Improving OLTP Scalability using Speculative Lock Inheritance

Ryan Johnson[†‡]
ryanjohn@ece.cmu.edu

Ippokratis Pandis[†]
ipandis@cs.cmu.edu

Anastasia Ailamaki[‡]
natassa@epfl.ch

[†]Carnegie Mellon University
Pittsburgh, PA

[‡]École Polytechnique Fédérale de Lausanne
Switzerland

ABSTRACT

Transaction processing workloads provide ample request level concurrency which highly parallel architectures can exploit. However, the resulting heavy utilization of core database services also causes resource contention within the database engine itself and limits scalability. Meanwhile, many database workloads consist of short transactions which access only a few database records each, often with stringent response time requirements. Performance of these short transactions is determined largely by the amount of overhead the database engine imposes for services such as logging, locking, and transaction management.

This paper highlights the negative scalability impact of database locking, an effect which is especially severe for short transactions running on highly concurrent multicore hardware. We propose and evaluate Speculative Lock Inheritance, a technique where hot database locks pass directly from transaction to transaction, bypassing the lock manager bottleneck. We implement SLI in the Shore-MT storage manager and show that lock inheritance fundamentally improves scalability by decoupling the number of simultaneous requests for popular locks from the number of threads in the system, eliminating contention within the lock manager even as core counts continue to increase. We achieve this effect with only minor changes to the lock manager and without changes to consistency or other application-visible effects.

1. INTRODUCTION

Online transaction processing is a vital and challenging database workload which demands scalable access to rapidly changing data, consistency among thousands of competing requests, tight response time requirements and stringent data reliability and availability. Such heavy use of core database services imposes high overhead, especially for the short transactions common in telecom [15], banking [2], and sales/retail [21][17] workloads. Logical locking is a particularly significant source of overhead in the DBMS. For example, Harizopoulos et. al. report overheads of 16-25% for TPC-C transactions [6] running on a single-core machine with no physical contention for lock data structures.

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 impose 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 [11].

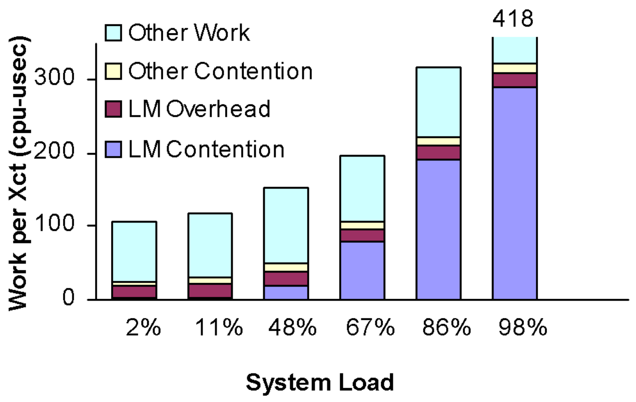

Figure 1. Lock manager overhead as system load increases

1.1 Contention Within the Lock Manager

Virtually all database engines use some form of hierarchical locking [5] to allow applications to trade off concurrency and overhead. For example, requests which access large amounts of data can acquire coarse-grained locks to reduce overhead at the risk of reduced concurrency. At the same time, small requests can lock precisely the data which they access and maximize concurrency with respect to other independent requests. The lock hierarchy is crucial for application scalability because it allows efficient fine-grained concurrency control at the logical level.

Ironically, however, hierarchical database locking causes a new scalability problem while addressing the first one: all transactions must acquire intention locks high in the hierarchy in order to access individual objects. Until recently, single-node database engines time-shared all requests on one to four processors, with little potential for physical contention. However, as core counts continue to double each processor generation the increased hardware concurrency leads to bottlenecks in the centralized lock manager, especially as hierarchical locking forces many threads to update repeatedly the state of a few hot locks.

Physical contention causes locking-related bottlenecks even for scalable database applications which cause few logical conflicts. Because of the inherent behavior of hierarchical locking, we expect that every system will eventually encounter this kind of contention within the lock manager, if it has not done so already. Figure 1 highlights how contention for database locks impacts performance as we increase load in our test system running the NDBB benchmark (see Section 5). The x-axis varies load on the system from very light (left) to very heavy (right) while the y-axis shows the fraction of CPU time each transaction spends in the lock manager (not counting time spent blocked on I/O or true lock conflicts). This figure, and those that follow, define overhead and contention as the useful and useless work, respectively, performed by the system when processing transactions. We can make two observations from Figure 1. First, the overhead (useful work) due to the lock manager is a small part of the total. Second, nearly all contention in the system arises within the lock manager and that contention component grows rapidly. As load increases, lock manager contention becomes a rapidly growing bottleneck responsible for nearly 75% of the transaction's CPU time. This suggests that, in order to improve scalability, we must focus on eliminating contention within the lock manager.

1.2 Reducing Lock Overhead and Contention

The guiding concept of speculative lock inheritance — not releasing locks between transactions — appears in Rdb/VMS [14] as a way to reduce network communication costs. Locks in this distributed database physically migrate to nodes whose transactions acquire them. The authors highlight very briefly a "lock carry-over" optimization which allows a node to avoid the overhead of returning the lock to its home node when transactions complete by caching it locally, as long as no conflicting lock requests have arrived. Each carry-over saves at least one round trip over the network in the event the lock is reused by a later transaction, improving the performance of a two-node system by over 60%. We apply the concept of lock carry-over to the single-node Shore-MT engine to solve the problem of contention for lock state, which did not exist with the high network overheads and low node counts (1-3 in the evaluation) experienced by Rdb/VMS. We also detail an implementation designed for modern database engines running on multicore hardware with shared memory and caches, and where transactions, not nodes, hold locks. SLI allows a centralized lock manager to distribute requests among the many threads that would otherwise contend with each other.

IBM's DB2 provides a performance tuning registry variable, DB2_KEEPTABLELOCK [8], which allows transactions or even connections to retain read-mode table locks between uses, again exploiting the idea of not releasing locks unless necessary. However, transactions only benefit from the setting if they repeatedly release and re-acquire the same locks, and the documentation notes that retaining table locks for the life of a connection leads to "poor concurrency" because other transactions cannot make updates until the connection closes. The setting is disabled by default.

H-Store [18] takes an extreme approach to reducing both overhead and contention by executing transactions one at a time, in a single thread, with no interleaving. However, where SLI requires only a scalable workload, H-Store requires a workload which can be partitioned evenly among multiple machines in order to avoid the high cost of distributed transactions.

