View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Decoupling Contention Management from Scheduling

F. Ryan Johnson
Ecole Polytechnique Fédérale de Lausanne (VD, Switzerland)
Carnegie Mellon University
Pittsburgh, PA, USA
ryanjohn@ece.cmu.edu

Radu Stoica
Ecole Polytechnique Fédérale de Lausanne (VD, Switzerland)
radu.stoica@epfl.ch

Anastasia Ailamaki
Ecole Polytechnique Fédérale de Lausanne (VD, Switzerland)
Carnegie Mellon University
Pittsburgh, PA, USA
natassa@epfl.ch

Todd C. Mowry
Carnegie Mellon University
Pittsburgh, PA, USA
tcm@cs.cmu.edu

Abstract

Many parallel applications exhibit unpredictable communication between threads, leading to contention for shared objects. The choice of contention management strategy impacts strongly the performance and scalability of these applications: spinning provides maximum performance but wastes significant processor resources, while blocking-based approaches conserve processor resources but introduce high overheads on the critical path of computation. Under situations of high or changing load, the operating system complicates matters further with arbitrary scheduling decisions which often preempt lock holders, leading to long serialization delays until the preempted thread resumes execution.

We observe that contention management is orthogonal to the problems of scheduling and load management and propose to decouple them so each may be solved independently and effectively. To this end, we propose a load control mechanism which manages the number of active threads in the system separately from any contention which may exist. By isolating contention management from damaging interactions with the OS scheduler, we combine the efficiency of spinning with the robustness of blocking. The proposed load control mechanism results in stable, high performance for both lightly and heavily loaded systems, requires no special privileges or modifications at the OS level, and can be implemented as a library which benefits existing code.

Categories and Subject Descriptors D.4.1 [**Operating Systems**]: Process Management – Synchronization, multiprocessing/multiprogramming/multitasking, scheduling; D.4.8 [**Operating Systems**]: Performance – Measurements; H.2.4 [**Database Management**]: Systems – Concurrency

General Terms Performance, Measurement

Keywords Concurrency control; load management; contention; spinning; blocking; scheduling; thread; multicore.

1. Introduction

The rise of multicore architectures has several important implications for parallel software. Stagnant single-thread performance favors multicore architectures, which in turn pressure software developers to parallelize all applications, not just ones which lend themselves easily to parallel execution. Further, exponentially growing core counts require increasingly fine-grained synchronization to minimize scalability-limiting contention among threads. Enforcing synchronization among threads imposes non-trivial overheads on the critical path, and a poor implementation can both aggravate existing serialization points and create new, artificial, ones. Software utilizing fine-grained parallelism is especially vulnerable because acquiring a mutex lock (even when uncontended) can take as long as executing the short critical section it protects. Further, even "well-behaved" applications, achieving over 99% parallel execution, face enough contention to limit their scalability on today's hardware. The net result is that application performance is increasingly sensitive to the implementations of synchronization primitives and contention management policies.

Synchronization primitives generally resolve contention by either busy waiting (spinning) until the desired resource becomes free, by descheduling the waiting thread (blocking), or through some combination of the two. Spinning is attractive because waiting threads respond to lock handoffs very quickly, minimizing the dead time where no thread has access to the resource. Unfortunately, polling is highly resource-intensive. Techniques such as exponential backoff [1] and queue-based primitives [24] ease pressure on caches and memory systems, but by design every type of spinning wastes CPU time and waiting threads compete with others in the system. In particular, spinning tends to produce a kind of priority inversion where spinning threads prevent the lock holder they wait on from running and releasing the lock; the problem becomes especially prominent in systems under high load and can limit scalability even in software that is otherwise well-behaved. In contrast, blocking removes waiting threads from the system, minimizing wasted resources and potentially allowing other threads to do useful work. However, blocked threads must be rescheduled once the lock is available, adding significant delays to the critical path and introducing a scheduling bottleneck which also limits scalability.

The weaknesses of spinning and blocking are well-known and have prompted many proposals which combine spinning with blocking [27], but hybrid implementations face the further challenge of balancing competing objectives: removing threads from the system while preserving fast response times. The optimal balance between spinning and blocking varies increasingly, and performance drops steadily, as core and thread counts grow [6]. Figure 1 illustrates the challenges that face the current state-of-the art synchronization primitives. We plot the performance of a database transaction processing workload running on a 64-thread Niagara II system (see Section 4 for details). Load, defined as the number of runnable threads in the system, varies from 1 to 192 threads along the x-axis and the y-axis plots the corresponding throughput in thousands of transactions per second. We plot results for two versions of the application: one uses a sophisticated and scalable spinlock implementation [15], while the other uses the state-of-the-art spin-then-block Solaris pthread mutex (modern operating systems avoid purely blocking primitives). The software is over 99% parallel and throughput should ideally increase with parallelism until the machine is fully loaded at 64 threads, then

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'10 March 13–17, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-839-1/10/03…$10.00.

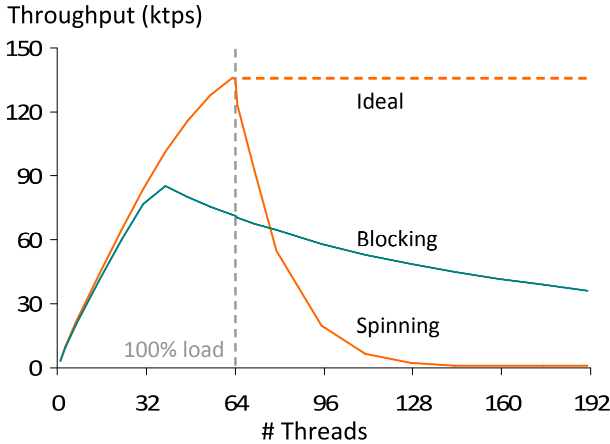

Figure 1. Weaknesses in state-of-the-art synchronization primitives which use blocking and spinning.

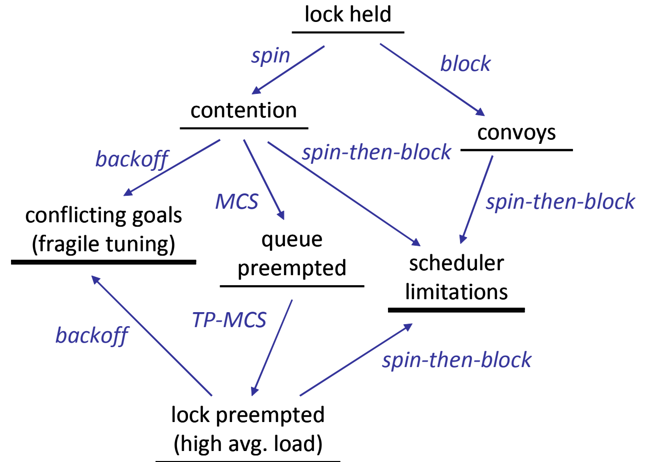

Figure 2. Problem space for contention management policies

level off but remain steady as load continues to increase. Instead, as load increases, blocking synchronization overwhelms the OS scheduler, causing poor performance after 32 threads. Spinning fares better at first, but as load crosses 100% its performance also drops off drastically due to priority inversions.

Nearly all of the challenges which arise with either spinning or blocking are due to scheduling concerns. Spinning gives optimal performance under light load (when no scheduling is needed), but performs poorly under high load because preempted lock holders trigger priority inversions. Similarly, blocking synchronization performs badly because it potentially causes a context switch with every lock handoff. Frequent context switching leads to scheduling bottlenecks and adds significant overhead to the critical path. Fine-grained synchronization aggravates the problem because it favors frequent, short critical sections — much shorter than a context switch — over longer and more coarse-grained ones.

We argue that the solution to the spinning-blocking trade-off lies not with a more effective hybrid scheme, but in decoupling load control from contention management. Effective contention management uses spinning for fast lock hand-offs and does not block in response to contention. Effective load control then prevents spinning threads from causing overload while keeping load low enough that lock holders are not preempted. We propose a mechanism which achieves both goals by notifying a random subset of spinning threads to block in response to overload, waking them when load drops or after a timeout of roughly one scheduler time slice. Spinning threads are attractive targets because they cannot make forward progress by definition, so removing them does not hurt performance in the short term. Further, removing some spinning threads from a loaded system ensures that lock holders responsible for the wait are able to run, while leaving enough other spinning threads to preserve fast lock handoffs. Finally, OS time slicing operates normally in the absence of contention, though load control remains active to disrupt any convoys which might arise.

