A New Look at the Roles of Spinning and Blocking

Ryan Johnson[†‡] **ryanjohn@ece.cmu.edu**

Manos Athanassoulis[‡] **manos.athanassoulis@epfl.ch**

Radu Stoica[‡] **radu.stoica@epfl.ch**

Anastasia Ailamaki[‡] **natassa@epfl.ch**

[†]**Carnegie Mellon University**
Pittsburgh, PA

[‡]**École Polytechnique Fédérale de Lausanne**
CH-1015 Lausanne

ABSTRACT

Database engines face growing scalability challenges as core counts exponentially increase each processor generation, and the efficiency of synchronization primitives used to protect internal data structures is a crucial factor in overall database performance. The trade-offs between different implementation approaches for these primitives shift significantly with increasing degrees of available hardware parallelism. Blocking synchronization, which has long been the favored approach in database systems, becomes increasingly unattractive as growing core counts expose its bottlenecks. Spinning implementations improve peak system throughput by a factor of 2x or more for 64 hardware contexts, but suffer from poor performance under load.

In this paper we analyze the shifting trade-off between spinning and blocking synchronization, and observe that the trade-off can be simplified by isolating the load control aspects of contention management and treating the two problems separately: spinning-based contention management and blocking-based load control. We then present a proof of concept implementation that, for high concurrency, matches or exceeds the performance of both user-level spinlocks and the pthread mutex under a wide range of load factors.

1. INTRODUCTION

Recent shifts in computer architecture have resulted in systems containing multiple cores per chip, with core counts projected to double every two years for the foreseeable future. While multicore architectures make available an unprecedented degree of hardware parallelism, they also pose new challenges for database engine design. Increasing the number of concurrent threads puts pressure on internal database engine components and exposes new bottlenecks in the system [7]. Database systems have long depended on blocking synchronization primitives (supplied by the operating system) to manage contention because they are both effective and offer predictable performance over a wide range of system load factors. However, we find that the trade-offs between spinning and blocking primitives shift significantly with increasing degrees of available hardware parallelism. In particular, blocking primitives become unattractive because they result in low system utilization for the high core counts which now appear in commodity servers. By utilizing fully, the machine spinlocks improve performance by 2x or more, but suffer from unstable performance under load.

Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN 2009), June 28, 2009, Providence, Rhode Island.
Copyright 2009 ACM 978-1-60558-701-1 $10.00.

The strengths and weaknesses of spinning and blocking as contention management strategies are well known, and several approaches have been proposed to address the weaknesses or balance their trade-offs. For example, mutex locks in certain operating systems make use of adaptive spinning to avoid the cost of context switches when the wait for a lock[1] is short, leading to performance competitive with spinlocks while avoiding most of the weaknesses of spinning. Other work suggests heuristics for optimizing the duration of spinning based on machine size and workload [3]. Recent research [6] has also addressed partially the negative interaction between spinning and thread preemptions due to OS activity. While these techniques are all helpful, however, they do not address the underlying sources of poor performance.

In a loaded system there are two separate, though interacting, requirements to ensure smooth operation: load control and contention management. Load control seeks to keep the number of active threads in the system low enough that they do not interfere excessively with each other. The goal of contention management, on the other hand, is to serialize threads in an orderly fashion when they contend for a resource. We show that, for both blocking and spinning primitives, performance problems arise from negative interactions between (largely non-existent) load control in the system and the chosen contention management scheme. Blocking primitives tend to deschedule too many threads in response to contention, and the inadvertent load control leads to low machine utilization. Spinning primitives are unable to respond to high load appropriately, leading to near-livelock as spinning threads crowd out lock holders and other threads which would otherwise contribute to forward progress in the system. We note that, in both cases, the lock implementation, not the application, limits scalability.

This paper makes the following contributions. First, we measure the scalability limits of existing synchronization primitives and identify negative interactions with load control and scheduling as the underlying cause of poor performance. We then propose to decouple load and contention management in order to optimize them separately, removing the negative interactions and trade-offs which arise when they are lumped together. In order to demonstrate the potential benefits of this approach we implement a proof-of-concept mutex library which applies explicit load management based on machine utilization rather than blocking in response to mutex contention. We test our implementation inside the Shore-MT storage manager [8] and find that it matches or exceeds the performance of both spinning- and blocking-based primitives as load varies from near-idle to heavily oversaturated, improving throughput by up to 50% for load factors where neither spinning nor blocking performs well.