Multiversioned buffer pools [3] allow writers to update copies of pages rather than waiting for readers to finish. Copying avoids the need for low-level locking because older versions remain available to readers, but it does not remove the need for hierarchical locks or the corresponding contention which SLI addresses. In addition, for the common case where a transaction updates only a few bytes per record accessed, multiversioning imposes the cost of copying an entire database page per record. Finally, multiversioning provides "snapshot isolation," which suffers from certain non-intuitive update anomalies that are only partly addressed to date [13].

1.3 Speculative Lock Inheritance

The key to reducing contention within the lock manager comes by observing that virtually all transactions request high-level locks in compatible modes; even requests for exclusive access to particular rows or pages in the database generate compatible intent locks higher up, and transactions which require coarse-grained exclusive access are extremely rare in scalable workloads. Further, in the absence of intervening updates, it makes no semantic difference whether a shared-mode lock is released and re-acquired or simply held continuously. Either way a transaction will see the same (unchanged) object, and other transactions are free to interleave their reads to the object as well.

Speculative Lock Inheritance exploits these characteristics of shared-mode locks to transparently and safely allow committing transactions to pass some of their locks to those which follow, without releasing and reacquiring them. Inheriting the hottest locks in the system alleviates contention because most transactions acquire those locks directly from their predecessor instead of making requests to the centralized lock manager. For our test system with 64 hardware contexts running a variety of short transactions, we find that SLI reduces contention in the lock manager to small, near-constant levels even as hardware concurrency increases.

1.4 Contributions and Paper Organization

This paper makes the following contributions. First, we identify the lock manager as a growing scalability bottleneck on multicore hardware, particularly for workloads made up of small transactions. We identify the source of the lock manager bottleneck as contention for updates to the state of the locks themselves. Based on the observation that this contention arises largely from compatible requests for shared-mode database locks, we propose Speculative Lock Inheritance, which modifies the lock and transaction managers to pass selected locks directly from transaction to transaction. Lock inheritance virtually eliminates contention within the lock manager by keeping nearly constant the number of threads which make simultaneous requests for hot locks, even as the number of threads using the locks continues to increase. Our approach takes the lock manager off the critical path in the system and yields throughput improvements of 10%-40% for short transactions.

The rest of this paper proceeds as follows: Section 2 introduces critical sections and discusses ways to alleviate bottlenecks. Section 3 gives an overview of a database engine lock manager and discusses sources of overhead introduced by database locking, and Section 4 describes speculative lock inheritance. Sections 5 through 7 present our experimental methodology and analysis of SLI performance, and we conclude in Section 8.

2. CRITICAL PATHS AND LATCHING

Serialization in a database engine falls into two basic categories: database locks protect (logically) the data stored in the database and enforce consistency between transactions. However, database code paths also include dozens of critical sections, or regions of code which access shared data structures — e.g. buffer pool pages, logs, and database lock state — in ways that must (physically) appear atomic to the rest of the system. Critical sections are protected by *latches*, which apply either mutual exclusion or, less often, reader-writer locking. In contrast with database locks, latches are held, one or two at a time, for very short intervals and acquired far more frequently than lock requests. For example, prior work has shown that the TPC-C Payment transaction, which accesses only 4-6 rows, acquires roughly 100 latches before committing [11]. Historically, systems with one or few processors have depended on latching mostly to prevent untimely OS scheduling decisions and other spurious thread interleavings from exposing inconsistent data; the current trend, however, is for the number hardware contexts to double every processor generation, tracking Moore's Law. As available hardware parallelism increases the serializing effect of latching becomes much more significant for overall performance because more and more threads can attempt to enter critical sections simultaneously. Any latch-protected critical section which causes non-trivial serialization delays is a *bottleneck*; we refer to the set of bottleneck critical sections for a given systems and workload as the *critical path* through the storage engine. We can identify bottlenecks because they either cause transactions to block (not for I/O or database lock conflicts), or spin (indicated by flat or decreasing performance even as CPU utilization grows).

Intuitively, latch-protected critical sections can be seen as a kind of server in traditional bottleneck analysis. Just as disk and CPU can only serve so many requests or perform so much work per second, critical sections have a maximum rate at which threads may enter them and which depends inversely on the length of the critical section. Where hardware bottlenecks can be resolved by adding more resources, however, bottleneck critical sections can only be resolved by making (often significant) changes to the code base. Optimizing the code inside the critical section increases its throughput by a (potentially large) constant factor, but provides only a temporary solution because the degree of hardware parallelism is projected to grow exponentially for the foreseeable future. Even if the optimizations ease the bottleneck for the moment, ever-increasing thread counts will quickly take up the slack and the bottleneck will eventually return.

Fine-grained synchronization is another approach for alleviating bottlenecks; instead of using one latch per *type* of data, fine-grained synchronization provides one latch per *piece* of data. This approach is highly effective for reducing contention by distributing threads to different locations. However, fine-grained synchronization can only eliminate contention if the data scales uniformly with available compute power. Inevitably, contention such as a database log or a few hot locks will become bottlenecks, and this contention cannot be alleviated without fundamental changes to the access pattern — causing fewer threads to request the shared resource in the first place. This kind of change is not always complex to implement, but requires re-evaluating the algorithms and data structures used by the database engine in order to identify the opportunity.

Speculative Lock Inheritance falls under the last category of contention-reducing approaches. The lock manager code has already been highly optimized and uses fine-grained synchronization, but high-level intent locks emerge as a bottleneck if many transactions request them concurrently. Historically this bottleneck was masked by the limited number of available hardware contexts, which in turn allowed only a few threads to request the lock at any given instant. Multicore hardware removes this limitation, pushing the lock manager onto the critical path of short transactions. In order to eliminate this new and growing bottleneck we must find ways to reduce significantly the number of threads which make simultaneous requests for the same hot locks.

3. DATABASE LOCKING

Logical locking is a significant and largely unavoidable overhead in the DBMS. Even with only one thread in the system, Harizopoulos et. al. report overheads between 16 and 25% for TPC-C transactions [6]; contention for database locks only increases the overhead they impose. The high overhead of database locking has even prompted proposals for hardware support [16].

In workloads characterized by large numbers of short transactions — such as telecom and banking workloads (e.g. NDBB [15] or TPC-B [20]) — many simultaneous requests for locks must serialize to check and update internal lock manager state *even though such requests nearly always specify compatible lock modes*. In addition, short transactions touch only one or a few records per table and cannot amortize the cost of acquiring those locks over many accesses or heavy computation. Exponentially growing core counts amplify the serialization effect by allowing more threads to run and compete for data structures at any given instant — an example of the internal bottlenecks modern hardware exposes in database systems [11]. Together these effects make the overhead of acquiring locks a significant fraction of total transaction cost.

