Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines

Ryan Johnson*, Ippokratis Pandis*
*Carnegie Mellon University

Anastasia Ailamaki*†
†École Polytechnique Fédérale de Lausanne

ABSTRACT

Critical sections in database storage engines impact performance and scalability more as the number of hardware contexts per chip continues to grow exponentially. With enough threads in the system, *some* critical section will eventually become a bottleneck. While algorithmic changes are the only long-term solution, they tend to be complex and costly to develop. Meanwhile, changes in enforcement of critical sections require much less effort. We observe that, in practice, many critical sections are so short that enforcing them contributes a significant or even dominating fraction of their total cost and tuning them directly improves database system performance. The contribution of this paper is two-fold: we (a) make a thorough performance comparison of the various synchronization primitives in the database system developer's toolbox and highlight the best ones for practical use, and (b) show that properly enforcing critical sections can delay the need to make algorithmic changes for a target number of processors.

1. INTRODUCTION

Ideally, a database engine would scale perfectly, with throughput remaining (nearly) proportional to the number of clients even for a large number of clients. In practice several factors limit database engine scalability. Disk and compute capacities often limit the amount of work that can be done in a given system, and badly-behaved applications (like TPC-C) generate high levels of lock contention and limit concurrency. However, these bottlenecks are all largely external to the database engine; within the storage manager itself, threads share many internal data structures. Whenever a thread accesses a shared data structure, it must prevent other threads from making concurrent modifications or data races and corruption will result. These protected accesses are known as critical sections, and can reduce scalability, especially in the absence of other, external bottlenecks.

For the forseeable future, computer architects will double the number of processor cores available each generation rather than increasing single-thread performance. Database engines are already designed to handle hundreds or even thousands of concurrent transactions, but with most of them blocked on I/O or database locks at any given moment. Even in the absence of lock or I/O bottlenecks, a limited number of hardware contexts used to bound contention for the engine's internal shared data structures. Historically, the database community has largely overlooked critical sections, either ignoring them completely or considering them a solved problem [1]. We find that as the number of active threads grows the engine's internal critical sections become a new and significant obstacle to scalability. Analysis of several open source storage managers [11] shows critical sections become bottlenecks with a relatively small number of active threads, with BerkeleyDB scaling to 4 threads, MySQL to 8, and PostgreSQL to 16. These findings indicate that many database engines are unprepared for this explosion of hardware parallelism.

Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008), June 13, 2008, Vancouver, Canada.
Copyright 2008 ACM 978-1-60558-184-2 ...$5.00.

As the database developer optimizes the system for scalability, algorithmic changes are required to reduce the number of threads contending for particular critical section. Additionally, we find that the method by which existing critical sections are enforced is a crucial factor in overall performance and, to some extent, scalability. Database code exhibits extremely short critical sections, such that the overhead of enforcing those critical sections is a significant or even dominating fraction of their total cost. Reducing the overhead of enforcing critical sections directly impacts performance and can even take critical sections off the critical path without the need for costly changes to algorithms.

The literature abounds with synchronization approaches and primitives which could be used to enforce critical sections, each with its own strengths and weaknesses. The database system developer must then choose the most appropriate approach for each type of critical section encountered in during the tuning process or risk lowering performance significantly.

To our knowledge there is only limited prior work that addresses the performance impact and tuning of critical sections, leaving developers to learn by trial and error which primitives are most useful. This paper illustrates the performance improvements that come from enforcing critical sections properly, using our experience developing Shore-MT [11], a scalable engine based on the Shore storage manager [4]. We also evaluate the most common types of synchronization approaches, then identify the most useful ones for enforcing the types of critical sections found in database code. Database system developers can then utilize this knowledge to select the proper synchronization tool for each critical section and maximize performance.

The rest of the paper is organized as follows. Sections 2 and 3 give an overview of critical sections in database engines and the scalability challenges they raise. Sections 4 and 5 present an overview of common synchronization approaches and evaluate their performance. Finally, Sections 6 and 7 discuss high-level observations and conclude.