1. Throughout the paper, we call low-level synchronization primitives "locks" instead of "latches" because their names all contain the former.

The rest of the paper is organized as follows: Section 2 provides background and related work in the area of contention management. Section 3 evaluates a variety of mutex locks using different contention management approaches and identifies the bottlenecks which limit their performance. Section 4 discusses potential avenues for improved synchronization and Section 5 concludes.

2. CONTENTION MANAGEMENT

In a balanced database system there are no I/O or other hardware resources bottlenecks and running applications are well-behaved (do not compete excessively for database locks). Under these circumstances, serialization arises mostly out of contention for the mutex locks which protect internal data structures. This contention, in turn arises because the corresponding critical sections are too long to support the number of threads attempting to enter. In a scalable system critical sections must either be very short (allowing many threads to enter per unit time) or well off the critical path (few threads attempting to enter per unit time).

Once contention occurs the system must apply some approach for serializing threads in an orderly fashion. Two basic contention management strategies exist: a waiting thread can spin (repeatedly poll the lock until it becomes free) or block (deschedule itself and wait for notification when the lock becomes free). Both approaches affect critical section behavior significantly, but in different ways. Several factors influence their performance, including delays caused by blocking, time wasted by spinning, and susceptibility to forming convoys, which we explore in the following subsections.

2.1 Blocking

Contention management through blocking has been a staple operating system feature for decades. Because the operating system is aware of which threads hold locks it can avoid descheduling them and also avoid scheduling threads which cannot make forward progress. In uniprocessor systems blocking is the only feasible contention management scheme because the lock holder cannot release the lock as long as another thread remains scheduled. Even with multiprocessors blocking is attractive because it is predictable and avoids squandering limited processor resources on threads which are not making progress.

The disadvantage of blocking is the high cost of context switching — typically 12-20 usec — which must be performed twice per lock handoff (once to sleep, and again to wake). For short critical sections the extra delay adds directly to the critical path of the entire system, capping processor utilization and leading to low, though stable, performance.

2.2 Spinning

In an unloaded system, spinning achieves high performance because it requires no coordination with the operating system and pays, on average, just two cache miss latencies per lock handoff.[1] On a chip multiprocessor with shared caches the critical path latency of a lock acquire and release can be as low as 150 nsec.

Once the system becomes loaded, however, spinning quickly begins to hurt performance. Spinning threads decrease the number of processors available for useful work, achieving suboptimal performance when there are more runnable threads than hardware contexts. Further, because spinning is not coordinated with the operating system, the latter does not take lock holders or spinners into account when making scheduling decisions. The lock holder as just as likely to be preempted as any other thread, and once descheduled it will likely not run again for 100msec or longer on many systems. Meanwhile, spinning threads are allowed to occupy processors even though the lock holder is not making forward progress. Preempted lock holders and useless spinning lead to a system of n threads where each thread must wait, on average, for O(n) other threads to finish their time slices before they are able to acquire and release the lock.

2.3 Hybrid Spinning/Blocking Approaches

Both spinning and blocking primitives can employ hybrid approaches which attempt to balance the trade-offs between spinning and blocking. For example, Solaris 8 introduced an adaptive mutex implementation where threads spin until they acquire the lock, detect that the lock holder is off cpu, exhaust their time slice, or time out; threads join the lock's sleep queue only if they could not acquire it by spinning. In addition, the system will always pass the lock to a spinning thread if it can, even at the risk of starving sleeping ones.[2] In Solaris version 9 and newer, the adaptive mutex has become the favored blocking primitive in both kernel and user code, including the pthreads library.

Spinning primitives also have several options available for descheduling waiting threads, including yielding the processor, sleeping, or even switching to a blocking mutex implementation if contention becomes too high. This form of blocking is difficult to exploit, however, for two reasons. First, sleeping or yielding is a coarse-grained and imprecise operation in most systems; the scheduler's clock resolution is typically around 10msec. Second, threads will not wake up before their requested time expires even if the lock becomes available sooner than expected. In contrast, blocking primitives explicitly block and unblock rather than sleeping, and the scheduler is immediately aware of threads which have been unblocked by a lock handoff.