3.1 Hierarchical Locking

Hierarchical locking schemes treat the database as a nested data structure of sorts. For example, a database contains tables, which in turn contain pages and rows, and each object at each level of the hierarchy has locks associated with it. For instance, a transaction which accesses only a few records must still acquire a lock on the table to prevent its being dropped by another transaction. To distinguish between direct accesses to an object (such as dropping a table) and indirect accesses to its children (such as reading a row from a table), hierarchical locking defines the following basic lock modes [5]. All database engines using lock-based concurrency control provide these lock modes, though most implementations define additional modes as well for performance reasons:

- Share (S): The holder can read this object, and implicitly holds an S-mode lock on all of its children as well.
- Exclusive (X): The holder can update this object and implicitly holds an X-mode lock on all of its children as well.
- Intention Share (IS): Notifies other transactions that the holder has shared locks on a subset of this object's children; coarse-grained updates are not permitted, but fine-grained updates may proceed if they do not conflict.
- Intention Exclusive (IX): Notifies other transactions that the holder has exclusive locks on a subset of this object's children; no coarse-grained locks are permitted, but fine-grained accesses are allowed if they do not conflict.

Intention locks are vital to application scalability because they allow transactions which access different subtrees in the hierarchy to proceed in parallel while still preventing coarse-grained opera-

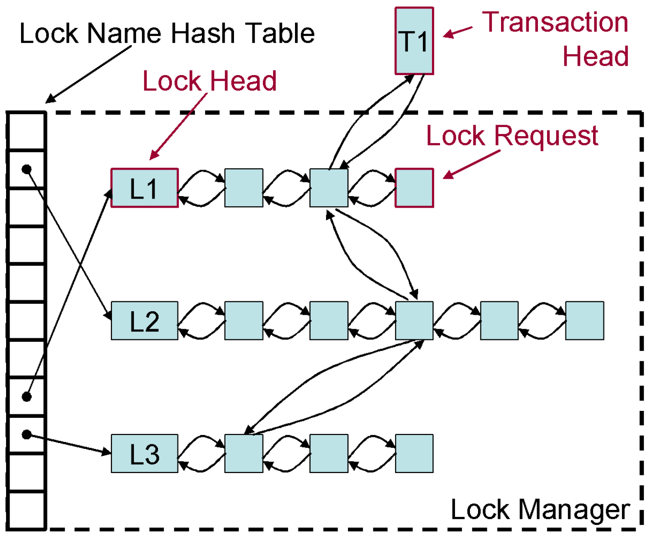

Figure 2. Inside a database engine's lock manager

tions from interfering with fine-grained ones. Acquiring intention locks on the parents of an object of interest also allows the database engine to identify conflicting transactions at the level of granularity where they occur, usually without requiring repeated searches through numerous low-level locks.

It is important to note that a transaction which accesses any children of an object (such as rows of a table) must always acquire *some* lock on that object, even if it applies fine-grained locking further down the hierarchy. This centralized coordination introduces a point of contention for transactions which access different children of the same object, as they must acquire intention locks on it. Reading and updating internal state of hot locks becomes a serialization point even though there is no logical conflict.

3.2 The Database Lock Manager

There are two main entities involved in database locking: the lock manager and transaction agents. The lock manager provides an interface for transactions to request, upgrade, and release locks. Behind the scenes it also ensures that transactions implicitly acquire proper intention locks, performs deadlock prevention and detection, and manages storage for lock state (there can easily be too many locks to store in memory at the same time). This section describes the lock manager found in Shore-MT [12]; we expect other lock manager implementations to be similar as they must provide the same services.[1] Figure 2 depicts the internal data structures of the Shore-MT lock manager. Every active lock in the system is represented by a *lock head* data structure which contains the lock's current state, the head of a linked list of current *lock requests*, and a latch which protects both lock head and list elements. When a transaction requests a lock, the manager first ensures the transaction holds higher-level intention locks, requesting them automatically if necessary. If an appropriate coarse-grained lock is found the request can be granted immediately; otherwise the manager probes an internal hash table to find the desired

1. For example, the DB2 documentation describes space requirements of the first vs. subsequent requests to each lock in a way that strongly suggests it also uses a linked list of lock requests.

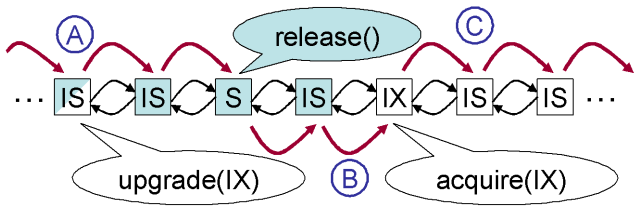

Figure 3. Lock release example

lock head. It then latches the lock head, appends a new request object to the request queue, and blocks the transaction if the request is incompatible with the lock's current holder. It then unlatches the lock head and returns the (granted) request to the transaction. If the transaction requests an upgrade for an existing lock, the lock manager finds and updates the existing request (again, possibly blocking the transaction).

In order to facilitate lock release at transaction completion, each transaction agent maintains a private list of requests for all locks it holds, in the order it acquired them. At transaction completion, the transaction repeatedly removes the youngest request from its private list and passes it back to the lock manager for release. The lock manager latches the corresponding lock head, unlinks the request from the lock queue, and searches the queue for other requests which may now be granted.

The effort required to grant or release a lock grows with the number of active transactions because of repeated searches within the lock queues. For example, Figure 3 shows a realistic lock request list for an S-lock held by a transaction; blue requests are granted and white ones are waiting. To release the lock the transaction must traverse the list from the beginning (A) to determine the new lock mode and identify any upgrade requests which can now be granted. In this case it identifies at least one upgrade to grant: `IS => IX`. Once all pending upgrades have been satisfied, the next waiting (new) request can be granted (B) if compatible, which in this case it is. All compatible requests directly after the first (C) can also be granted (at least two more in this example). Heavily accessed locks such as table locks will have many requests and upgrades in progress at any given point, and each release operation must traverse the list. Deadlock detection usually results in further list traversals to identify dangerous transactions which must be aborted to guarantee forward progress in the system.

Lock queues impose high overhead as the number of active transactions in the system increases both because more threads contend to access the list simultaneously and because the longer queues require more work per critical section. Intention locks tend to worsen the problem because they allow many transactions to hold the lock simultaneously, and they also introduce many upgrade requests as transactions change from IS to IX to modify records in the hierarchy. In contrast, releasing an exclusive request requires looking at only one request (the holder's successor). Combined, these costs lead to the spiralling overheads shown in Figure 1.