2. CRITICAL SECTIONS INSIDE DBMS

Database engines purposefully serialize transaction threads in three ways. Database *locks* enforce consistency and isolation between transactions by preventing other transactions from accessing the lock holder's data. Locks are a form of logical protection and can be held for long durations (potentially several disk I/O times). *Latches* protect the physical integrity of database pages in the buffer pool, allowing multiple threads to read them simultaneously, or a single thread to update them. Transactions acquire latches just long enough to perform physical operations (at most one disk I/O), depending on locks to protect that data until transaction commit time. Locks and latches have been studied extensively [1][7]. Database locks are especially expensive to manage, prompting proposals for hardware acceleration [21].

Critical sections form the third source of serialization. Database engines employ many complex, shared data structures; critical sections (usually enforced with semaphores or mutex locks) protect the physical integrity of these data structures in the same way that latches protect page integrity. Unlike latches and locks, critical sections have short and predictable durations because they seldom span I/O requests or complex algorithms; often the thread only needs to read or update a handful of memory locations. For example, a critical section might protect traversal of a linked list. Critical sections abound throughout the storage engine's code. In Shore-MT, for example, we estimate that a TPC-C Payment transaction — which only touches 4-6 database records — enters roughly one hundred critical sections before committing. Under these circumstances, even uncontended critical sections are important because the accumulated overhead can contribute a significant fraction of overall cost. The rest of this section presents an overview of major storage manager components and lists the kinds of critical sections they make use of.

Buffer Pool Manager. The buffer pool manager maintains a pool for in-memory copies of in-use and recently-used database pages and ensures that the pages on disk and in memory are consistent with each other. The buffer pool consists of a fixed number of *frames* which hold copies of disk pages and provide latches to protect page data. The buffer pool uses a hash table that maps page IDs to frames for fast access, and a critical section protects the list of pages at each hash bucket. Whenever a transaction accesses a persistent value (data or metadata) it must locate the frame for that page, *pin* it, then latch it. Pinning prevents the pool manager from evicting the page while a thread acquires the latch. Once the page access is complete, the thread unlatches and unpins the page, allowing the buffer pool to recycle its frame for other pages if necessary. Page misses require a search of the buffer pool for a suitable page to evict, adding yet another critical section. Overall, acquiring and releasing a single page latch requires at least 3-4 critical sections, and more if the page gets read from disk.

Lock Manager. Database locks preserve isolation and consistency properties between transactions. Database locks are hierarchical, meaning that a transaction wishing to lock one row of a table must first lock the database and the table in an appropriate *intent* mode. Hierarchical locks allow transactions to balance granularity with overhead: fine-grained locks allow high concurrency but are expensive to acquire in large numbers. A transaction which plans to read many records of a table can avoid the cost of acquiring row locks by *escalating* to a single table lock instead. However, other transactions which attempt to modify unrelated rows in the same table would then be forced to wait. The number of possible locks scales with the size of the database, so the storage engine maintains a lock pool very similar to the buffer pool.

The lock pool features critical sections that protect the lock object freelist and the linked list at each hash bucket. Each lock object also has a critical section to "pin" it and prevent recycling while it is in use, and another to protect its internal state. This means that, to acquire a row lock, a thread enters at least three critical sections for each of the database, table, and row locks.

Log Manager. The log manager ensures that modified pages in memory are not lost in the event of a failure: all changes to pages are logged before the actual change is made, allowing the page's latest state to be reconstructed during recovery. Every log insert requires a critical section to serialize log entries and another to coordinate with log flushes. An update to a given database record often involves several log entries due to index and metadata updates that go with it.

Free Space Management. The storage manager maintains metadata which tracks disk page allocation and utilization. This information allows the storage manager to allocate unused pages to tables efficiently. Each record insert (or update that increases record size) requires entering several critical sections to determine whether the current page has space and to allocate new pages as necessary. Note that the transaction must also latch the free space manager's metadata pages and log any updates.

Transaction Management: The system maintains a total order of transactions in order to resolve lock conflicts and maintain proper transaction isolation. Whenever a transaction begins or ends this global state must be updated. In addition, no transaction may commit during a log checkpoint operation, in order to ensure that the resulting checkpoint is consistent. Finally, multi-threaded transactions must serialize the threads within a transaction in order to update per-transaction state such as lock caches.