All hybrid approaches must decide how long to spin before giving up and blocking. If too few threads sleep too little the system will remain overloaded; if delays are too long or too common processor utilization will plummet. Prior work has suggested heuristics for tuning the amount of spinning in the system as the thread and processor counts vary [3].

2.4 Convoys and Preemption Resistance

Convoys [2] are a form of quasi-deadlock that arises when the lock holder is preempted by the operating system, triggering a chain reaction of preemptions as waiting threads deschedule themselves or exhaust their time slices. Once a convoy forms, every lock handoff must pay the cost of waking a blocked thread (or worse, waiting for the system to reschedule a sleeping thread). Purely blocking primitives form convoys by design, but only pay a context switch cost to wake each successive thread where a spinlock

1. The releasing thread pays one miss to fetch the line for update, and the acquiring thread pays another to retrieve it again for reading.

2. The source code comments point out that reduced context switching always improves performance for contention-bound applications while well-behaved (scalable) applications would not block in the first place.

pays a time slice (10usec vs 100msec). Blocking primitives which allow limited spinning can break up convoys by dissolving the sleep queue whenever they run out of spinning threads.

Typical queue-based spinlocks are highly vulnerable to preemption because they enforce a strict FIFO ordering of threads. When the lock holder is preempted its successor in the queue is likely to be near the end of its time slice (having spun the longest) and, if descheduled, will be gone for 100msec or more. Strict FIFO ordering and 10000x longer delays than the OS-supplied primitive combine to virtually freeze the system as it services a few tens of critical sections per second. Preemption-resistant variants [6] improve the situation by removing preempted threads from the FIFO queue, but can do little to prevent the lock holder from being preempted by its spinning successors. As we will see in Section 3, with preemption resistance performance only falls off rapidly instead of instantly once load exceeds 100%.

Blocking and spinning primitives also respond differently when load is reduced after a convoy forms. Because spinning threads detect lock hand-offs nearly instantly, the convoy will dissipate on its own if load drops back below 100%. In contrast, because blocked threads must be woken by a lock handoff, reducing load does not affect convoys on a blocking mutex, though dissolving the queue will restore peak performance if expected wait time drops back to acceptable levels. Hybrid implementations will partially end convoys if load drops because spinning threads become available again. However, if load remains high (but not over 100%) hybrid approaches may leave a long queue of threads to starve as the few spinning threads claim the lock repeatedly.

3. MUTEX PERFORMANCE IN DBMS

In this section we evaluate the performance of several different contention management techniques in a full database system, as we vary both the load and the number of available hardware contexts. Our results indicate that there are no simple trade-offs in this space and that the effects of different strategies depend strongly on the operating region of the system.

3.1 Experimental Setup

We perform all experiments on a Sun T5220 “Niagara II” machine running Solaris 10. The T5220 has 8 cores, each with two processor pipelines and support for 8 hardware contexts (a 64 “CPU” machine to the OS). We chose this machine because it offers more hardware contexts on one chip than any other currently available, giving a glimpse into the future for all platforms as on-chip core counts rise. In addition, Solaris provides high-resolution accounting of where processes spend time, as well as other tools which we use in our measurements. We use processor sets to examine two cases: 16 and 64 contexts. The latter provides maximum parallelism, while the former is representative of the lower parallelism most operating systems (and database engines) were designed for.

For a software platform we choose Shore-MT [8], a highly scalable version of the Shore storage manager [5] which is able to utilize fully the parallelism available in the T5220. We gather statistics by instrumenting Shore-MT — both directly and using DTrace [4] — as well as through profiling and monitoring system utilization. We modified all significant critical sections (those with any contention at all) to use the different lock implementations in

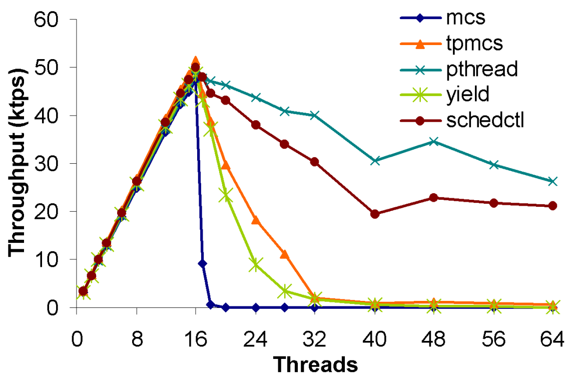

Figure 1. Lock implementation performance for 16 contexts.

turn. We run the TM-1 [10] benchmark in order to minimize database I/O and lock contention, and to stress the synchronization primitives with short transactions.