4. SPECULATIVE LOCK INHERITANCE

Speculative lock inheritance is based on the key observation that a scalable application will nearly always request hot (often-accessed) locks in a compatible mode; otherwise a significant fraction of transactions in the system would be blocked on the (exclusive) lock at any given moment and not competing to update the lock's internal state. If most transactions request the hot lock in a shared mode, a transaction could hold the lock for a significant length of time without reducing available concurrency as a result. The lock is only necessary to guarantee that rare transactions which do make updates can exclude other transactions properly.

Speculative lock inheritance exploits the lack of logical contention for hot, shared database locks in order to virtually eliminate physical contention for internal lock state. SLI allows a completing transaction to pass on some locks which it acquired to transactions which follow. This avoids a pair of release and acquire calls to the lock manager for each such lock. During the lock release phase of transaction commit, the transaction's agent thread identifies promising candidate locks and places them in a thread-local lock list instead of releasing them. It then initializes the next transaction's lock list with these previously acquired locks hoping that the new transaction will use some of them. Successful speculation improves performance in two ways. First, a transaction which inherits useful locks makes fewer lock requests, with corresponding lower overhead and better response time; short transactions amortize the cost of the lock acquire over many row accesses instead of just one. Second, other transactions which do request the lock will face less contention in the lock manager.

4.1 Extensions to the Lock Manager

When a transaction attempts to release a lock the lock manager determines whether it is a good candidate for inheritance (see next section), and if so, does not remove the request from the lock queue. Instead, it changes the request status from *granted* to *inherited* and moves it from the transaction's private list to a different private list owned by the transaction's agent thread. When the agent thread executes its next transaction, it pre-populates the new transaction's lock cache with the inherited locks. The speculation succeeds if the new transaction attempts to request an inherited lock: it will find the request already in its cache, update its status from *inherited* back to *granted*, and add it to its lock list as if it had just acquired it. The status update uses an atomic compare-and-swap operation and does not require calling into the lock manager, allocating requests, or updating latch-protected lock state. Inheritance fails harmlessly if the transaction does not make use of the lock(s) it inherited: they do not cause overhead during transaction execution and the transaction simply releases them at commit time along with the locks it did use. If another transaction encounters an inconvenient *inherited* lock request and an atomic compare-and-swap to *invalid* state succeeds, it simply unlinks the request from the queue and continues. Future attempts to reclaim the lock will fail, and the next time the owning agent completes a transaction it will deallocate any invalid requests it finds.

Lock inheritance is a very lightweight operation regardless of whether it eventually succeeds or not. In the worst case a transaction does not use the lock it inherited, and pays the cost of releasing the lock which the previous transaction avoided. Both invalidations and garbage collection are performed only when a transaction is already traversing the queue and add only minimal overhead. In the best case the lock manager will be completely relieved of requests for hot locks, with a corresponding boost to performance.

4.2 Criteria for Inheriting Locks

Our speculative lock inheritance scheme uses the following five criteria to identify candidates which are likely to benefit subsequent transactions with minimal risk of reducing concurrency:

1. The lock is page-level or higher in the hierarchy
2. The lock is "hot" (i.e. contention for the latch protecting it)
3. The lock is held in a shared mode (e.g. S, IS, IX)
4. No other transaction is waiting on the lock
5. The previous conditions also hold for the lock's parent, if any

The first two criteria favor locks which are likely to be reused by a subsequent transaction. Low-level locks such as row locks are so numerous that the overhead of tracking them outweighs the benefits, while a lock which has only one outstanding request at release time is unlikely to have another request arrive in the near future. We detect a "hot" lock by tracking what fraction of the most recent several acquires encountered latch contention and enabling SLI when the ratio crosses a tunable threshold.

The second two rules ensure lock inheritance does not hurt performance or concurrency, while the last rule ensures that lock inheritance maintains the hierarchical locking protocol. Though database designers are often willing to make sacrifices in consistency or other areas if it improves performance [7][13], speculative lock inheritance will be most useful if it does not change transaction consistency semantics or introduce other anomalies. It must be transparent and automatic, and impose minimal performance penalty for unsuccessful speculations.

4.3 Ensuring Correctness

SLI preserves consistency semantics by only passing share-mode locks from one transaction to another. Assuming the first transaction acquired its locks in a consistent way, the new transaction will inherit consistent locks. In addition, shared locks ensure that the previous transaction did not change the corresponding data objects.

From the perspective of a new transaction, an inherited lock request looks just like a new request that happened to be granted with no intervening updates since it was last released. Two-phase locking semantics are preserved because the inheritance is not finalized until the new transaction actually requests the lock. If an exclusive request arrives before then it invalidates the inheritance and the inheriting transaction must make a normal request. Therefore, from a semantic perspective an inherited lock was released and reacquired; only the underlying implementation has changed. From the perspective of both the inheriting and any competing transactions which arrive after the original transaction completes, the request was granted in the same order it would have been had SLI not intervened. A mixture of inherited and non-inherited locks is consistent and serializable for the same reasons. SLI preserves the hierarchical locking protocol by only inheriting locks whose parents are also eligible. Any inherited lock "orphaned" when its parent is invalidated will also be invalidated before any transaction tries to use it, thus avoiding the case where a low-level lock is held without appropriate locks on its ancestors.

A transaction could also potentially acquire locks in a different order than expected if it inherits locks which it would have

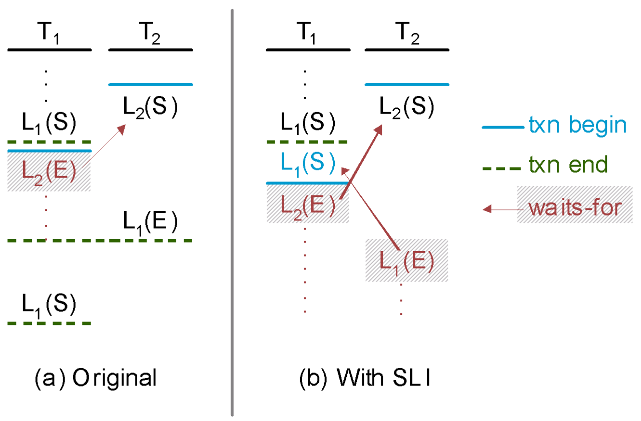

Figure 4. Example of potential SLI-induced deadlock

requested later than the beginning of the transaction. For example, Figure 4 shows how SLI could potentially induce deadlocks between transactions that are otherwise well-behaved. During normal execution (left), transaction agents T1 and T2 both acquire lock L2 followed by L1. Whichever agent requests the lock second has to wait until the other commits its current transaction, but no deadlock is possible. However, enabling SLI (right) allows T1 to inherit L1 from a previous transaction. If agents could not invalidate inherited but not-yet-used locks, T1 would have effectively acquired its locks in reverse order and could deadlock with T2.