3. THE DREADED CRITICAL SECTION

By definition, critical sections limit scalability by serializing the threads which compete for them. Each critical section is simply one more limited resource in the system that supports some maximum throughput. As Moore's Law increases the number of threads which can execute concurrently, the demand on critical sections increases and they invariably enter the critical path to become the bottleneck in the system. Database engine designers can potentially improve critical section capacity (i.e. peak throughput) by changing how they are enforced or by altering algorithms and data structures.

3.1 Algorithmic Changes

Algorithmic changes address bottleneck critical sections by either reducing how often threads enter them (ideally never), or by breaking them into several "smaller" ones in a way that distributes contending threads as well (ideally, each thread can expect an uncontended critical section). For example, buffer pool managers typically distribute critical sections by hash bucket so that only probes for pages in the same bucket must be serialized.

In theory, algorithmic changes are the superior approach for addressing critical sections because they can remove or distribute critical sections to ease contention. Unfortunately, developing and implementing new algorithms is challenging and time consuming, with no guarantee of a breakthrough for a given amount of effort. In addition, even the best-designed algorithms will eventually become bottlenecks again if the number of threads increases enough, or if non-uniform access patterns cause hotspots.

3.2 Changing Synchronization Primitives

The other approach for improving critical section throughput is by altering how they are enforced. Because the critical sections we are interested in are so short, the cost of enforcing them is a significant — or even dominating — fraction of their overall cost. Reducing the cost of enforcing a bottleneck critical section can improve performance a surprising amount. Also, critical sections

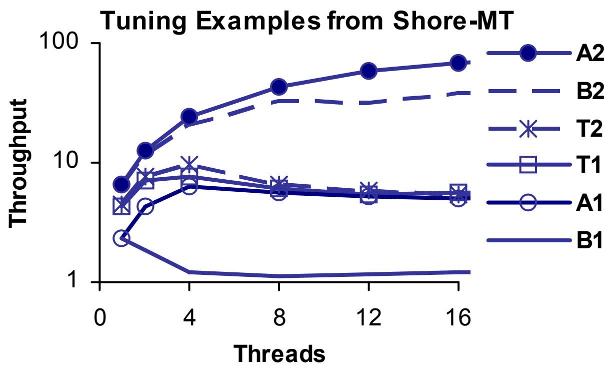


Figure 1.*Algorithmic changes and tuning combine to give best performance.A<n> is an algorithm change; B<n> is a baseline, T<n> is synchronization tuning.*

tend to be encapsulated by their surrounding data structures, so the developer can change how they are enforced simply by replacing the existing synchronization primitive with a different one. These characteristics make critical section tuning attractive if it can avoid or delay the need for costly algorithmic changes.

3.3 Both are Needed

Figure 1 illustrates how algorithmic changes and synchronization tuning combined give the best performance. It presents the performance of Shore-MT at several stages of tuning, with throughput given on the log-scale y-axis as the number of threads in the system varies along the x-axis. These numbers came from the experience of converting Shore to Shore-MT [11]. The process involved beginning with a thread-safe but very slow version of Shore and repeatedly addressing critical sections until internal scalability bottlenecks had all been removed. The changes involved algorithmic and synchronization changes in all the major components of the storage manager, including logging, locking, and buffer pool management. The figure shows the performance and scalability of Shore-MT at various stages of tuning. Each thread repeatedly runs transactions which insert records into a private table. These transactions exhibit no logical contention with each other but tend to expose many internal bottlenecks. Note that, in order to show the wide range of performance the y-axis of the figure is log-scale; the final version of Shore-MT scales nearly as well as running each thread in an independent copy of Shore-MT.

The “B1” line at the bottom represents the thread-safe but unoptimized Shore; the first optimization (A1) replaced the central buffer pool mutex with one mutex per hash bucket. As a result, scalability improved from one thread to nearly four, but single-thread performance did not change. The second optimization (T1) replaced the expensive pthread mutex protecting buffer pool buckets with a fast test and set mutex (see Section 4 for details about synchronization primitives), doubling throughput for a single thread. The third optimization (T2) replaced the test-and-set mutex with a more scalable MCS mutex, allowing the doubled throughput to persist until other bottlenecks asserted themselves at four threads.