We examine the following mutex lock implementations:

- pthread - The standard Solaris mutex. It uses a spin-then-block strategy to minimize the number of context switches experienced at lock handoff time.
- mcs - An queue-based spinlock [9] in which each thread spins on an independent memory location; lock hand-off is extremely orderly as a result.
- tpmcs - A “time-published” MCS lock [6] which neutralizes the vulnerability of MCS to convoys which form when the OS preempts a spinning thread.
- yield - An extension of tpmcs where spinning threads yield if they suspect the lock holder has been preempted by the OS.[1]
- schedctl - A further extension of tpmcs which uses Solaris’ schedctl mechanism to temporarily reduce the probability of the lock holder being preempted by the OS.

3.2 Low Parallelism (16 contexts)

Our first experiment compares the performance of the different mutex implementations when there are 16 contexts available to Shore-MT. In this configuration each hardware context has a dedicated processor pipeline and there is little or no performance penalty due to hardware threading. Figure 1 plots the resulting throughput for each implementation as we vary the number of threads in the system along the x-axis.

For loads of 100% or less, all the primitives perform equally well and with linear scaling. While it appears surprising at first that the pthread mutex works so well we note that, under light load and low contention the Solaris implementation behaves essentially as a spinlock, but with the benefit of OS support under load. The schedctl implementation achieves the next best performance because it is able to notify the OS about which threads hold locks. However, enabling schedctl for a lock holder does not always prevent it from being preempted, and when preemptions do occur it still suffers the same weakness as the other spinlocks.

1. The original TPMCS paper proposes the use of yield() but we separate the two variants because they perform quite differently.

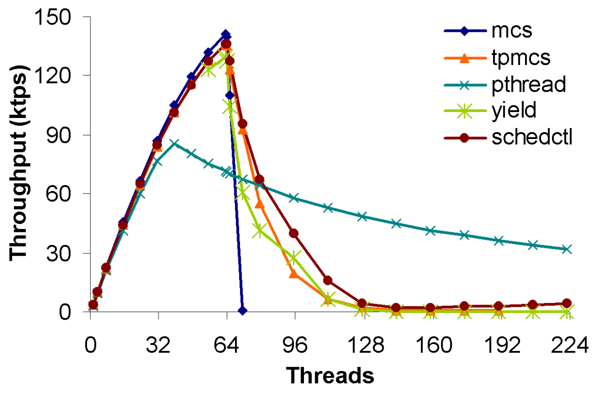

Figure 2. Lock implementation performance for 64 contexts.

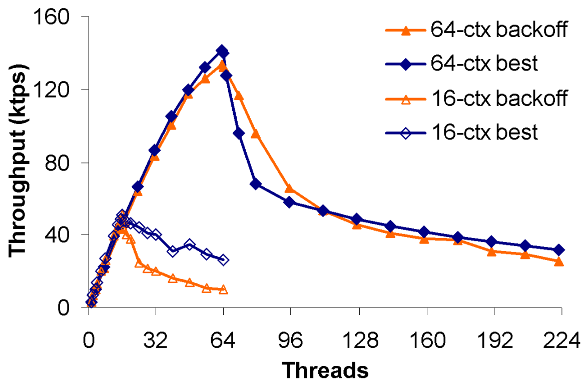

Figure 3. Performance of the load-sensitive backoff scheme.

The tpmcs implementation successfully prevents convoys and does not suffer instant livelock the way mcs does, but performance still drops rapidly because it does nothing to prevent spinning threads from competing with lock holders for CPU time. Adding yielding to tpmcs actually hurts performance because spinning threads mostly yield to other spinning threads rather than the lock holder.

Given that 16 cores is still considered a high-end machine at this point, these results justify the widespread practice of using pthread mutex even in commercial database engines, as it is the best-performing implementation by a significant margin.