In summary, this paper makes three main contributions:

1. We show that scheduler activity on the critical path of lock handoffs underlies performance problems with the current state-of-the-art in both spinning and blocking primitives.
2. We propose to decouple contention management from scheduling, which moves the OS scheduler completely off the critical path and allows applications to exploit the best properties of spinning and blocking instead of merely trading them off.
3. We design and implement a load control mechanism which achieves the proposed decoupling without modifications to the OS kernel or scheduler. For a variety of benchmarks, we achieve peak performance for lightly loaded machines, while retaining 85% of that peak even with 200% load (two runnable threads per hardware context).

The rest of the paper is organized as follows. The next section expands on the evolution of synchronization algorithms, and related issues such as scheduling and preemption resistance. Section 3 introduces our proposed load control mechanism and discusses implementation issues. Sections 4-6 present and evaluate the load control implementation. Section 7 compares load control with alternative approaches, followed by conclusions in Section 8.

2. Managing Load and Contention

This section examines different approaches related to conflict resolution for locking primitives (see Section 7 for a discussion of alternatives to locking). We focus on locking because it is a general-purpose and widely-utilized approach to synchronization. Conflict resolution is necessary because threads which encounter contention must wait for the lock to be released. As mentioned, there are two fundamental contention management approaches — spinning and blocking — as well as variants which extend and combine the two to mitigate their various weaknesses. Figure 2 illustrates the space of challenges encountered in implementing locking primitives and how solutions for these evolved; each underlined text block is a challenge and connecting arrows are existing solutions which attempt to overcome the challenge.

Under blocking schemes (grouped toward the right of Figure 2), threads are descheduled in response to contention. Blocking has the primary benefit of freeing the CPU until the waiting thread can make progress again. As an added advantage, the scheduler can cooperate with blocking synchronization, for example by descheduling threads which wait for a preempted lock. Blocking is an expensive operation, however, because it requires two context switches (with corresponding OS scheduling decisions), adding 10-15µs to the critical path of the system. A longer critical path increases the likelihood that other threads will encounter contention and block, forming a vicious cycle of extremely slow lock handoffs known as a *convoy* [5]. Because convoys are so damaging to scalability, purely blocking contention management is only used in uniprocessor systems where spinning leads to deadlock.

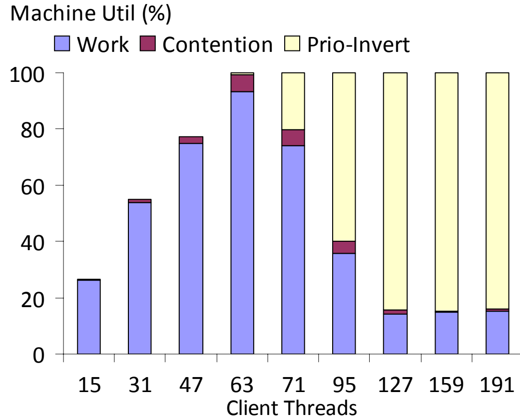

Figure 3. Spinning: priority inversion

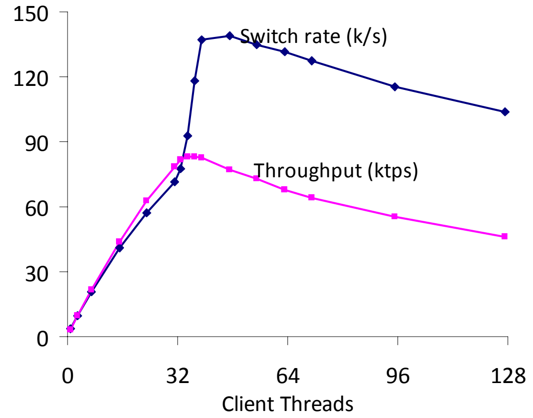

Figure 4. Blocking: scheduler overload

In contrast to blocking, spinning or "busy waiting" schemes (grouped on the left side of Figure 2) leave waiting threads on CPU as they poll a memory location for changes that indicate a lock handoff. Pure spinning is highly responsive (1-2 cache miss delays per handoff) and avoids context switching or system calls on the critical path. However, it also wastes CPU time other threads might have been able to use. In addition, naive spinlock implementations create heavy traffic in the memory system and thus interfere with computation. Finally, the OS scheduler cannot distinguish between threads which spin and those which make forward progress, leading to situations where a lock holder gets preempted, only to have the new thread waste its time slice spinning.

To show how severe the problem of preempted lock holders can be, we run a database telecommunication benchmark (TM-1) on a 64-context machine (see Section 4 for details), using a state-of-the-art spinlock. We instrument the code to differentiate between spinning due to true contention and spinning due to priority inversion. Figure 3 shows the resulting breakdown of work. We vary the number of threads along the x-axis, and measure CPU time spent doing useful work, spinning due to contention, and spinning due to priority inversion. For fewer than 64 active threads, machine utilization is less than 100% and contention is low. However, as soon as utilization passes 100% priority inversions quickly dominate, wasting up to 85% of CPU time. It is important to note that true contention is not the concern: at peak performance, less than 10% of CPU time is wasted spinning on contended locks, and that fraction drops rapidly when the OS scheduler preempts lock holders.

2.1 Preemption-resistant Spinlocks

Queue-based spinlocks [22][24] and to a lesser extent, ticket locks [29], provide excellent scalability because waiting threads form a FIFO queue and each lock handoff targets a specific thread ("MCS" in Figure 2). Queue-based locks also give each thread its own memory location to spin on, eliminating unnecessary coherence traffic. Further, the orderly handoff is an elegant solution for the "thundering herd" problem, where all waiting threads race for the lock at each release and cause both contention and memory traffic. However, the same FIFO ordering makes such algorithms especially vulnerable to preemptions because *every* thread in the queue is effectively a lock holder: A thread preempted from the queue will almost certainly become the lock holder before it wakes again, and other threads cannot bypass it even if it was preempted before acquiring the lock. As a result, load must remain strictly below 100% in order to avoid convoys.

Time-published MCS locks [15] ("TP-MCS" in Figure 2) allow lock holders to remove preempted threads from the lock queue instead of passing the lock to them. By only handing the lock to running threads, time-published locks eliminate the main weakness of queue-based spinlocks while retaining their superior scalability. However, even with TP-MCS locks, a few extra threads in a 32-processor system add 50-100% to the execution time of some SPLASH-2 benchmarks [15]. This behavior arises because time-published locks only protect the queue, leaving lock holders vulnerable to preemption (the results in Figure 3 are based on TP-MCS). Preempted lock holders impact all locks which do not cooperate with the OS scheduler, and are the focus of this work.

2.2 Backoff and Spin-then-block Hybrids

Many approaches exist to ameliorate some of the weaknesses of spinning and blocking. Backoff-based spinning provides another solution to the "thundering herd" problem by limiting the number of waiting threads which can respond simultaneously. Test-and-test-and-set with exponential backoff [1] and spin-then-yield variants [14][27], fall into this category, with the latter removing threads from the CPU completely. Backoff schemes suffer from a fundamental weakness, however, in that they impose competing objectives: Long backoffs are best for reducing wasted resources, but shorter backoffs give the fastest response to lock handoffs. The best tuning for backoff-based schemes does not necessarily perform well (see next subsection), and tuning for the general case is challenging because the hardware, OS, application, and the number of active threads all influence the optimal balance [6].

Hybrid spin-then-block schemes [6][27] improve on backoff by allowing the lock holder to explicitly wake waiting threads. The capability to both sleep and wake threads allows threads to block without timeouts, without the risk of leaving a contended lock idle. Where spin-then-yield schemes are essentially spinlocks which use the scheduler as a form of backoff, hybrid spin-then-block schemes use spinning to reduce context switching imposed by a blocking primitive. However, as with backoff, hybrid schemes can cause undesirable side effects on load (see below). Heavyweight OS mutex implementations usually employ spin-then-block strategies, including the Solaris adaptive mutex [23] and the Linux futex [12].