B2 represents the performance of Shore-MT after many subsequent optimizations, when the buffer pool again became a bottleneck. Because the critical sections were already as efficient as possible, another algorithmic change was required (A2). This time the open-chained hash table was replaced with a cuckoo hash table to further reduce contention for hash buckets, improving scalability from 8 to 16 threads and beyond (details in [11]).

This example illustrates how both proper algorithms and proper synchronization are required to achieve the highest performance. In general, tuning primitives improves performance significantly, and sometimes scalability as well; algorithmic changes improve scalability and might help or hurt performance (more scalable algorithms tend to be more expensive). Finally, we note that the two tuning optimizations each required only a few *minutes* to apply, while each of the algorithmic changes required several *days* to implement and debug. The performance impact and ease of reducing critical section overhead makes tuning an important part of the optimization process.

4. SYNCHRONIZATION APPROACHES

The literature abounds with different synchronization primitives and approaches, each with different *overhead* (cost to enter an uncontended critical section) and *scalability* (whether, and by how much, overhead increases under contention). Unfortunately, efficiency and scalability tend to be inversely related: the cheapest primitives are unscalable, and the most scalable ones impose high overhead; as the previous section illustrated, both metrics impact the performance of a database engine. Next we present a brief overview of the types of primitives available to the designer.

4.1 Synchronization Primitives

The most common approach to synchronization is to use a synchronization primitive to enforce the critical section. There are a wide variety of primitives to choose from, all more or less interchangeable with respect to correctness.

Blocking Mutex. All operating systems provide heavyweight blocking mutex implementations. Under contention these primitives deschedule waiting threads until the holding thread releases the mutex. These primitives are fairly easy to use and understand, in addition to being portable. Unfortunately, due to the cost of context switching and their close association with the kernel scheduler, they are not particularly cheap or scalable for the short critical sections we are interested in.

Test-and-set Spinlocks. Test-and-set (TAS) spinlocks are the simplest mutex implementation. Acquiring threads use an atomic operation such as a SWAP to simultaneously lock the primitive and determine if it was already locked by another thread, repeating until they lock the mutex. A thread releases a TAS spinlock using a single store. Because of their simplicity TAS spinlocks are extremely efficient. Unfortunately, they are also among the least-scalable synchronization approaches because they impose a heavy burden on the memory subsystem. Variants such as test-and-test-and-set [22] (TATAS), exponential back-off [2], and ticket-based [20] approaches reduce the problem somewhat, but do not solve it completely. Backoff schemes, in particular, are very difficult (and hardware-dependent) to tune.

Queue-based Spinlocks. Queue-based spinlocks organize contending threads into a linked list queue where each thread spins on a different memory location. The thread at the head of the queue holds the lock, handing off to a successor when it completes. Threads compete only long enough to append themselves to the tail of the queue. The two best-known queuing spinlocks are MCS [16] and CLH [5][15], which differ mainly in how they manage their queues. MCS queue links point toward the tail, while CLH