3.3 High Parallelism (64 contexts)

The next experiment repeats the same measurements as the first, but this time with all 64 hardware contexts available. Figure 2 plots, as before, the performance of the different mutex implementations as we vary load along the x-axis. We see that the behavior of mcs, tpmcs, and yield are essentially unchanged. However, there are two significant differences from Figure 1 with the increased concurrency. The pthread implementation, which clearly performed the best before, stops scaling at about 39 threads. This occurs because threads increasingly block rather than spinning, and processor utilization actually declines as the high context switch frequency overburdens the OS scheduler. Between 37 and 39 threads the number of context switches per second increases by 30%, and more than doubles between 37 and 47 threads.

The schedctl mechanism, which was fairly effective for low core counts, becomes nearly useless at higher counts. This occurs for two reasons. First, the number of potential lock holders did not increase (the software remains unchanged), but a preempted lock holder competes with four times as many threads for CPU time. Second, with longer queues at contended critical sections, a thread is more likely near the end of its time quantum once it acquires the lock; threads which get preempted before reaching head of queue must rejoin it when they wake. Overall the OS is less willing (or able) to extend the lock holder's time slice to avoid an inconvenient preemption.

3.4 Separating Load from Contention

As we have seen earlier, the negative interactions between load and contention are responsible for lost performance as we increase load in the system. With these observations in mind we propose to break with the traditional trade-off between blocking and spinning, and instead treat them as different problems to optimize separately:

- Blocking forms the basis for effective load management by removing excess threads from the system.
- Spinning provides low-cost contention management because lock hand-offs do not interact with the OS thread scheduler.

In this section we develop a simple load control mechanism to use as a proof of concept. It extends a tpmcs mutex and takes a first step at managing load independently of contention. The mechanism consists of two parts: a daemon thread which maintains statistics about system load, and the regular worker threads which use those statistics to make intelligent decisions about whether to block or spin for contention. Every 10 msec the daemon thread wakes and samples the thread time breakdowns maintained for each process by the Solaris kernel and available via the "proc" file system. The information includes the total time executing on a CPU as well as time spent blocked or waiting on a processor run queue. The daemon maintains a running average of the overload factor in a global variable, where overload is defined as follows:

$$\text{overload} = (\text{queue\_time})/(\text{cpu\_time} + \text{queue\_time})$$

Note that this measure does not tell how many processors are available because the computed value is near zero as long as threads do not wait to run, on average. Once there are more threads than processors the ratio rises to match the fraction of threads in the system which are waiting for a processor. For a severely overloaded machine the overload factor approaches one.

Whenever a thread reaches the end of a transaction, it sleeps with probability equal to the current overload factor (ie never for an unloaded system). The length of the sleep is exponentially distributed, with a tunable mean. Note, however, that the system clock resolution bounds the sleep to at least 10 msec; finer scale sleep requests are ignored. We found that a mean value around 100 msec worked best for our system.

Figure 3 compares the performance of the backoff scheme with the maximum performance of the other implementations. For 64 contexts the backoff scheme allows the underlying tpmcs mutex to track the performance of the best implementation for each operating region: tpmcs for an unloaded machine and pthread as load passes 100%. For 16 contexts the backoff scheme cannot match the

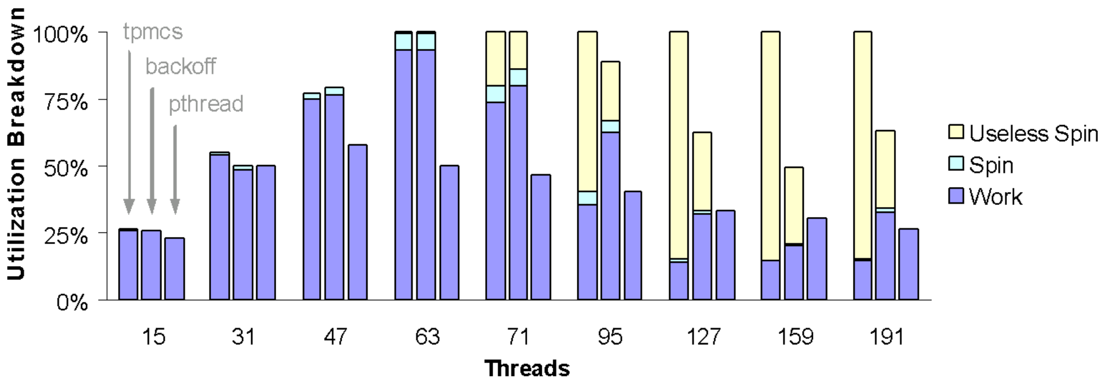

Figure 4. Comparison of utilization impact for spinning (left), backoff (center), and blocking (right) strategies

superior performance of the pthread mutex because the latter still relies largely on spinning and has not hit the context switching bottleneck yet. We note, however, that the backoff scheme achieves the same relative performance for both 16 and 64 contexts, suggesting that it will perform similarly over a range of machine sizes.