The Solaris adaptive mutex is an advanced spin-then-block design that minimizes the need for context switching under low and moderate contention, and which switches to blocking under high contention. However, as presented in Figure 1, its behavior still leaves much room for improvement. To identify the reason behind the lock's poor performance we modify the TM-1 bench-

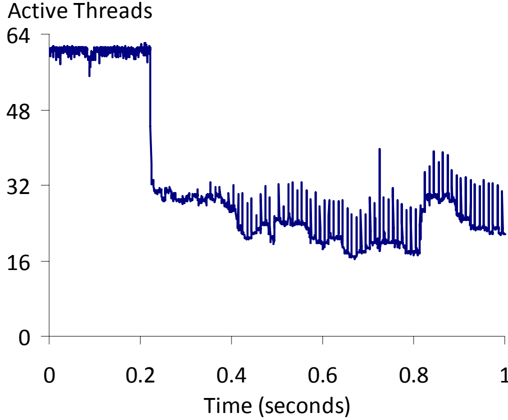

Figure 5. Blocking backoff: variability

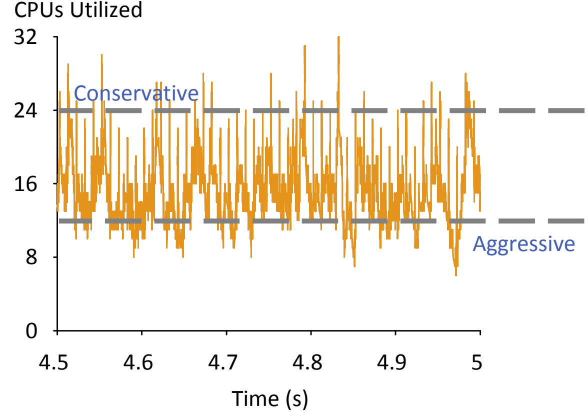

Figure 6. Workload variability at short time scales

mark to use the Solaris mutex instead of spinning, then measure both throughput and context switching rates of the system as load increases. Figure 4 shows that, for fewer than around 37 client threads on the x-axis, threads do not block; there is only one context switch per transaction due to the database log commit, which requires a disk I/O. After 37 threads, however, the adaptive mutex begins to break down as more and more threads exhaust their patience for spinning and block. Soon, all waiting threads have blocked and every lock handoff requires a context switch. This both increases significantly the critical path length and saturates the OS scheduler, leading to a steady performance drop as contention rises.

2.3 Load-triggered Backoff

We proposed recently a form of contention management [19] that combines aspects of the backoff and spin-then-block strategies. In the proposed approach the system monitors load and spinning threads sleep for an exponentially distributed amount of time whenever an overload situation is detected. The scheme partly decouples load and contention management in that threads never block if load remains below 100%; it matches the performance of spinning under low load while retaining the robustness of blocking under overload situations. While load-triggered backoff is effective in the sense that it achieves the best of both worlds, its performance approaches that of a blocking mutex for load even slightly over 100%, leaving significant room for improvement.

We now present further analysis which shows that load-triggered backoff suffers from a one-sided control mechanism. Once a thread has gone to sleep it cannot be woken until it times out. The fundamental weakness of backoff is especially clear as it causes both dips in load (because sleeping threads did not wake soon enough) and spikes (when the OS wakes groups of sleeping threads simultaneously at scheduler clock ticks). The net effect is that load variability increases significantly and yet, on a loaded system, contended lock acquires are still associated with context switches. Figure 5 captures this behavior: we set artificially the load target at 32 contexts, then enable the backoff scheme during execution of the TM-1 benchmark with 63 client threads. The number of running threads is shown on the y-axis as time progresses along the x-axis. Though the baseline benchmark includes significant I/O and other sources of context switches, its behavior is relatively uniform. Once the backoff scheme becomes active, however, the number of active threads begins to fluctuate wildly, with large variations around the 32 context target even at the relatively long time scale shown. The unpredictability takes two forms: the long-scale average tends to wander around low values, while frequent spikes push load much higher for short intervals. The behavior shown reflects a careful tuning of the backoff distribution. Poor tuning leads to complete unpredictability, with utilization swinging between 0 and 100% at millisecond time scales.

In this work we develop further the idea behind load-triggered backoff to produce a scheme which retains the performance of spinning even under overload situations, to the extent that performance is better in a somewhat overloaded system than a nearly-loaded one. Further, where load-triggered backoff required some cooperation from the application, the new scheme is completely transparent to both application and OS.

2.4 Variability and Admission Control

Many important applications, including server and database workloads, exhibit irregular parallelism and scheduling behavior as threads block and unblock due to changes in demand, contention, and I/O requests. In response to this challenge, admission control strategies [3][8][25] monitor key statistics (CPU, memory, request response times, even lock contention) which they use to tune the amount of work allowed into the system. Such monitoring is important to keep system load within reasonable limits over macroscopic time scales. On the other hand, unpredictable and irregular thread behavior over short time scales leads to large variations in load about the average maintained by admission control. These variations occur because, at any moment, many threads are asleep waiting for resources or I/O completions. Threads wake at unpredictable times, often in groups, and trigger load spikes that can lead to preemptions if the average is high. When a load spike triggers a lock holder preemption, threads which would have normally acquired the lock and finished their computation will instead spin and prolong the spike's duration. With more threads in the system than processors, the effect reinforces itself and quickly overloads the machine. Admission control is too coarse-grained to prevent such priority inversion and bad scheduling decisions directly, and it can only mitigate the effects indirectly by reducing the number of threads it allows to enter the system. However, this solution tends to leave the machine underutilized most of the time.

Figure 6 shows how the number of runnable threads in the system varies over half a second of the TPC-C database benchmark with 32 client threads running on a machine with 64 hardware contexts (see Section 4 for details). This setup allows us to see the number of threads which can run at any moment, given enough processing power. We used the Solaris DTrace utility [7] to record every context switch during the measurement interval, giving an accurate view of variations in instantaneous load. Most of the time between 12 and 24 threads are active, with the average near 16.

The dotted lines mark two possible extremes in admission control policies. For a machine with 12 processors, allowing 32 client threads to run as shown is quite aggressive in that the machine would be 100% utilized nearly all the time (bottom line). In a machine with 24 processors, on the other hand, allowing only 32 client threads is quite conservative and the machine would be lightly loaded except during especially tall spikes (top line).

Our load control scheme is designed to manage variations in load over millisecond time scales and relies on admission control to maintain a reasonable long-term average. We assume an aggressive control policy that maintains high but not overwhelming load.

3. Effective Load Control

The previous section highlights the fact that spinning and blocking are best suited to very different tasks: Spinning is good for achieving quick lock hand-offs, while blocking works best at longer time scales to reduce competition among threads for processor cycles. Complications arise when either is used for other purposes. We therefore propose to decouple load control, with its associated scheduling overheads, from contention management, which must occur on the critical path and favors spinning. By applying spinning and blocking where each is most effective, we achieve highly responsive lock hand-offs while keeping load low enough to avoid the OS preemptions that cause convoys and priority inversion. The proposed mechanism has several desirable features:

- It discourages the OS scheduler from preempting lock holders (causing priority inversions) and encourages rapid rescheduling when an unwanted preemption does occur.
- Scheduling and other high-overhead operations occur off the critical path in order to avoid creating or aggravating bottlenecks in the system.
- It has a stabilizing influence on the system by counteracting quickly variations in load which arise. When more threads become active in the system, the mechanism compensates by removing spinning threads that are wasting CPU time.
- Decisions are based on a view of the entire process rather than local information available to any one mutex lock.

The last point marks an important difference between our approach and the state-of-art lock implementations (e.g. Solaris mutex, Linux futex): load control acts on all locks as a group rather than making local decisions the way a traditional blocking primitive would. This is a key feature as we are no longer limited by static trade-offs between spinning and blocking. The most contended locks in the system will still tend to donate the most threads as de-scheduling victims (as they cause the most spinning), but contention will not cause load to drop excessively or context switch rates to increase drastically. Bottleneck locks also see reduced contention as a beneficial side effect.