Fortunately, SLI-induced deadlocks can easily be avoided because the transaction must still reclaim the lock before accessing the data; if any exclusive requests arrive for an inherited lock before the inheriting transaction first requests it, the lock manager can invalidate the inheritance. Once the request has been reclaimed the transaction has effectively acquired the lock in its natural order and conflicting requests will have the same risk of deadlock as in the unmodified system.

4.4 Non-uniform Locking Patterns

One minor concern with SLI is that, for real-world workloads, it will not achieve its full potential due to locking patterns which interfere with normal operations. We discuss two such patterns here: the roving hotspot and the bimodal workload.

Many workloads do not access data uniformly over time. Instead, the object of interest shifts, and contention with it. A common example is a table (such as a history or log) with heavy append traffic. For a given page of the table, for instance, high contention will disappear as soon as the page fills and transactions begin inserting records in a different page. This moving target presents two potential difficulties for SLI. First, old (now unnecessary) locks might start to pollute transaction caches and lock lists, and waste space in the lock manager's hash table as well. Second, newly hot locks will not be inherited at first, leading to contention. Fortunately, neither problem occurs in practice because SLI has a short memory: if transactions do not use inherited locks their agent thread will release them quickly; if new sources of contention appear, SLI will quickly begin inheriting the problematic locks.

A bimodal workload consists of two groups of transactions which access very different sets of locks. If the distribution of transactions to agent threads is random, a high fraction of transactions will not utilize the locks they inherited (from a previous, different transaction type), causing the lock manager to stop inheritance even though it would be beneficial to continue. There are several potential ways to make SLI resistant to this sort of workload:

1. Identify groups of transaction types which acquire similar locks, and bias the assignment of transactions to agent threads so that similar transactions execute with the same agent(s) most of the time. This approach would require either application developer assistance or some form of cluster identification based on observing which high-level locks each transaction type tends to acquire.
2. Apply a small hysteresis or momentum which prevents the lock manager from dropping inheritance just because one transaction did not use the lock. This approach is straightforward and inexpensive to implement using only local knowledge, but would tend to increase the number of useless locks which pass between transactions.
3. Do nothing. The fewer locks in common different transaction types acquire, the less contention their requests will cause and the less opportunity SLI has in the first place. Additionally, because contention tends to grow quadratically,[1] even a minor reduction in the number of threads competing for a lock request provides a significant improvement.

With our current hardware setup and benchmarks we find that the third approach — do nothing — works well in practice, though as the number of cores per chip continues to increase, contention may grow to the point that only a subset of the total threads are required to cause significant contention.

5. EXPERIMENTAL METHODOLOGY

We evaluate several benchmark transactions on highly concurrent multicore machine to identify both the opportunity for, and the effectiveness of, lock inheritance. We make use of three metrics to determine the effectiveness of SLI. First, we consider the numbers and types of locks which are responsible for contention, using software counters. Second, we use Sun's profiling tools to identify bottlenecks by their time breakdowns. Finally, we measure system throughput for several short transactions and transaction mixes to quantify the performance impact of SLI.

It is important to note that profiling a parallel machine can be somewhat counterintuitive because it measures work, not time. Both time and the number of hardware contexts in the system influence the amount of work which the system can perform. For example, Figure 5 spans 5 hardware contexts and 15 seconds, for a total of 75 cpu-sec of potential work. Offered load and the severity of any bottlenecks determines how much work is actually performed. In the example above, two daemon threads spend most of the time asleep while two other threads serialize at some critical section; only one thread remains busy all the time. Contention further complicates matters because excessive spinning results in a system which is fully utilized but does not produce the throughput

1. If N threads all contend for the same object, each can expect to wait for N/2 threads, for $O(N^2)$ total time wasted blocking or spinning.

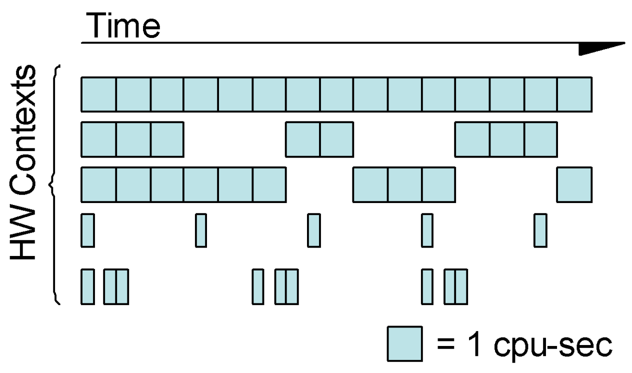

Figure 5. Simplified example of profiler output

expected, while blocking results in an underutilized system in spite of there being many live threads.

Using the profiler we can identify how much work was actually performed by the system during a measurement, as well as how much of that work was wasted due to contention. Further, we can filter the output to isolate the components of the storage manager responsible for both blocking- and spinning-based contention.

5.1 Benchmark Descriptions

We evaluate SLI using a set of ten transactions taken from three benchmarks: Nokia's Network Database Benchmark [15] ("NDBB," also known as "TM1"), TPC-B [20], and TPC-C [21].