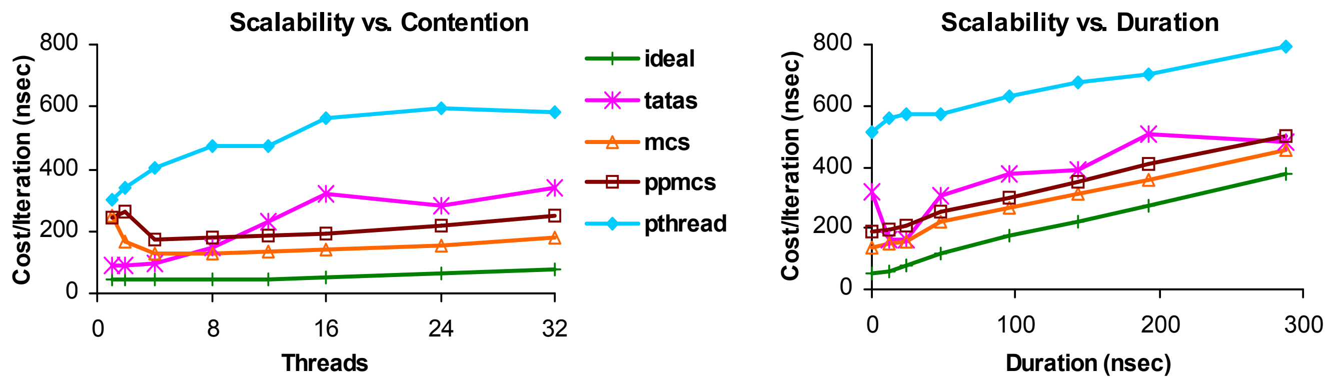


Figure 2.*Performance of mutex locks as the contention (left) and the duration of the CS (right) vary.*

links point toward the head. Queuing improves on test-and-set by eliminating the burden on the memory system and also by decoupling lock contention from lock hand-off. Unfortunately, each thread is responsible to allocate and maintain a queue node for each lock it acquires. In our experience, memory management can quickly become cumbersome in complex code, especially for CLH locks, which require heap-allocated state.

Reader-Writer Locks. In certain situations, threads enter a critical section only to prevent other threads from changing the data to be read. Reader-writer locks allow either multiple readers or one writer to enter the critical section simultaneously, but not both. While operating systems typically provide a reader-writer lock, we find that the pthreads implementation suffers from extremely high overhead and poor scalability, making it useless in practice. The most straightforward reader-writer locks use a normal mutex to protect their internal state; more sophisticated approaches extend queuing locks to support reader-writer semantics [17][13].

A Note About Convoys. Some synchronization primitives, such as blocking mutex and queue-based spinlocks, are vulnerable to forming stable quasi-deadlocks known as convoys [3]. Convoys occur when the lock passes to a thread that has been descheduled while waiting its turn. Other threads must then wait for the thread to be rescheduled, increasing the chances of further preemptions. The result is that the lock sits nearly idle even under heavy contention. Recent work [8] has provided a preemption-resistant form of queuing lock, at the cost of additional overhead which can put medium-contention critical sections squarely on the critical path.

4.2 Alternatives to Locking

Under certain circumstances critical sections can be enforced without resorting to locks. For example, independent reads and writes to a single machine word are already atomic and need no further protection. Other, more sophisticated approaches such as optimistic concurrency control and lock-free data structures allow larger critical sections as well.

Optimistic Concurrency Control. Many data structures feature *read-mostly* critical sections, where updates occur rarely, and often come from a single writer. The reader's critical sections are often extremely short and overhead dominates the overall cost. Under these circumstances, optimistic concurrency control schemes can improve performance dramatically by assuming no writer will interfere during the operation. The reader performs the operation without enforcing any critical section, then afterward verifies that no writer interfered (e.g. by checking a version stamp). In the rare event that the assumption did not hold, the reader blocks or retries. The main drawbacks to OCC are that it cannot be applied to all critical sections (since side effects are unsafe until the read is verified), and unexpectedly high writer activity can lead to livelock as readers endlessly block or abort and retry.

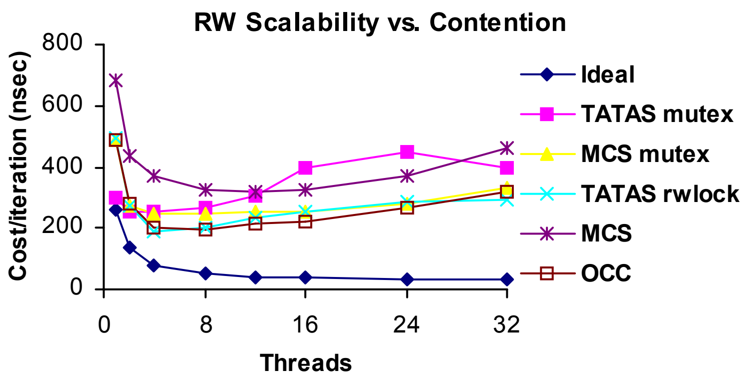
Lock-free Data Structures. Much current research focuses on lock-free data structures [9] as a way to avoid the problems that come with mutual exclusion (e.g. [14][6]). These schemes usually combine optimistic concurrency control and atomic operations to produce data structures that can be accessed concurrently without enforcing critical sections. Unfortunately there is no known general approach to designing lock free data structures; each must be conceived and developed separately, so database engine designers are have a limited library to choose from. In addition, lock-free approaches can suffer from livelock unless they are also *wait-free*, and may or may not be faster than the lock-based approaches under low and medium contention (many papers provide only asymptotic performance analyses rather than benchmark results).