3.1 Load Controller Design Overview

Our load control mechanism consists of two parts: an application visible spinlock and a dedicated control daemon thread (called from now on the *controller*) that maintains system statistics and decides when to (de)schedule threads. The controller and user threads communicate through the *sleep slot buffer,* a shared data structure that captures scheduling decisions taken by both controller and user threads.

3.1.1 Controller implementation

Figure 7 gives a more detailed overview of the mechanism's components, where the controller (left) and application threads (right) communicate via the shared sleep slot buffer (center). The controller uses existing OS services such as high-resolution timers to wake periodically (out of phase with the OS scheduler tick). The controller uses OS process accounting facilities to monitor a single "sensor" value: *overload*, defined as the number of excess runnable threads in the system (negative in underload situations). The controller makes scheduling decisions based on the intuitive policy that threads should be removed from CPU in response to overload, and re-scheduled when load drops.

The heart of the load control mechanism is the *sleep slot buffer*, which serves a dual purpose. The controller uses it to notify spinning threads how many should block in response to overload. Threads also register themselves in the sleep slot buffer before blocking so the controller can identify and wake them in response to underload. The controller adapts to changes in load by adjusting the size of the sleep slot buffer. Spinning threads check the buffer for empty slots, which they "claim" by placing their thread ID inside before blocking. Whenever the buffer grows (or threads time out and leave), any spinning threads present will quickly claim the empty space and block, reducing load. When the buffer shrinks, the controller wakes any threads which no longer need to sleep, allowing them to re-enter the system immediately (in contrast to load-triggered backoff, where they must time out before waking). To avoid races the controller uses atomic instruction to clear slots of threads that need to wake up, and each thread checks whether the controller cleared its slot before going to sleep.

3.1.2 User side thread load control mechanism

The user-visible part of load control is an augmented spinlock. Our experiments extend the TP-MCS lock, though any high-performance spinlock which allows threads to abort attempts would also work. Once a thread joins the lock queue, it checks for space in the sleep slot buffer while polling for a lock handoff; if it finds available space it uses an atomic compare-and-swap (CAS) to place its TID at the buffer's head. CAS failure indicates that another thread arrived first and the thread continues polling. If the CAS succeeds, the thread must sleep and attempts to leave the lock queue. If the thread acquires the lock during this time, it simply clears the sleep slot it claimed and enters the critical section. Otherwise, it blocks as long as the sleep slot remains set to its own TID (the controller may clear it before the thread blocks even once) or until 100ms pass, whichever comes first. Once the thread wakes it restarts the lock acquire process as if it just arrived.

It is important to note that the controller responds to load, not to individual preemptions, which is key to fast response. In an overloaded system with no contention, the controller maintains a suitable sleep target, but most or all sleep slots remain empty because there are no spinning threads to claim them. However, when untimely preemptions trigger convoys the resulting spinning threads immediately claim sleep slots and block. The drop in load allows the OS to reschedule the preempted lock holder and break the cycle, all without specific intervention by the controller thread.

3.2 Implementation and Performance Issues

This subsection presents some of the low-level issues which arise when implementing the load control mechanism. For example, load control, which is implemented entirely in user space, depends on existing operating system facilities to work. In addition, because the sleep slot array is a central point of communication, it must be accessed carefully to avoid lengthening the critical path of the computation or causing excessive memory traffic.

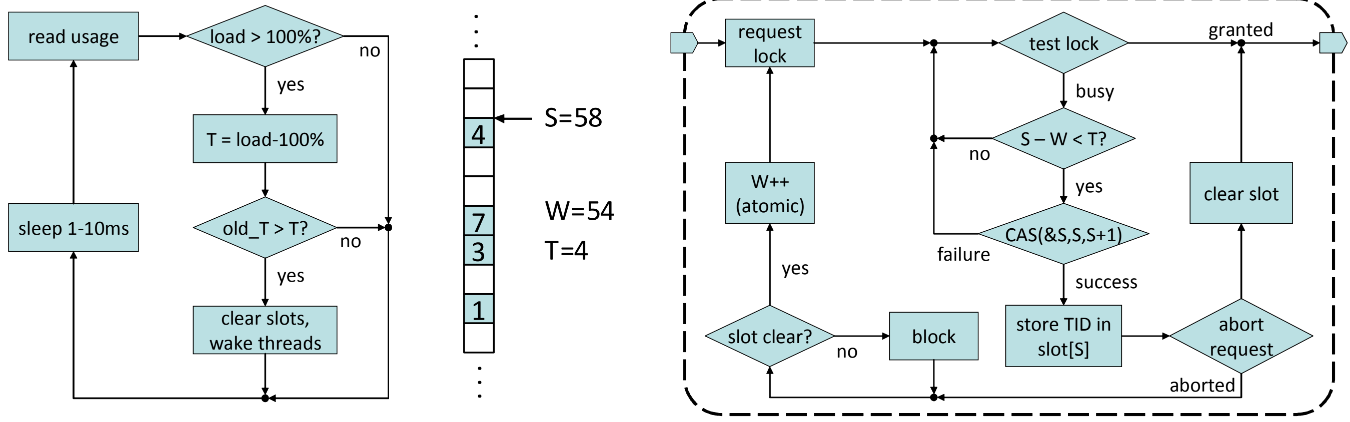

Figure 7. Load controller overview: daemon algorithm (left), sleep slot buffer (center), client thread algorithm (right). S is the number of threads which have ever slept due to load control, and W is the number of such threads which have since woken and left; T is the number of sleeping threads load control targets target number of threads which should sleep. In the above example no additional threads will sleep.

3.2.1 OS support required for load control

Load control depends on three OS facilities: the ability to *(a)* schedule periodic daemon thread wakeups independent from the system clock tick (to avoid measurement and scheduling anomalies), *(b)* measure load accurately and with high resolution (~100µs), and *(c)* deschedule and wake threads efficiently.

High-resolution timers are an effective way to wake the controller thread at scheduler-independent intervals, and are widely available as POSIX real-time extensions. To measure load, we use the Solaris microstate accounting statistics, which have nanosecond precision and include time spent on CPU and waiting in runqueues. The statistics can be used to derive the number of CPUs which the process would have used had they been available. Microstate accounting has also been prototyped in at least two Linux variants (Gelato and Carrier Grade Linux). It is also possible to use dynamic tracing facilities such as DTrace or SystemTap (the Linux equivalent to DTrace) to emulate microstate accounting, though admittedly the emulation approach is cumbersome and imposes high "probe effect" overhead (at least for the DTrace version we tested).

In order to deschedule and wake threads efficiently we use lightweight system call primitives available in different forms in both Linux and Solaris. In Linux, the `futex` [12] system call adds the target thread to a sleep queue associated with an arbitrary address and forms the basis of all user-space blocking primitives. Solaris provides a `lwp_park` syscall that removes the calling thread from the OS scheduler until a `lwp_unpark` call from another thread reinstates it. Unlike `futex`, however, `lwp_park` is libc-private and requires some effort to make available to application code. Because a real world implementation of load control would likely reside in libc, we opt to use `lwp_park` for our experiments. However, with some effort we were also able to emulate the syscall using per-thread `pthread_mutex` and `pthread_cond` pairs.

3.2.2 Implementing an effective sleep slot buffer

The sleep slot buffer has several important requirements: many threads must be able to access it concurrently without causing undue contention, spinning threads must be able to find available slots efficiently, and there should be no races that leave threads asleep during underload. The first requirement is somewhat easier than it sounds because threads only access the buffer when there is significant contention elsewhere already, and because the number of threads which actually claim slots is quite limited (some hundreds per second at most). Further, priority inversions are not a problem because threads accessing the buffer are already attempting to sleep and do not hold the lock.

To keep fast response time in the common case where no sleep slots are available and to avoid excessive memory traffic, threads must be able to determine quickly whether there is space in the sleep buffer. We achieve this goal using a circular buffer implemented over a large array. We maintain two counters associated with the buffer: the number of threads which have ever slept (*S*), and the number which have awoken and left (*W*). The controller also maintains a sleep target (*T*), the number of threads which should sleep. *S* serves as the head pointer of the buffer (modulo its physical size), where threads join. There is no explicit tail pointer because threads may leave in any order and the buffer usually contains gaps. Instead, threads compute $S\text{-}W < T$ when deciding whether to sleep, and the controller reclaims space by scanning from the last-known-end during each cycle as it searches for threads to wake.