NDBB is a telecom workload benchmark originally developed by Nokia to evaluate offerings by telecom vendors. It consists of seven transactions, operating on four large database tables, which implement various Home Location Register operations ("HLR") executed by mobile networks during cell phone calls and other common events such as call forwarding. The transactions are extremely short, usually accessing only 1-4 database rows, and must execute with very low latency even under extreme load. The benchmark is unusual in that many transactions fail due to invalid inputs (25-75%). Three of the transactions are read-only while the other four perform updates; we evaluate the specified transaction mix as well as individual transactions; we do not evaluate the call forwarding transactions individually because they change the database's distribution and quickly approach 0% success rate:

- GET_SUBSCRIBER_DATA ("**getSub**" read-only, 35% of mix, 0% fail). Retrieves subscriber and location information.
- GET_NEW_DESTINATION ("**getDest**" - read-only, 10% of mix, 76.1% fail). Retrieves the current call forwarding destination for a subscriber, if any.
- GET_ACCESS_DATA ("**getAccess**" - read-only, 35% of mix, 37.5% fail). Returns subscriber access validation data.
- UPDATE_SUBSCRIBER_DATA ("**updateSub**" - update, 2% of mix, 37.5% fail). Updates a subscriber's profile.
- UPDATE_LOCATION ("**updateLoc**" - update, 14% of mix, 0% fail). Updates the current location of a subscriber.
- INSERT_CALL_FORWARDING (update, 2% of mix, 68.75% fail). Adds a call forwarding destination.
- DELETE_CALL_FORWARDING (update, 2% of mix, 68.75% fail). Removes a call forwarding destination.
- "**Forward mix**." 71.4/28.5/28.5% mix of the getDest, INSERT_CALL_FORWARDING, and DELETE_CALL_-FORWARDING transactions.
- "**NDBB Mix**." Full transaction mix, specified frequencies.

TPC-B is a database stress test consisting of a single transaction type: customer deposits/withdrawals at the branches of a hypothetical bank. Like NDBB it defines four database tables, but unlike NDBB each transaction accesses all four of them.

TPC-C models an online transaction processing database for a retailer. It consists of five transactions which follow customer orders from initial creation to final delivery and payment:

- New Order (update, 45%) inserts a new sales order into the database. It is a medium-weight transaction with 1% failure rate due to invalid inputs.
- Payment (update, 43% of mix) makes a payment on an existing order. It is a short transaction (though still larger than any from the other two benchmarks).
- Order Status (read-only, 4%) computes the shipping status an order's line items. It is somewhat larger than Payment.
- Delivery (update, 4%) is the largest update transaction and also the most contentious.
- Stock Level (read-only, 4%) is a medium-sized query which examines roughly 200 order line items and their corresponding stock entries.
- "Small Mix" combines Payment, New Order, and Order Status in a 46.7/48.9/4.3% mix.
- "TPC-C Mix' combines all five TPC-C transactions at their specified frequencies.

We primarily use the "small mix" in for our analysis of SLI because we are particularly interested in the three smaller transactions — Payment, Order Status, and New Order — which together comprise 92% of the workload. Delivery is very contention-prone and tends to serialize transactions while Stock Level is very large and amortizes the cost of high-level locks across more than 1000 low-level locks; neither causes significant contention within the lock manager with the number of hardware contexts available today. As the degree of parallelism continues to increase, however, the size of transaction which suffers contention in the lock manager will also increase.

5.2 System Configuration

We perform all our experiments on a Sun T5220 "Niagara II" machine running Solaris 10. The Niagara II chip contains 8 cores, each capable of supporting 8 hardware contexts, for a total of 64 OS-visible "CPUs." Each core has two execution pipelines which accept instructions from any two threads simultaneously. 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 continue to double.

For a database engine we use Shore-MT [12], a version of the Shore storage manager [4] which has been modified to provide

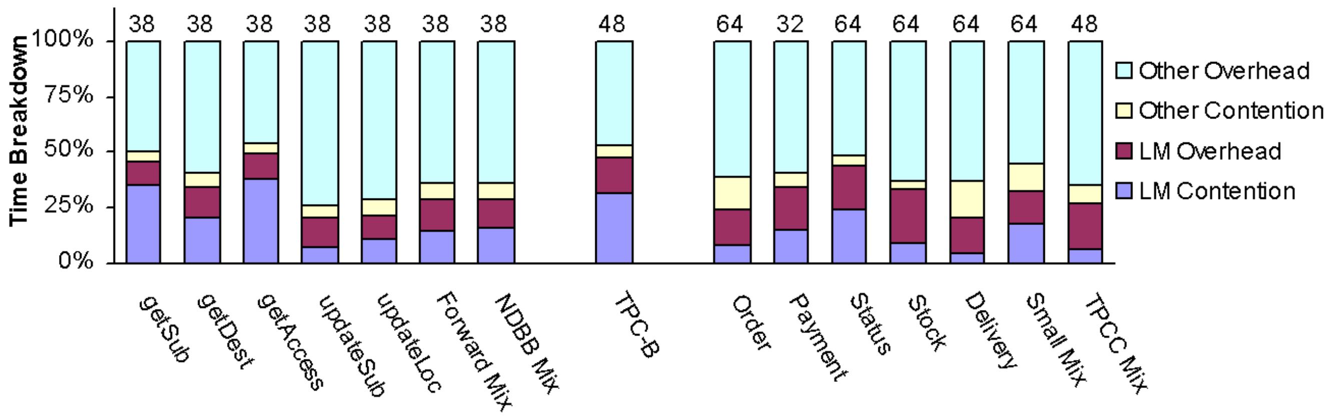

Figure 6. Execution time breakdowns at peak performance for NDBB Mix, TPC-B, and TPC-C

scalable performance on modern multicore systems. We use Shore-MT because it scales on highly parallel hardware, is open-source so we can easily modify the lock manager, and behaves like commercial database engines at the micro-architectural level [1]. We implemented SLI as a straightforward extension to the lock manager's existing behavior. The changes affected three source files and three headers (out of 461), and about 500 lines of added or changed code (out of 175kloc).

Because Shore does not have a SQL front end, all transactions have been partially hard-coded — the database metadata and back-end processing are schema-agnostic and general purpose, but the transaction code is schema-aware. This arrangement is similar to the statically compiled stored procedures which commercial engines support, converting annotated C code into a compiled object which is bound to the database and executed directly. For example, DB2 allows the developer to generate compiled "external routine" in a shared library for the engine to dlopen and execute directly within the engine's core for maximum performance [8].

The Niagara II is capable of very high performance, and demand on the I/O subsystem scales with throughput due to dirty page evictions and log writes. For the random I/O generated by transaction processing workloads, hundreds or even thousands of disks may be necessary to meet the demand. We therefore decouple I/O subsystem performance from the measurements by storing the database on an in-memory file system and modifying Shore to impose a 6 msec penalty for each I/O operation. The artificial delay simulates a high-end disk array having many spindles, such that all requests can proceed in parallel but must each still pay the cost of a disk seek. This arrangement is somewhat pessimistic because it assumes every access requires a full seek even if there is some sequential component to the access pattern, but it ensures that all aspects of the storage manager are exercised.