4. CAN WE DO BETTER?

Our evaluation so far indicates clearly that all the mutex implementations, including the backoff scheme, have significant shortcomings due to their interactions with scheduling and load control in the system. In this section we discuss potential ways to improve the behavior of locking and make it more predictable as load varies. As a starting point we instrumented Shore-MT using DTrace to track the amount of useless spinning which occurs due to preempted lock holders. For the pthread mutex there is no wasted spinning possible because threads block as soon as the lock holder is preempted. For implementations which block, and for thread counts less than 64, the total utilization can be less than 100%.

Figure 4. shows the breakdown of machine utilization for the tpmcs, backoff, and pthread implementations running on all 64 hardware contexts. Each cluster of bars shows the breakdown for each implementation for the number of threads indicated on the x-axis. As expected, tpmcs saturates the machine with spinning, a growing fraction of which is due to a preempted lock holder and is therefore counterproductive. Once spinning is factored out we see that effective utilization is actually far lower than the blocking mutex. We do not discuss tpmcs further, other than to compare it with other implementations, because we have established that we face a scheduling problem and purely spin-based approaches cannot interact with the OS scheduler. Some combination of spinning and blocking is required for successful load control. The next two subsections discuss the other implementations.

4.1 Ideal Spin-then-block Mutex

The weakness of the pthread implementation is that it never utilizes fully the machine, peaking around 60% utilization between 31 and 47 threads for our system. As load continues to increase utilization continues to drop, explaining the downward performance trend exhibited in Figure 2.

The principal weakness of a blocking mutex is that, in the absences of spinning threads, every lock handoff must wake a blocked thread. The resulting context switch takes 10-20 usec and adds directly to the critical path of the system. Once lock handoff begins

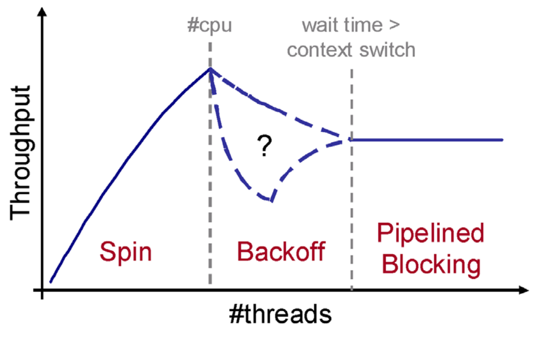

Figure 5. Ideal spin-then-block adaptive mutex

newly arrived spinning threads must wait for the next handoff and are likely to give up and block as well, continuing the cycle of slow hand-offs. If it were possible to take the context switching off the critical path a blocking mutex could achieve much higher performance while still avoiding the weaknesses of spinning.

Figure 5 illustrates the operating regions of an ideal adaptive mutex implementation. On an unloaded system it would behave exactly like a spinlock for minimal overhead. At the other extreme, where there are so many threads in the queue that expected wait time is longer than a context switch penalty, we can envision “pipelining” thread wakeups. Instead of waking its immediate successor, the thread releasing the mutex would wake the i^{th} thread in the queue, with i chosen carefully (and adaptively) so the lock passes to that thread just after it finishes waking. However, pipelining wakeups would only be effective when the queue wait time is long enough to completely overlap the cost of a context switch.

In between the two extremes, a truly adaptive spinning approach such as the one presented by Boguslavky et al [3] would optimize the trade-off between spinning and blocking to minimize the performance loss, while also applying partial pipelining so context switches occur at least partly off the critical path. In the best case performance drops off gradually from peak until pipelining is effective; in the worst case performance drops off rapidly at first (as happens currently with the spinlocks), then recovers somewhat as pipelining becomes effective. Without a specific implementation it is unclear what the shape of the performance curve would be for the central operating region, but we would expect better performance than the existing pthread mutex delivers because the latter does not adapt the length of its spin to reflect load.