Figure 7 (center) shows an example of the sleep buffer where *S=58* threads have ever slept and *W=54* threads have since woken and left. The four sleeping threads which remain are spread over seven slots. Because the sleep target is *T=4*, the sleep buffer is currently full and no new threads will attempt to sleep. If T increases or a thread wakes and leaves, the next thread to join the buffer will insert itself at the arrow (above TID 4).

3.2.3 Augmenting spinloops unobtrusively

Though the sleep slot buffer and sleep target are efficient and fairly straightforward to implement, in the common case where contention is minor and there are no open slots in the buffer, the extra overhead still slows down lock handoffs. For an in-order architecture such as the Niagara II, the delay is especially painful because it triggers cache misses that the hardware cannot hide, reducing peak performance by about 10% on our benchmarks even when load was less than 100%. One solution is to spin for some period of time before checking the sleep target. However, this imposes the trade-off that comes with other back-off schemes: if a thread checks load control infrequently enough to not impact its response to lock handoff, it does not respond as well to changing sleep targets (and vice-versa). Our solution is to manually unroll the TP-MCS polling loop several times, placing appropriate prefetch

instructions and interleaving load control operations with frequent checks for lock handoff. Though tedious, this approach allows a thread to poll the lock and the sleep buffer at the same time.

4. Experimental Methodology

This section describes our platform and our methodology for obtaining the different measurements reported in this paper. We perform all experiments on a Sun T5220 "Niagara II" server with 64GB RAM, running Solaris 10. The Niagara II has 16 processor pipelines which are shared by a total of 64 hardware contexts. This machine offers more hardware contexts on one chip than any other server platform currently available, giving a glimpse into the future for all platforms as on-chip core counts rise. We implement load control as a library which can be replaced with either time-published spinlocks or pthread_mutex in order to simplify comparisons. We use DTrace [7] to capture information — such as instantaneous load and preemptions of spinlock holders — which would otherwise be difficult to obtain, and measure load with the OS microstate accounting facilities. Microstate accounting uses high-resolution timers to track precisely how much time a process spends in various states: user, system, interrupts, blocked, waiting for a processor, etc. Because it does not depend on sampling, microstate accounting is immune to sampling anomalies.

In this paper, we evaluate three classes of benchmark: a series of microbenchmarks to isolate and examine specific behaviors; Raytrace, a member of the SPLASH-2 benchmark suite [33]; and database transaction processing workloads executing in the open source Shore-MT storage manager [20]. We describe each of these in detail in the following paragraphs.

In order to isolate performance characteristics of the load control mechanism, we employ a microbenchmark modeling a straightforward scenario: M threads running on N hardware contexts acquire and release repeatedly a single global lock. The "critical section" consists of a single call to `gethrtime`, which takes between 40 and 80ns to execute on our machine.[1] Between lock acquires, threads busy-wait for a fixed period of time before trying again. All threads begin executing before the first measurement and stop after the last one. Threads increment a local counter with each lock release, and the benchmark harness computes throughput by comparing two successive readings of each thread's counter while threads continue to run.

To compare with previous work in preemption resistant spinlocks [15] and as an example of the medium-sized applications developers commonly encounter, we evaluate the Raytrace application from the SPLASH-2 suite [33]. Unlike most of the benchmarks in the suite, the load balancing in this application introduces irregular parallelism, meaning it cannot be easily pipelined or partitioned in a way that eliminates all contention. This irregularity makes it a good candidate for load control, especially since contention levels depend on thread counts, not input size. We use the `car.geo` input, rendered at 1280x1024 resolution with an anti-aliasing radius of one pixel. With these parameters and 64 active threads the application is more than 99% parallel.[2] Because runs complete in 5 seconds or less on our machine, we report an average of six runs per data point.

Our final workload, database transaction processing, differs significantly from the traditional parallel benchmarks because it is large and complex (150kLoC), utilizes OS services extensively, and exhibits fine-grained and irregular parallelism. It also represents an important class of commercial applications which are otherwise underrepresented. The database server experiments employ transactions from two benchmarks, both running within Shore-MT:

- TPC-C [31] models an online retailer which receives, processes, and delivers orders made by customers over the Internet, developed by a consortium of database, OS, and hardware vendors. It features a mix of update transactions and queries ranging from very small to fairly large and complex, exhibits heavy application-level contention (from database locks), and generates intense disk I/O.
- TM-1 (a.k.a NDBB and TATP[3]) [26][18] models a cell phone provider database and was originally developed by Nokia to vet the offerings of database vendors. It consists of seven very small transactions, both update and read-only. The application exhibits little logical contention, but the workload generates significant physical contention within the engine itself [19].

We choose these workloads because they exercise the database engine differently. We also note that database workloads are unusual in that they must cope with two forms of contention: transactions serialize data accesses at the logical level (typically by acquiring database locks), and threads must acquire large numbers of mutex locks (often called "latches") to protect the internal data structures of the engine as it executes transactions.

TPC-C experiments use a 100 Warehouse dataset (~13GB) and a 12GB buffer pool. Though the workload is memory-resident on our machine, it still generates a heavy stream of random writes to disk because all updates must eventually be flushed disk for durability. The stream easily overwhelms our 48-disk array, so we place both the database and log files in tmpfs, then use Shore-MT's fake I/O latency setting to force each "disk request" to take at least 6msec. All requests thus proceed in parallel, but with latency as if from a random seek by one of many disk heads in a very large disk array. TM-1 uses a 100,000 subscriber dataset (~200MB), also with all files in tmpfs. However, TM-1 is not I/O-intensive and disk performance does not affect it strongly.

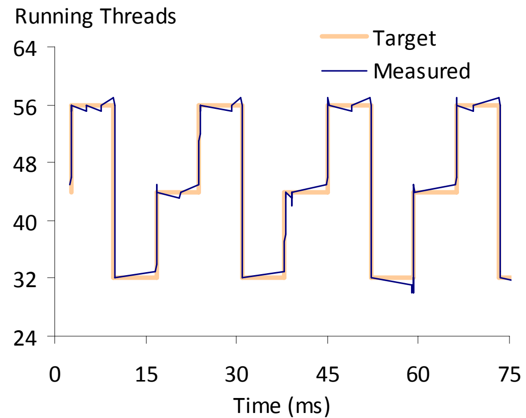

Figure 8. Response to a fixed-timing pattern of control output.

1. We break with the tradition of accessing N cache lines because contended data can be accessed very quickly thanks to a shared L2 cache.
2. Replacing the custom memory allocator in Raytrace with Solaris' threaded malloc eliminates previously-reported scalability problems.
3. TM-1 became NDBB, which in turn became TATP while this paper was under review. The earlier two benchmarks have since disappeared.

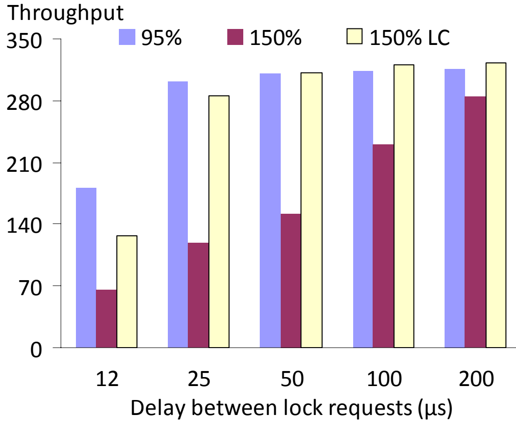

Figure 9. Impact of varying contention for 95% and 150% load.