We evaluate both in-memory and "disk-resident" datasets; a 100,000 subscriber NDBB dataset occupies only a few hundred MB and operates entirely in memory; a 1000 TPC-B branches consume 20GB, simulating a balanced workload, and our 300 warehouse TPC-C configuration requires 40GB — too large to replicate the data both in memory and on disk, simulating a disk-resident workload. All dataset sizes can support enough concurrent requests to saturate the machine

To control the number of hardware contexts utilized for each measurement we bind the database engine to a Solaris processor set of the appropriate size, spreading threads over as many cores as possible. We then choose the degree of request concurrency that maximized performance for each processor set. For each run, the benchmark harness spawns clients and allows them to start working, measures throughput over several 30 second intervals, then notifies the clients to stop.

6. SLI OPPORTUNITY ANALYSIS

This section presents an opportunity analysis to demonstrate the potential for SLI to improve performance. The analysis includes two components. We first profile our system under high load to produce a breakdown of overheads arising out of the lock manager, giving an upper bound on the performance improvement SLI could achieve. We also examine the number and types of locks acquired by transactions to verify that SLI can exploit shared locks to reduce contention.

6.1 Lock Manager Overhead and Contention

In order to identify the magnitude of contention within the lock manager, we profile each of the transactions and transaction mixes discussed in the previous section. We use the experimental methodology outlined in Section 5, then plot the profiler breakdown for the number of clients which maximizes performance (usually 32-48 clients). Figure 6 shows the normalized work breakdown extracted from the profiler output at peak throughput achieved by each transaction (mix). The number of hardware contexts utilized is shown at the top of each bar — note that many transactions peak long before utilizing all 64 available contexts! Each column in the graph shows the fraction of cpu time a transaction spent in both work and contention, both inside and outside the lock manager. The results confirm that the lock manager is a large bottleneck in the system, especially for the smallest transactions such as those from NDBB. As expected, the largest TPC-C transactions do not suffer from the lock manager bottleneck: Stock queries a large amount of data and thus amortizes the cost of acquiring high-level locks; Delivery is not only large but also introduces true lock contention which blocks transactions so they do not compete for the lock manager. The measured lock manager overheads range between 10-20%, corroborating the results in [6].

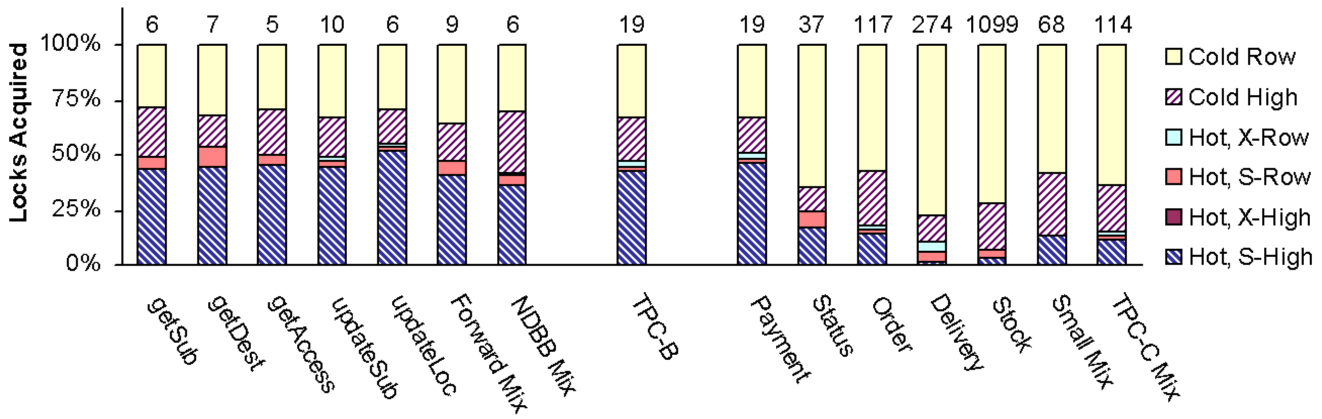

Figure 8. Breakdown of SLI-related characteristics for locks acquired by transactions

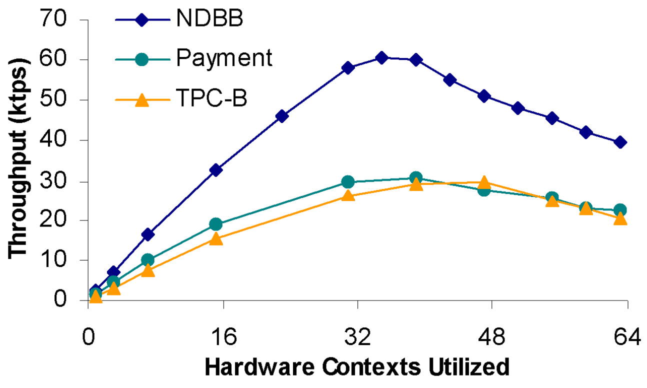

Figure 7. Impact of lock manager bottleneck as load varies.

The profiler results also indicate that the lock manager bottleneck is smaller for mixes of transactions which access the widest variety of tables (NDBB and TPC-C mix), though transaction size has a far stronger effect. Mixing different transactions together reduces the bottleneck for two reasons: different access patterns spread contention over more types of locks and agents running long transactions spend less time in the lock manager, easing pressure. For the workloads with small transactions we expect the bottleneck to grow over time as more cores per chip allow multiple different hotspots in the lock manager at the same time. Distributing hotspots over multiple tables will not eliminate contention in the long run because, even if the number of heavily-accessed tables in a workload grows over time, we do not expect it to grow uniformly or nearly as fast as core counts.

Figure 7 illustrates the performance impact of the lock manager bottleneck as we increase the load on the system along the x-axis from near-idle to saturated. Each data series shows the throughput achieved at different CPU utilizations for the NDBB mix, as well as the TPC-B and TPC-C Payment transactions. For small numbers of hardware contexts we see that the system scales well, with throughput increasing nearly linearly. However, as the number of hardware contexts increases past 32 contexts the lock manager bottleneck begins to impact performance, and by 48 contexts the bottleneck becomes severe enough that throughput starts to drop — the system is unable to utilize effectively the additional processing power available to it.

