, a dynamic mechanism that can detect deadlocks involving only blocked

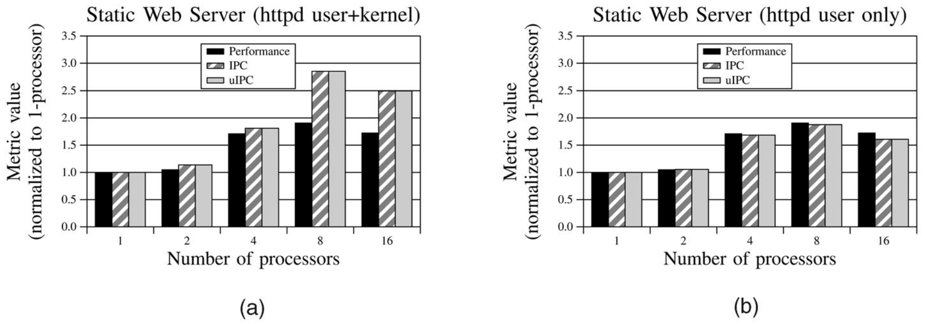


Fig. 9. Useful IPC results for the `httpd` process. (a) User and kernel instructions of `httpd`. (b) User-level instructions only of `httpd`.

threads. Our hardware support complements Pulse by enabling detection of deadlocks involving spinning threads.

8.3 Multithreaded System Performance Analysis

Redstone [32] showed the importance of studying spinning on SMT and evaluated spinning in the OS kernel. Nayfeh et al. [38] discussed that synchronization can cause problems for IPC when used to evaluate multiprocessor performance, but found that IPC tracked performance in their experiments because their workloads did not have much spinning. Similar to us, Yamauchi et al. [26] defined effective IPC and used it to evaluate a single-chip multiprocessor. However, they did not describe how they identified spinning instructions. Recently, Lepak et al. [39] studied how to redeem IPC as a valid metric for multithreaded systems. However, they did not consider the negative impact of spinning on the accuracy of IPC. To analyze spin lock performance, many tools use source instrumentation [40], [41] or execution traces [42] to identify spins, or simple hardware counters to estimate spinning costs [43], whereas our hardware spin detector releases the burden on programmers and supports unmodified software.

9 Conclusions

Spinning occurs extensively in multithreaded applications and operating systems. Detecting spinning is important for identifying performance bottlenecks and program bugs. In this paper, we defined general conditions for identifying spinning. Based on these conditions, we developed efficient hardware that can detect spinning in unmodified programs. We showed how to extend our hardware to improve scheduling and power management by not allocating resources to spinning threads or virtual machines. To facilitate debugging, we applied our hardware to support dynamic detection of livelock in multithreaded programs. To improve performance monitoring, we developed performance counters that accurately reflect system performance by factoring out spinning instructions. Using full-system simulation with SPEC OMP, SPLASH-2, and Wisconsin commercial workloads, we demonstrated that our mechanisms can effectively improve the management of multithreaded systems.

Acknowledgments

The authors thank the anonymous reviewers for their comments and suggestions on the early draft of this paper. They thank David Becker and Jaidev Patwardhan for their help with the experiments. This work is supported in part by the US National Science Foundation (CCR- 0312561, EIA-9972879, CCR-0204367, CCR-0208920, and CCR-0309164), Intel, IBM, Microsoft, and the Duke University Graduate School. Daniel J. Sorin is supported by a Warren Faculty Scholarship.

References

[1] 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," *Proc. 20th ACM Symp. Operating System Principles,* pp. 164-177, Oct. 2003.

[2] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C. Martins, A.V. Anderson, S.M. Bennett, A. Kägi, F.H. Leung, and L. Smith, "Intel Virtualization Technology," *Computer,* vol. 38, no. 5, pp. 48-56, May 2005.

[3] C.A. Waldspurger, "Memory Resource Management in VMware ESX Server," *Proc. Fifth Symp. Operating Systems Design and Implementation,* pp. 181-194, Dec. 2002.

[4] A. Whitaker, M. Shaw, and S.D. Gribble, "Scale and Performance in the Denali Isolation Kernel," *Proc. Fifth Symp. Operating Systems Design and Implementation,* pp. 195-210, Dec. 2002.