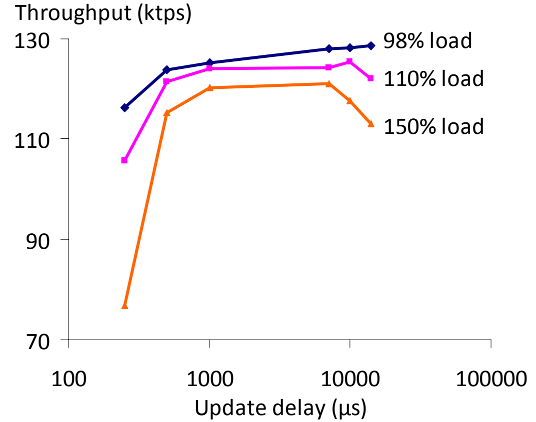

Figure 10. Effect of changing the load controller update interval

5. Evaluating Load Control

In the subsections which follow we analyze the behavior of load control using the microbenchmarks and applications introduced in Section 4. We begin by evaluating the potential of load control, then show its impact on full applications, and discuss special cases.

5.1 Response to Control ("Bump test")

In order for load control to be effective, the system must respond quickly and predictably to changes in the control output. For example, the load-triggered backoff scheme we discussed earlier suffers from unpredictable system response as the sleep target changes. In contrast, Figure 8 illustrates the effectiveness of our proposed approach using a "bump test" which modifies the sleep target of our microbenchmark in a fixed pattern over time. A controllable system will respond to sudden changes in control with a fast and proportional (predictable) change in its steady state behavior. As the figure shows, every change in the sleep target results in an immediate adjustment to the number of active threads. The first thread responds within 30µs of a change and the system has stabilized at the new thread count within 200µs. These results indicate that the control mechanism is sound, assuming we can update the sleep target accurately and often enough (see Section 5.3).

5.2 Effectiveness as Contention Levels Vary

Because load control can only remove spinning threads from the system, the amount of contention in the system impacts the responsiveness of load control. Though low contention leads to little spinning and a small pool of suitable victim threads, load control remains effective for two reasons. First, normal OS scheduling causes very few priority inversions in the absence of contention, and load control has no need for victim threads. In the more common case of locks which are heavily used but not contended, preempting a lock holder triggers priority inversion which produces spinning threads for load control to work with.

Figure 9 demonstrates this effect using a microbenchmark where threads contend for a single global lock, with a fixed delay between requests. High contention occurs for short requests on the left of the x-axis and drops off moving toward the right. We consider three cases, the base case where the machine is 95% loaded (61 threads) as well as 150% loaded machine (96 threads) both with and without load control. As we move right along the x-axis, decreasing contention leads to two effects. First, the 95% loaded system quickly reaches a state of low contention where throughput is determined only by the number of active threads, not the amount of time they spend holding the lock. Second, for the overloaded case, less time spent in critical sections means a lower probability of a preemption catching a lock holder, steadily reducing the performance penalty due to priority inversions. With extremely high contention (the 12µs case), load control is less effective because lock holders are preempted too often. Even though load control responds quickly by removing a spinning thread, the lock holder must still be rescheduled, leading to roughly a 12µs delay on the critical path. Overall, however, we see a significant benefit from load control over a wide range of contention levels.

5.3 Sensitivity to Controller Update Rate

We have already seen that load control has response times in the tens of µs. However, the control output must also be accurate and timely or the system will respond faithfully to unhelpful load targets. For example, if load has dropped back to normal by the time load control registers a load spike, too many threads will sleep and causing underutilization of the machine. On the other hand, updating the load control target requires an expensive syscall, and doing so too often hurts performance. Figure 10 illustrates the trade-off between timeliness and overhead as we execute TM-1 and vary the load control update interval along the x-axis. The y-axis gives system throughput for 98%, 110%, and 150% load (63, 72, and 96 threads, respectively) with load control active. For extremely frequent load measurements, high overhead slows all three cases. The cost increases with load because the Solaris traverses every thread in the process for each load measurement. In the middle region (3-10ms), the benefits of load control outweigh the overheads, except for the 98% load, which sees only the overhead. Finally, as the interval increases past 10ms (the system tick interval), load control makes poor decisions due to stale data. In order to maximize performance for all load factors, we set the update interval to 7msec for our experiments.

5.4 Graceful Degradation Under High Load

The most important measure of load control is its performance as the number of threads in the system increases. Ideally, the extra threads would not change the throughput of the system, through per-thread throughput would necessarily drop. In practice, context switching is not free and preemptions still occur occasionally, leading to a gradual drop in performance as load continues to rise. The goal is therefore to allow performance to degrade gracefully, depending on admission control to keep load from going so high as to exhaust system resources. Figure 11 compares the performance of Raytrace, TM-1, and TPC-C as load varies from near-idle to overloaded (128 threads). Results for each application are clus-

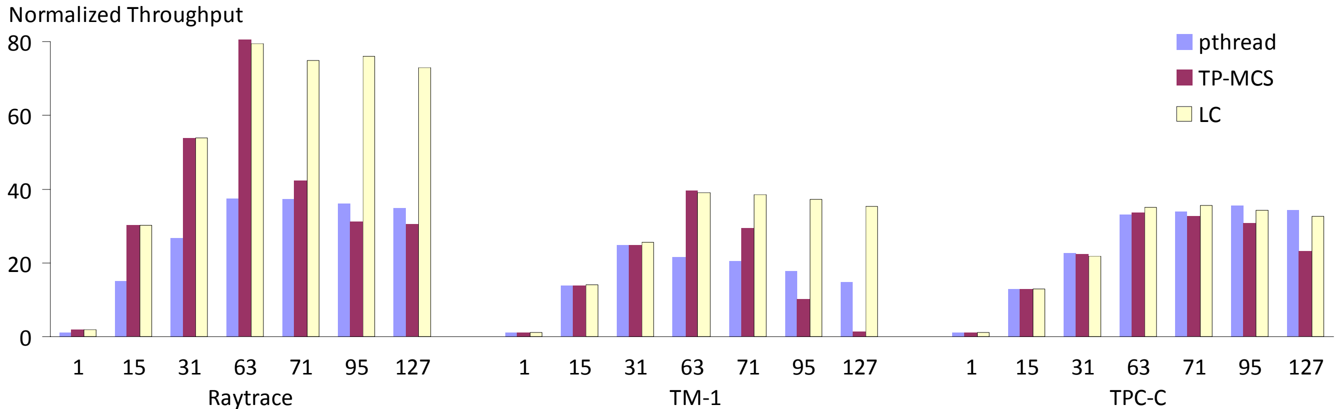

Figure 11. Application performance as the thread count varies (64 threads is 100% load).

tered together, with throughput given for pthreads, TP-MCS, and load control as different colors.

As the figure shows, with TP-MCS Raytrace outperforms the standard pthread_mutex by a wide margin as long as load remains under 100%, corroborating prior findings that heavyweight OS mutex locks are ill-suited for high-performance computing. However, even with preemption resistance, the priority inversions which afflict all spin-based primitives quickly destroy performance. At the highest load shown in the figure the spinlock loses more than 60% of its peak performance. In contrast, load control makes spinlocks perform far better, with a slight advantage over TP-MCS even for load below 100% as the OS will occasionally preempt threads to allow other processes to run. The results for TM-1 are similar to Raytrace, except single-thread performance is unaffected by the higher cost of acquiring the OS mutex compared to a lightweight spinlock. This occurs because the database engine spends less time inside critical sections. However, TM-1 is still highly sensitive to preemption, leading to the same poor performance as spinning without load control to protect it.

For TPC-C, the high levels of application-level contention reduce significantly the pressure on internal mutex locks because threads block frequently on database locks instead. As a result, the system becomes less sensitive to preemptions, and the TP-MCS lock provides acceptable performance even near 150% load. For the same reason, pthread_mutex does not become overloaded and performs as well as load control. This is also to be expected, because an adaptive mutex under low contention is just a spinlock with the ability to deschedule spinning threads if the lock holder is preempted — the same effect load control provides. We verified that the behavior is due to contention for database locks by removing the badly behaved Delivery transaction from the workload mix. Doing so boosted performance significantly, eliminated nearly all variance in throughput, and made all synchronization approaches behave similarly to TM-1.

Overall, load control allows spinning to perform well for a wide variety of application behaviors and load levels. It imposes virtually no overhead for light load while preserving performance as load passes 100%. Even for the highest loads, load control maintains 85-92% of peak performance, making spinning a viable approach to contention management. In fact, load control is so effective that replacing the preemption-resistant TP-MCS with a standard MCS lock gives only a minor performance penalty, confirming that destructive convoys are no longer able to form.