4.2 Adaptively Controlling Load

Though we see no real technical barriers to extending the spin-then-block mutex in the ways described above, there are several compelling reasons why we might not want to do so. First, a well-balanced system would not be excessively loaded, meaning that normal operating conditions would fall on the boundary of the left and center regions of Figure 5 where pipelining would be ineffective and performance is least predictable. Second, and more important, though pipelining would take context switching off the critical path, it would not change the underlying behavior of the system. Every lock handoff would still require a context switch, and the frequency of context switches would actually increase to match the improved performance of the lock. Even assuming the scheduler can handle huge numbers of context switches (discussed more below), the cost of these context switches would still add to system load and reduce performance. The expected performance penalty would depend on the ratio of critical section length to context switch cost as well as the fraction of time spent in critical sections. Once the parameters are known we can apply Amdahl's law to approximate the penalty, but it will always exist. If context switching increases without bound the resulting overhead would reduce significantly the number of processors available for useful work, in effect becoming an especially costly form of spinning.

Returning to Figure 4, we see that the backoff scheme exhibits characteristics of both tpmcs and pthread, and counterintuitively produces both wasted spinning and low machine utilization at the same time, though the wasted spinning is greatly reduced when compared with the tpmcs lock. The odd behavior of both the pthread and backoff mutex — overloaded yet idling — arises because the OS can only support a limited number of context switches per second (around 134k/sec on our machine). Once the scheduler becomes overloaded threads cannot run even when there are idle processors available. The previous subsection discussed how a blocking mutex generates a high rate of context switching, and the backoff scheme suffers a similar flaw. Because there is no way to wake a thread that has gone into a timed sleep (as opposed to sleeping in a queue), the backoff scheme must be conservative in how long it removes threads from the system. With too long a backoff the load controller cannot respond to changes in load, but making it too short exposes the context switching bottleneck; this observation explains why the backoff scheme tracks the pthread implementation for the 64-context case (which has blocking) but not for the 16-context case (mostly spinning).

Based on our experience with the pthread and backoff locks, we predict that a truly effective load control mechanism must:

1. wake threads when load drops, in addition to blocking them
2. remove extra threads from the system long enough at a time that to avoid significant context switching overhead
3. detect and respond quickly to changes in system load in order to provide consistent performance.

Designing such a load controller will pose challenges as it must be as lightweight as possible but requires significant interaction with the operating system. However, if successful, the effort would result in both faster and more predictable performance than a traditional spin-then-block mutex.

5. CONCLUSION

Blocking mutexes have long been a favored synchronization primitive for database engines, but the scalability limitations of blocking primitives become painfully apparent as hardware becomes ever more parallel. Our experimental study highlights the weaknesses in current approaches to contention management, and further makes a case for splitting load control and contention management into separate components, optimizing them independently and letting them operate at different time scales: nsec-usec for contention management (spinning) vs msec for load control (blocking). The net effect of this approach is to reduce scheduler load while decoupling the overheads of OS interactions from the critical path of lock hand-off.

6. ACKNOWLEDGEMENTS

This work was partially supported by Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and SNF funds.

7. REFERENCES

[1] G. Amdahl. "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", In Proc. *AFIPS* (30), pp. 483-485, 1967.

[2] M. Blasgen, J. Gray, M. Mitona, and T. Price. "The Convoy Phenomenon." ACM SIGOPS, 13(2), 1979.

[3] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. "Optimal strategies for spinning and blocking." Journal of Parallel and Distributed Computing, 21(2), 1994.

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

[5] M. Carey, D. J. DeWitt, D. J., M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. K. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up persistent applications. In *Proc. SIGMOD*, 1994.

[6] B. He, W. N. Scherer III, and M. L. Scott. "Preemption adaptivity in time-published queue-based spin locks." In Proc. HiPC, 2005.

[7] R. Johnson, I. Pandis, and A. Ailamaki. Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines. In *Proc DaMoN'08*, Vancouver, Canada, 2008.

[8] R. Johnson, I. Pandis, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. *In Proc EDBT'09*, St. Petersburg, Russia, 2009.

[9] J. Mellor-Crummey, and M. Scot. "Algorithms for scalable synchronization on shared-memory multiprocessors." ACM TOCS, 9(1), 1991.

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