[5] VMware, Inc., "High CPU Utilization of Inactive Virtual Machines," VMWare Knowledge Base, Answer ID 1077, http://www.vmware.com/support/kb/enduser/std_adp.php?p_faqid=1077, 2006.

[6] Intel, "Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor,"Intel Corp., Order Number: 248674-002, May 2001.

[7] SPEC, "SPEC OpenMP Benchmark Suite V3. 0," http://www.spec.org/omp, Dec. 2003.

[8] A. Whitaker, S.D. Gribble, and M. Shaw, "Rethinking the Design of Virtual Machine Monitors," *Computer,* vol. 38, no. 5, pp. 57-62, May 2005.

[9] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 24-37, June 1995.

[10] E. Artiaga, N. Navarro, X. Martorell, and Y. Becerra, "Implementing PARMACS Macros for Shared Memory Multiprocessor Environments," Technical Report UPC-DAC-1997-07, Dept. of Computer Architecture, Polytechnic Univ. of Catalunya, Jan. 1997.

[11] A.R. Karlin, K. Li, M.S. Manasse, and S. Owicki, "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor," *Proc. 13th ACM Symp. Operating System Principles,* pp. 41-55, Oct. 1991.

[12] B.-H. Lim and A. Agarwal, "Waiting Algorithms for Synchronization in Large-Scale Multiprocessors," *ACM Trans. Computer Systems,* vol. 11, no. 3, pp. 253-294, Aug. 1993.

[13] K.M. Lepak and M.H. Lipasti, "Silent Stores for Free," *Proc. 33rd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* pp. 22-31, Dec. 2000.

[14] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood, "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News,* vol. 33, no. 4, pp. 92-99, Sept. 2005.

[15] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer,* vol. 35, no. 2, pp. 50-58, Feb. 2002.

[16] C.J. Mauer, M.D. Hill, and D.A. Wood, "Full System Timing-First Simulation," *Proc. 2002 ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems,* pp. 108-116, June 2002.

[17] A.R. Alameldeen, M.M.K. Martin, C.J. Mauer, K.E. Moore, M. Xu, M.D. Hill, D.A. Wood, and D.J. Sorin, "Simulating a $2M Commercial Server on a $2K PC," *Computer,* vol. 36, no. 2, pp. 50-57, Feb. 2003.

[18] J. Goodacre and A.N. Sloss, "Parallelism and the Arm Instruction Set Architecture," *Computer,* vol. 38, no. 7, pp. 42-50, July 2005.

[19] Compaq, *Alpha 21264 Microprocessor Hardware Reference Manual,* Compaq Computer Corp., July 1999.

[20] T. Li, C.S. Ellis, A.R. Lebeck, and D.J. Sorin, "Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution," *Proc. 2005 USENIX Ann. Technical Conf.,* pp. 31-44, Apr. 2005.

[21] A. Charlesworth, "Starfire: Extending the SMP Envelope," *IEEE Micro,* vol. 18, no. 1, pp. 39-49, Jan./Feb. 1998.

[22] D. Folegnani and A. González, "Energy-Effective Issue Logic," *Proc. 28th Ann. Int'l Symp. Computer Architecture,* pp. 230-239, July 2001.

[23] J.L. Lo, L.A. Barroso, S.J. Eggers, K. Gharachorloo, H.M. Levy, and S.S. Parekh, "An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors," *Proc. 25th Ann. Int'l Symp. Computer Architecture,* pp. 39-50, June 1998.

[24] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang, "The Case for a Single-Chip Multiprocessor," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* Oct. 1996.

[25] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* pp. 191-202, May 1996.

[26] T. Yamauchi, L. Hammond, K. Olukotun, and K. Arimoto, "A Single Chip Multiprocessor Integrated with High Density DRAM," *IEICE Trans. Electronics,* vol. E82-C, no. 8, pp. 1567-1577, Aug. 1999.

[27] D.M. Tullsen, J.L. Lo, S.J. Eggers, and H.M. Levy, "Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor," *Proc. Fifth IEEE Symp. High-Performance Computer Architecture,* pp. 54-58, Jan. 1999.