Transactional Memory. Transactional memory approaches enforce critical sections using database-style “transactions” which complete atomically or not at all. This approach eases many of the difficulties of lock-based programming and has been widely researched. Unfortunately, software-based approaches [23] impose too much overhead for the tiny critical sections we are interested in, while hardware approaches [10][19] generally suffer from complexity, lack of generality, or both, and have not been adopted. Finally, we note that transactions do not inherently remove contention; at best transactional memory can serialize critical sections with very little overhead.

5. CHOOSING THE RIGHT APPROACH

This section evaluates the different synchronization approaches using a series of microbenchmarks that replicate the kinds of critical sections found in database code. We present the performance of the various approaches as we vary three parameters: Contended vs. uncontended accesses, short vs. long duration, and read-mostly vs. mutex critical sections. We then use the results to identify the primitives which work best in each situation.

Each microbenchmark creates N threads which compete for a lock in a tight loop over a one second measurement interval (typically 1-10M iterations). The metric of interest is cost per iteration per thread, measured in nanoseconds of wall-clock time. Each iteration begins with a delay of *To* ns to represent time spent out-



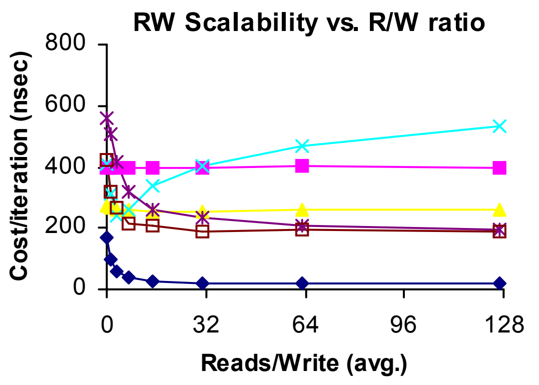


Figure 3. *Performance of reader-writer locks as contention (left) and reader-writer ratio (right) vary.*

side the critical section, followed by an acquire operation. Once the thread has entered the critical section, it delays for *Ti* ns to represent the work performed inside the critical section, then performs a release operation. All delays are measured to 4 ns accuracy using the machine's cycle count register; we avoid unnecessary memory accesses to prevent unpredictable cache misses or contention for hardware resources.

For each scenario we compute an ideal cost by examining the time required to serialize *Ti* plus the overhead of a memory barrier, which is always required for correctness. Experiments involving reader-writers are set up exactly the same way, except that readers are assumed to perform their memory barrier in parallel and threads use a pre-computed array of random numbers to determine whether they should perform a read or write operation.

All of our experiments were performed using a Sun T2000 (Niagara [12]) server running Solaris 10. The Niagara chip is a multi-core architecture with 8 cores; each core provides 4 hardware contexts for a total of 32 OS-visible "processors". Cores communicate through a shared 3MB L2 cache.

5.1 Contention

Figure 2 (left) compares the behavior of four mutex implementations as the number of threads in the system varies along the x-axis. The y-axis gives the cost of one iteration as seen by one thread. In order to maximize contention, we set both *To* and *Ti* to zero; threads spend all their time acquiring and releasing the mutex. TATAS is a test-and-set spinlock variant. MCS and ppMCS are the original and preemption-resistant MCS locks, respectively, while pthread is the native pthread mutex. Finally, "ideal" represents the lowest achievable cost per iteration, assuming that the only overhead of enforcing the critical section comes from the memory barriers which must be present for correctness.