5.5 Interference from Other Processes

In an unmanaged system where processes receive CPU time based on the number of runnable threads they produce, load control potentially puts its host process at a disadvantage compared to processes which do not use it. The worry is that a load-controlled process will detect an overload due to some other process and respond by putting its own threads to sleep. In the worst case, a load-controlled process would gradually disappear as more and more of its threads sleep in response to outside pressure. In order to quantify this risk we run two TM-1 benchmarks at the same time, forcing them to compete for processor time.

Figure 12 shows the outcome of the following scenario: Suppose process "self" uses 100% of the machine's processing power and applies load control. When process "other" appears and competes with "self" for CPU time, it should not be able to cause starvation, regardless of who uses load control. We vary the number of runnable threads in "other" along the x-axis and plot the resulting throughput for both processes. Each pair of bars shows the effect when "other" does not or does use load control. As expected, competition from "other" reduces the throughput "self" can achieve, but it turns out that load control is relatively safe from adversaries. When both processes use load control they share the system quite effectively, with only a 10-15% drop in aggregate performance and a reasonable balance of power. Even when "other" does not use load control at all and creates excessive numbers of threads, "self" still retains roughly 35% of its peak performance while "other" suffers low performance due to priority inversions.

The robustness of load control in the face of external processes leads us to conclude that, for normal competition, load control poses little risk to a process. However, if an adversary were to create a process whose only purpose is to consume CPU time (with no regard for its own performance), load control is somewhat vulnerable. However, we note that this vulnerability exists independent of load control, and has a straightforward solution. All operating systems provide mechanisms for isolating processes from each other, including processor sets, usage caps, and priority schemes. Any mechanism which ensures a process receives CPU time independent of the number of threads other processes create will prevent adversarial processes from pushing an important process off CPU, whether the latter uses load control or not.

6. Discussion

The previous section demonstrates that load control provides an effective way to manage heavy load without resorting to blocking

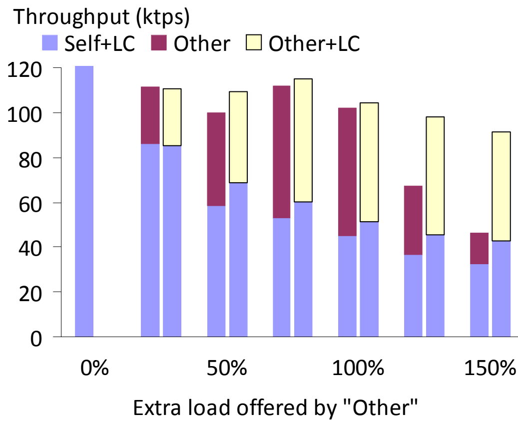

Figure 12. Cost of interference from other processes (TM-1)

synchronization. We next discuss some of the limitations of load control, as well as possible extensions to the scheme.

6.1 Limitations of load control

Though load control is generally quite robust, there exist a handful of situations in which it performs poorly. This section discusses two: large, transient changes in load, and nested lock acquires.

6.1.1 Large, transient changes in load

As indicated by the database workload results, normal scheduler activity due to threads waking and sleeping does not cause difficulties for load control. This occurs for two reasons. First, no change in load is extremely large, and the changes vary around a relatively stable average (see Figure 6). However, whenever the system encounters a large spike (or dip) in load which is small relative to the load controller's update interval, load control will respond too late or not at all, leading to rapid oscillation between priority inversions and low utilization. The challenge, visible in Figure 10, is to keep controller's update interval long enough to ignore unimportant spikes in load while still being responsive to important trends, and if spikes are too large no good balance can be achieved. Benchmarks which introduce some kind of "think time" are especially pernicious because, unlike I/O completions or condition variable signals, the OS will only process timeouts once per scheduler tick, usually at 10msec intervals. Load control responds poorly to the resulting long periods of inactivity punctuated by large load spikes at scheduler ticks. Fortunately, real-world applications do not usually rely on timeouts in this way, and we argue that this effect is uncommon outside the benchmarking world.

6.1.2 Nested critical sections

Our load control mechanism is based on the assumption that threads which spin are not important to forward progress in the short term and are thus good candidates to block in response to overload. However, if a thread holding a lock enters load control while spinning to acquire a second lock, any other thread which attempts to acquire the first lock will encounter priority inversion caused by load control. While this effect is clearly undesirable, neither of our large-scale benchmarks exhibits the problem. Raytrace acquires no nested locks at all. Shore-MT acquires several, but the outer lock is virtually contention-free and serves only to protect against rare races with at most one other thread, which limits the risk load control introduces.

In general, we expect scalable applications not to nest acquires of contended locks because it increases critical section lengths and causes threads to wait for resources they do not want (because the thread holding their desired resource waits). However, it should be straightforward to extend the load control mechanism to allow threads to request waking lock holders which were load controlled while spinning. Doing so would limit the duration of priority inversions to roughly a context switch time, the same as any other unwanted preemption which load control compensates for.

6.2 Extending Load Control

Although the load control implementation presented in this paper performs well under a variety of circumstances, several extensions could potentially enable even better performance, especially for the kinds of corner cases outlined in the previous section.

6.2.1 Applying principles from control theory

Many of the weaknesses of the load control implementation presented here — such as sensitivity to high-frequency oscillations and inaccurate measurements — are well-known challenges addressed by a large body of work in the field of control theory. The robustness of load control would likely improve greatly with the application of a low-pass filter to smooth oscillations in measured load and a well-tuned PID control loop [13] to stabilize control output. More sophisticated approaches such as Kalman filtering [21], which filter out noise and predict the state of the system in between measurements, could yield even greater improvements. When incorporating control theory principles, however, the designer must ensure that the overhead imposed by a more sophisticated scheme does not impact the critical path and cause a net loss in performance (see Section 3.2.3, for example).

6.2.2 Improved OS support

The primary limitation of the current load control scheme is difficulty of obtaining accurate process usage statistics. Mainstream Linux kernels do not even track these statistics, and Solaris mechanism is not designed for frequent use: cost increases linearly with the number of threads (and, hence, the importance of load control), and serializes other kernel operations which affect those threads, particularly scheduling decisions. As a result, scheduler-intensive applications, such as TPC-C, see a small performance penalty even for the relatively infrequent load control update interval we use.

Another weakness of load control is that it must measure load at periodic intervals because it cannot communicate with the scheduler. If the OS provided a lightweight way to notify applications about sudden changes in the number of runnable threads (such as raising a signal), load control could respond asynchronously to transient load changes and avoid the tradeoffs that polling imposes.

Finally, Solaris "scheduling control" (which our spinlock implementations use) allows threads to discourage the scheduler from preempting them [2]. However, the effectiveness of scheduling control appears to decrease quickly with rising load and contention [19] because the OS must maintain fair scheduling regardless of an application's wishes to the contrary. One potential improvement would be to allow threads to not only activate scheduling control, but indicating a different thread from the same application which the scheduler should preempt instead. This would allow the OS to honor scheduling control requests without penalizing other processes — at worst a process can only penalize its own performance by specifying victims unwisely.

The difficulty with relying on, or suggesting, OS enhancements is that they are seldom accepted immediately (if ever), and even less often do they become widespread enough that applications

using them are reasonably portable. Scheduling control (Solaris only, originally part of a much larger proposal [2]), futex (Linux only), and microstate accounting (Solaris only), all illustrate the problem.

7. Alternative Approaches

Our proposed load control mechanism strives to make lock-based synchronization and OS scheduling work well together by avoiding the situations which cause bad interactions between them. However, a large body of research proposes alternatives which instead circumvent either locks or the OS scheduler. This section highlights briefly these approaches and how they relate to load control. In general, the alternatives are complimentary to load control in that they provide high-performance solutions for specific cases, at the expense of significant design and implementation effort. In contrast, the goal of load control is to provide reasonable performance in a way that both is general-purpose and easy to deploy.

7.1 Alternatives to Locking

Lock-free approaches [16] exploit a deep knowledge of the underlying algorithms and data structures to break critical sections into atomic operations which each leave the system in a consistent state. Contention manifests as operations which must be retried or compensated (spinning), but lock-free approaches are immune to preemption because other threads ignore or undo the preempted thread's partial work instead of waiting for it to complete. The drawback of lock-free approaches is that they lead to specialized and complex designs which may not apply to the problem at hand.