[28] S.W. Keckler, W.J. Dally, D. Maskit, N.P. Carter, A. Chang, and W.S. Lee, "Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor," *Proc. 25th Ann. Int'l Symp. Computer Architecture,* pp. 306-317, June 1998.

[29] L.K. McDowell, S.J. Eggers, and S.D. Gribble, "Improving Server Software Support for Simultaneous Multithreaded Processors," *Proc. Ninth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP),* pp. 37-48, June 2003.

[30] R. Kalla, B. Sinharoy, and J.M. Tendler, "IBM POWER5 Chip: A Dual-Core Multithreaded Processor," *IEEE Micro,* vol. 24, no. 2, pp. 40-47, Mar./Apr. 2004.

[31] J. Li, J.F. Martinez, and M.C. Huang, "The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors," *Proc. 10th IEEE Symp. High-Performance Computer Architecture,* Feb. 2004.

[32] J.M. Redstone, "An Analysis of Software Interface Issues for SMT Processors," PhD dissertation, Univ. of Washington, Dec. 2002.

[33] S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM Trans. Programming Languages and Systems,* vol. 4, no. 3, pp. 455-495, July 1982.

[34] G.J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Eng.,* vol. 23, no. 5, pp. 279-295, May 1997.

[35] S.C. Cheung and J. Kramer, "Context Constraints for Compositional Reachability Analysis," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 4, pp. 334-377, Oct. 1996.

[36] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlock," *Proc. 20th ACM Symp. Operating System Principles,* pp. 237-252, Oct. 2003.

[37] J.C. Mogul and K.K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-Driven Kernel," *ACM Trans. Computer Systems,* vol. 15, no. 3, pp. 217-252, Aug. 1997.

[38] B.A. Nayfeh, L. Hammond, and K. Olukotun, "Evaluation of Design Alternatives for a Multiprocessor Microprocessor," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* pp. 67-77, May 1996.

[39] K.M. Lepak, H.W. Cain, and M.H. Lipasti, "Redeeming IPC as a Performance Metric for Multithreaded Programs," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* Sept. 2003.

[40] R. Bryant and J. Hawkes, "Lockmeter: Highly Informative Instrumentation for Spin Locks in the Linux Kernel," *Proc. Fourth Ann. Linux Showcase & Conf.,* pp. 271-282, Oct. 2000.

[41] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tools," *Computer,* vol. 28, no. 11, pp. 37-46, Nov. 1995.

[42] R.W. Wisniewski and B. Rosenburg, "Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems," *Proc. 2003 ACM/IEEE Conf. Supercomputing,* pp. 3-16, Nov. 2003.

[43] Y. Solihin, V. Lam, and J. Torrellas, "Scal-Tool: Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors," *Proc. 1999 ACM/IEEE Conf. Supercomputing,* Nov. 1999.

Tong Li received the BS degree from Northwestern Polytechnical University, China, the MS degree from the University of Kentucky, and the PhD degree from Duke University, all in computer science. He joined Intel Labs in 2005 after completing his PhD. His research interests are in scalable processor architectures, operating systems, applications, and their interactions.

Alvin R. Lebeck received the BS degree in electrical and computer engineering, and the MS and PhD degrees in computer science from the University of Wisconsin—Madison. He is an associate professor of computer science and of electrical and computer engineering at Duke University. His research interests include architectures for emerging nanotechnologies, high-performance microarchitectures, hardware and software techniques for improved memory hierarchy performance, multiprocessor systems, and energy efficient computing. He received the best paper award at the 31st IEEE/ACM International Symposium on Microarchitecture. He is the recipient of a 1997 US National Science Foundation (NSF) CAREER Award, has received funding from the NSF, DARPA, Intel, Compaq, Microsoft, IBM, and is a member of ACM and a senior member of the IEEE and the IEEE Computer Society.

Daniel J. Sorin received the BSE degree in electrical and computer engineering from Duke University and the MS and PhD degrees in electrical and computer engineering from the University of Wisconsin—Madison. He is an assistant professor of electrical and computer engineering and of computer science at Duke University. His research interests include highly available computer architecture, multithreaded memory system design, and the architectural impact of emerging technologies. He is the recipient of a US National Science Foundation (NSF) CAREER Award and a Duke Warren Faculty Scholarship. He is a member of the IEEE and the IEEE Computer Society.