As the degree of contention of the particular critical section changes, different synchronization primitives become more appealing. The native pthread mutex is both expensive and unscalable, making it unattractive. TATAS is by far the cheapest for a single thread, but quickly falls behind as contention increases. We also note that all test-and-set variants are extremely unfair, as the thread which most recently released it is likely to re-acquire it before other threads can respond. In contrast, the queue-based locks give each thread equal attention.

5.2 Duration

Another factor of interest is the performance of the various synchronization primitives as the duration of the critical section varies (under medium contention) from extremely short to merely short. We assume that a long, heavily-contended critical section is a design flaw which must be addressed algorithmically.

Figure 2 (right) shows the cost of each iteration as 16 threads compete for each mutex. The inner and outer delays both vary by the amount shown along the x-axis (keeping contention steady). We see the same trends as before, with the main change being the increase in ideal cost (due to the critical section's contents). As the critical section increases in length, the overhead of each primitive matters less; however, ppMCS and TATAS still impose 10% higher cost than MCS, while pthread more than doubles the cost.

5.3 Reader/Writer Ratio

The last parameter we study is the ratio between the readers and the writers. Figure 3 (left) characterizes the performance of several reader-writer locks when subjected to 7 reads for every write and with *To* and *Ti* both set to 100 ns. The cost/iteration is shown on the y-axis as the number of competing threads varies along the x-axis. The TATAS mutex and MCS mutex apply mutual exclusion to both readers and writers. The TATAS rwlock extends a normal TATAS mutex to use a read/write counter instead of a single "locked" flag. The MCS rwlock comes from the literature [13]. OCC lets readers increment a simple counter as long as no writers are around; if a writer arrives, all threads (readers and writers) serialize through an MCS lock instead.

We observe that reader-writer locks are significantly more expensive than their mutex counterparts, due to the extra complexity they impose. For very short critical sections and low reader ratios, a mutex actually outperforms the rwlock; even for the 100ns case shown here, the MCS lock is a usable alternative.

Figure 3 (right) fixes the number of threads at 16 and varies the reader ratio from 0 (all writes) to 127 (mostly reads) with the same delays as before. As we can see, the MCS rwlock performs well for high reader ratios, but the OCC approach dominates it, especially for low reader ratios. For the lowest read ratios, the MCS mutex performs the best — the probability of multiple concurrent reads is too low to justify the overhead of a rwlock.

6. DISCUSSION AND OPEN ISSUES

The microbenchmarks from the previous section illustrate the wide range in performance and scalability among the different

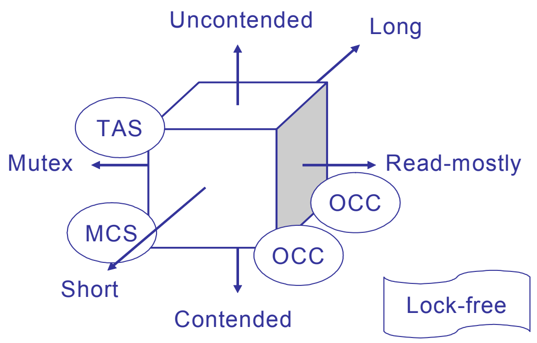


Figure 4.*The space of critical section types. Each corner of the cube is marked with the appropriate synchronization primitive to use for that type of critical section.*

primitives. From the contention experiment we see that the TATAS lock performs best under low contention due to having the lowest overhead; for high contention, the MCS lock is superior thanks to its scalability. The experiment also highlights how expensive it is to enforce critical sections. The ideal case (memory barrier alone) costs 50 ns, and even TATAS costs twice that. The other alternatives cost 250 ns or more. By comparison a store costs roughly 10 ns, meaning critical sections which update only a handful of values suffer more than 80% overhead. As the duration experiment shows, pthread and TATAS are undesirable even for longer critical sections that amortize the cost somewhat. Finally, the reader-writer experiment demonstrates the extremely high cost of reader-writer synchronization; a mutex outperforms rwlocks at low read ratios by virtue of its simplicity, while optimistic concurrency control wins at high ratios. Figure 4 summarizes the results of the experiments, showing which of the three synchronization primitives to use under what circumstances. We note that, given a suitable algorithm, the lock free approach might be best.