Transactional memory trades the complexity and overhead of mutual exclusion for an intuitive transactional model which detects and resolves conflicts after they arise. Software [30] implementations are available, if somewhat slow, and hardware [17] support has also been proposed. Transactions relieve the user from most concerns about spinning, blocking, and preemptions, but these issues resurface in the underlying conflict management implementation. Optimistic approaches retry conflicting transactions repeatedly until they succeed (spinning), while pessimistic approaches block conflicting threads until they can safely continue or deadlock is detected. Lock-based (blocking) contention management schemes typically outperform more optimistic ones under load [10]; the latter respond poorly to OS preemptions [11] for the same reasons that spinlocks suffer. These results indicate that transactional memory systems also have the potential to benefit from advances in contention management and scheduling.

7.2 Alternatives to OS Scheduling

User level threading provides a cooperative threading model to avoid unwanted preemptions. Such environments can safely use spinlocks because accidental priority inversion never arises. However, the developer must reimplement (and supplant) the OS scheduler in order to add domain-specific knowledge, and user-level scheduling requires OS kernel support to handle asynchronous events and competing processes robustly. Proposals as Scheduler Activations [2], which would provide such support, have not yet been adopted by mainstream OS vendors.

Event-based programming models convert control flow into dataflow, replacing contention management with queue management and load balancing. This approach is highly effective for applications such as web servers, where request context is lightweight or even stateless, and frameworks such as SEDA [32] ease significantly the burden of scheduling and load balancing. However, the specialization these approaches enable is also their main drawback. The programming model is more complex than threading, and each application requires a tailored solution. In particular, the user must identify opportunities to exploit dataflow, pipelining, and data partitioning; and effective load balancing is a particular challenge.

The DORA database engine [28] highlights some of the challenges posed by migrating to an event-based programming model. DORA converts Shore-MT into an event-driven database engine which schedules transactions based on the data they access rather than the actions they perform. Careful scheduling, asynchronous I/O, and data partitioning reduce significantly the irregular data accesses common to transaction processing, but the new model requires a significant redesign of the database execution engine and opens up new research challenges. Transactions are much more stateful than web applications, and database operations vary greatly in complexity and relative utilization, which impedes load balancing efforts. Identifying (automatically) fine-grained dataflow from user-specified SQL remains an open problem, and data dependencies which cannot be resolved statically impede efforts to partition data.

8. Conclusions

Contention management and thread scheduling are fundamental concerns in any parallel application, and unwanted interactions between the two often lead to poor performance. Spinning-based synchronization primitives suffer from priority inversion due to preempted lock holders, while blocking-based primitives add high overhead to the critical path of lock handoff. This paper proposes a third approach, using blocking to control the number of runnable threads and then spinning in response to contention. Treating load and contention separately allows applications to avoid blocking on the critical path and is effective even in the presence of high load and competition from other processes. We find that load control allows superior performance to blocking synchronization for all of the workloads examined, while remaining robust to OS preemptions, unlike normal spinning. Our implementation is transparent to the application and could easily be incorporated as a standard library to improve performance of existing applications which rely on OS-provided mutex locks. The effectiveness of the straightforward load control scheme presented in this paper indicates that there remains a significant opportunity for operating systems and libraries to make parallel programming easier and better performing. Finding lightweight methods of improving the communication between OS and application is key to achieving this goal.

Acknowledgements

This work was partially supported by Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and Swiss National Foundation funds. The authors would like to thank the anonymous reviewers for their extremely helpful comments, and Ippokratis Pandis for his feedback during later revisions.

References

[1] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proc. ISCA* (1989), pp. 396-406.

[2] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *ACM Transactions on Computer Systems (TOCS)* 10,2 (Feb 1992), pp. 53-79.

[3] N. Bartolini, G. Bongiovanni, Simone Silvestri. Self-* through self-learning: Overload control for distributed web systems. In *International Journal of Computer and Telecommunications Networking* 53,5 (Apr 2009), pp. 727-743.

[4] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. PACT* (Oct 2008). Source package available at http://parsec.cs.princeton.edu.

[5] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *ACM SIGOPS Operating Systems Review* 13,2 (1979) pp. 20-25.

[6] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, A. Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing* 21,2 (May 1994), pp. 246-254.

[7] B. Cantrill, M. Shapiro, and A. Leventhal. 2004. Dynamic instrumentation of production systems. In *Proc. Usenix Annual Technical Conference,* 2004.

[8] M. Carey, S. Krishnamurthi, and M. Livny. Load control for locking: the "half-and-half" approach. In *Proc. Symposium on Principles of Database Systems (PODS)* (Apr 1990).

[9] J. Carlstrom and R. Rom. Application aware admission control and scheduling in web servers. In *Proc. IEEE INFOCOM (2002).*

[10] D. Dice and N. Shavit. What really makes transactions fast? In Proc. *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Compouting (Transact)* (Jun 2006).

[11] A. Dragojevi, R. Guerraoui, and M. Kapalka. Dividing Transactional Memories by Zero. In *Proc. Transact* (2008, Salt Lake City, UT).

[12] H. Franke, R. Russell, M. K. Fuss. Futexes and furwocks: Fast userlevel locking in linux. In *Proc. 2002 Ottawa Linux Summit* (2002).

[13] G. Franklin, J. Powell, and A. Emami-Naeini. *Feedback control of dynamic systems*, 4th edition. Prentice Hall, NJ, USA.

[14] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating scheduling policies and synchronization methods on the performance of parallel applications. In *Proc. ACM SIGMETRICS Conference on Measuring and Modeling Computer Systems* (May 1991).

[15] B. He, W. N. Scherer III, and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *Proc. High Performance Computing (HiPC)* (2005).

[16] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems (TOPLAS)* 13,1 (Jan 1991), pp. 124-149.

[17] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News* 21, 2 (May 1993), pp. 289-300.

[18] IBM. *Telecom Application Transaction Processing (TATP) Benchmark Description.* Available online at http://tatpbenchmark.sourceforge.net/TATP_Description.pdf.

[19] R. Johnson, M. Athannassoulis, R. Stoica, and A. Ailamaki. A new look at the roles of spinning and blocking. In *Proc. ACM SIGMOD Workshop on Data Management on New Hardware (DaMoN)* (Jul 2009, Providence, RI).

[20] R. Johnson, I. Pandis, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. *In Proc EDBT'09* (Mar 2009, St.Petersburg). Source code and benchmark kit available at http://diaswww.epfl.ch/shore-mt/

[21] R. Kalman. A New Approach to Linear Filtering and Prediction Problems. Transactions of the ASME — Journal of Basic Engineering 82,D (1960), pp. 35-45.

[22] P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proc. International Symposium on Parallel Processing* (Apr. 1994), pp. 165-171.

[23] J. Mauro and R. McDougall. Solaris Internals: Core Kernel Components. Sun Microsystems Press (2001).

[24] J. M. Mellor-Crummey, M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM TOCS* 9,1 (Feb 1991), p.21-65.

[25] A. Monkeberg, G. Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data contention thrashing. In *Proc. Very Large Databases (VLDB)* (1992).

[26] Nokia. *Network Database Benchmark.* Specification and reference implementation available online at http://hoslab.cs.helsinki.fi/homepages/ndbbenchmark/

[27] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. Conf. on Dist. Computing Systems* (1982).

[28] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-Oriented Transaction Execution. To appear, *Proc. of the VLDB Endowment 3,1* (Aug 2010).

[29] D. P. Reed and R. K. Kanodia. Synchronization with Eventcounts and Sequencers. Communications of the ACM, 22(2):115–23 (Feb. 1979).

[30] N. Shavit and D. Touitou. Software transactional memory. In *Proc ACM Symposium on Principles of Distributed Computing (PODC)* (Aug 1995), pp. 204-213.

[31] Transaction Processing Council (TPC). *TPC benchmark C (OLTP) standard specification, revision 5.9*. Available online at http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[32] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. Symposium on operating systems principles (SOSP)* (Dec 2001).

[33] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc ISCA* (Jun 1995), pp. 24-38.