The results also suggest that there is much room for improvement in the synchronization primitives that protect small critical sections. Hardware-assisted approaches (e.g. [18]) and implementable transactional memory might be worth exploring further in order to reduce overhead and improve scalability. Reader-writer primitives, especially, do not perform well as threads must still serialize long enough to identify each other as readers and check for writers.

7. CONCLUSION

Critical sections are emerging as a major obstacle to scalability as the number of hardware contexts in modern systems continues to grow and a large part of the execution is computation-bound. We observe that algorithmic changes and proper use of synchronization primitives are both vital to maximize performance and keep critical sections off the critical path in database engines and that even uncontended critical sections sap performance because of the overhead they impose. We identify a small set of especially useful synchronization primitives which a developer can use for enforcing critical sections. Finally, we identify several areas where currently available primitives fall short, indicating potential avenues for future research.

8. ACKNOWLEDGEMENTS

We thank Brian Gold and Brett Meyer for their insights and suggestions, and the reviewers for their helpful comments. This work was partially supported by grants and equipment from Intel; a Sloan research fellowship; an IBM faculty partnership award; and NSF grants CCR-0205544, CCR-0509356, and IIS-0133686.

9. REFERENCES

[1] R. Agrawal, M. Carey, and M. Livny. “Concurrency control performance modeling: alternatives and implications.” ACM TODS, 12(4), 1987.

[2] T. Anderson. “The performance of spin lock alternatives for shared-memory multiprocessors.” IEEE TPDS, 1(1), 1990.

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

[4] M. Carey, et al. “Shoring up persistent applications.” In Proc. SIGMOD, 1994.

[5] T. Craig. “Building FIFO and priority-queueing spin locks from atomic swap.” Technical Report TR 93-02-02, University of Washington, Dept. of Computer Science, 1993.

[6] M. Fomitchev, and E. Rupert. “Lock-free linked lists and skip lists.” In Proc. PODC, 2004.

[7] V. Gottemukkala, and T. J. Lehman. “Locking and latching in a memory-resident database system.” In Proc. VLDB, 1992.

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

[9] M. Herlihy. “Wait-free synchronization.” ACM TOPLAS, 13(1), 1991.

[10] M. Herlihy and J. Moss. “Transactional memory: architectural support for lock-free data structures.” In Proc. ISCA, 1993.

[11] R. Johnson, I. Pandis, N. Hardavellas, and A. Ailamaki. “Shore-MT: A Quest for Scalability in the Many-Core Era.” CMU-CS-08-114.

[12] P. Kongetira, K. Aingaran, and K. Olukotun. “Niagara: A 32-Way Multithreaded SPARC Processor.” IEEE MICRO, 2005.

[13] O. Krieger, M. Stumm, and R. Unrau. “A Fair Fast Scalable Reader-Writer Lock.” In Proc. ICPP, 1993

[14] M. Maged. “High performance dynamic lock-free hash tables and list-based sets.” In Proc. SPAA, 2002.

[15] P. Magnussen, A. Landin, and E. Hagersten. “Queue locks on cache coherent multiprocessors.” In Proc. IPPS, 1994.

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

[17] J. Mellor-Crummey, and M. L. Scott. “Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors.” In Proc. PPoPP, 1991.

[18] R. Rajwar, and J. Goodman. “Speculative lock elision: enabling highly concurrent multithreaded execution.” IEEE MICRO, 2001.

[19] R. Rajwar and J. Goodman. “Transactional lock-free execution of lock-based programs.” SIGPLAN Notices, 37(10), 2002.

[20] D. P. Reed, and R. K. Kanodia. “Synchronization with event-counts and sequencers.” Commun. ACM 22(2), 1979.

[21] J. T. Robinson. “A fast, general-purpose hardware synchronization mechanism.” In Proc. SIGMOD, 1985.

[22] L. Rudolph and Z. Segall. “Dynamic decentralized cache schemes for MIMD parallel processors.” In Proc ISCA, 1984.

[23] N. Shavit and D. Touitou. “Software Transactional Memory.” In Proc. PODC, 1995.