6.2 Opportunity for Lock Inheritance

Two conditions must hold in order for SLI to improve system scalability: most contention in the system should center around the internal state of hot locks, and those locks must be *heritable*, meeting the conditions that allow SLI to pass them between transactions. The previous section illustrates that, for short transactions, the lock manager is indeed the primary source of contention in the system. We now analyze lock access patterns to evaluate the opportunity for SLI to reduce that contention. The analysis considers three characteristics: hot vs. cold lock, shared vs. exclusive requests and row-level locks vs. those higher in the hierarchy. SLI targets hot, shared-mode, high-level locks. We are not interested in cold locks because do not cause contention within the lock manager, SLI cannot work with exclusive lock modes because it would impact concurrency, and we hypothesize that hot, shared-mode row-level locks are too rare to be worth considering. Therefore, SLI will have the most potential to improve performance if a large fraction of locks are hot, shared, and high-level; and if most remaining locks are cold. We note that it is entirely possible for many transactions to wait on “cold” locks, especially in badly-behaved workloads. However, true lock contention serializes transactions, and the resulting low concurrency reduces contention for the lock’s internal state, making SLI unnecessary.

Figure 8 shows a breakdown of the types of locks acquired by each transaction or transaction mix; the number at the top of each column is the average number of locks acquired per transaction. SLI targets locks which are both hot and heritable. Any hot locks which remain cannot be addressed by SLI and ideally contribute a small fraction of the total. As expected, the smallest transactions acquire few locks but most of those locks are heritable and many are hot. As transactions acquire more and more locks the number of hot and heritable locks does not increase as quickly, indicating lower contention in the lock manager and less opportunity for SLI. We observe that, for the workloads analyzed here, there are very few, if any hot non-heritable locks and that transactions with the most hot and heritable locks also experience the highest contention in the lock manager in Figure 7. Together these indicate that SLI has the potential to reduce or eliminate the lock manager bottleneck. We note that, though there are relatively few hot and heritable locks in

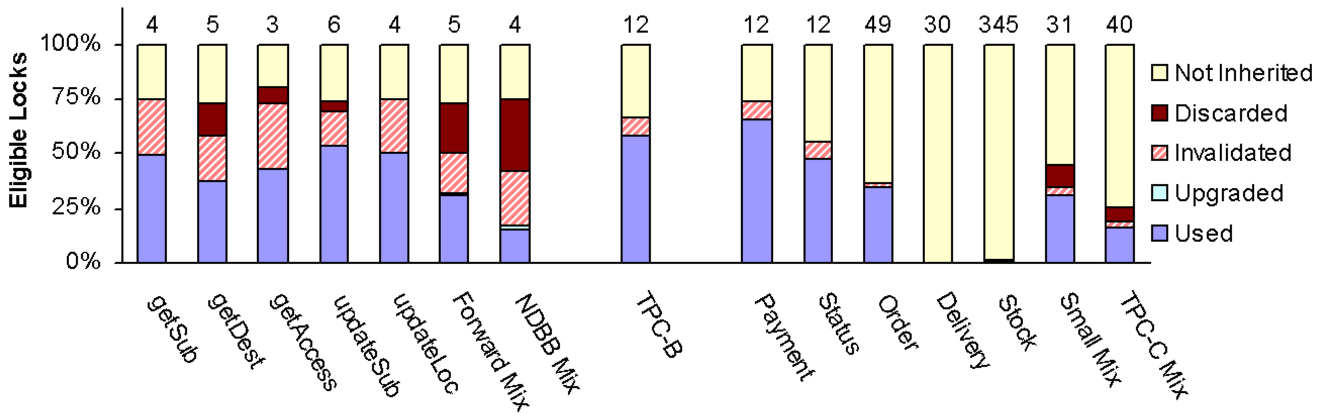

Figure 9. Breakdown of outcomes for locks which SLI could choose to pass between transactions.

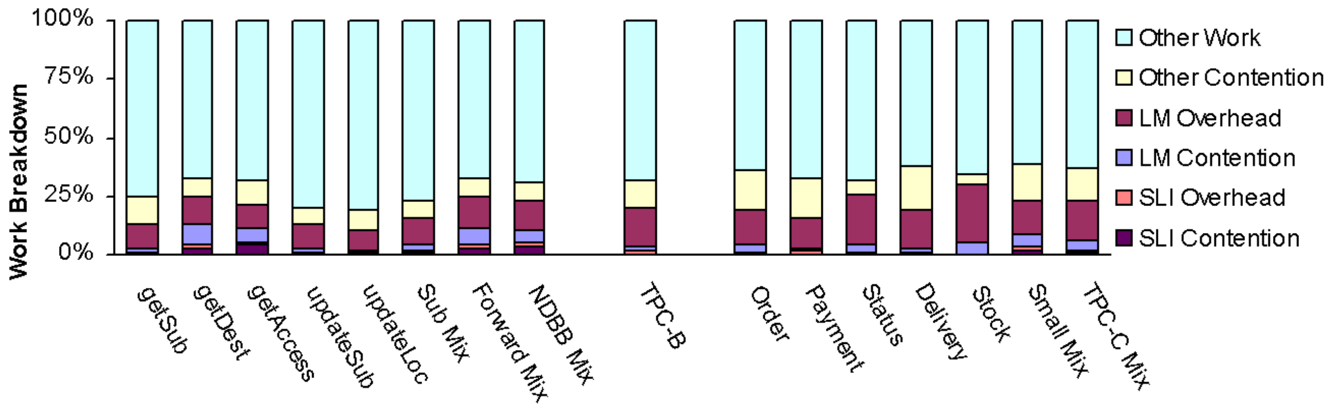

Figure 10. Breakdown of transaction execution time on loaded system with SLI enabled (64 contexts utilized).

the breakdown, even a few transactions inheriting them (and avoiding the lock manager) will have a disproportionate impact in reducing contention; row locks, though numerous, are not usually hot, and even less often both hot and shared.

7. PERFORMANCE ANALYSIS

In this section we quantify the performance improvement that SLI gives for transactional workloads under various conditions, and analyze the source of those improvements. We expect that SLI will work best when many small and (usually) non-conflicting transactions execute simultaneously on many hardware contexts; workloads with low load, or with transactions which are large or conflicting, will not benefit nearly as much.

While the goal of SLI is to eliminate contention within the lock manager so it does not impede scalability. It might also reduce transaction overhead by avoiding calls into the lock manager. We observe this effect to be negligible (< 4%) for even the shortest transactions, given the fraction of locks which are never inherited.

7.1 Effectiveness of Lock Inheritance

We first examine the effectiveness of SLI in passing locks between transactions. When inheritance is effective most hot locks in the system are inherited and used by succeeding transactions. SLI will not eliminate fully the lock manager bottleneck if hot locks cannot be inherited, remain unused and are discarded, or are invalidated before a transaction can reclaim them.

Figure 9 shows the breakdown of outcomes for hot locks in the system for each transaction and mix. SLI is selective, passing only hot locks between transactions. For shorter transactions most locks are hot, though a significant fraction of them are invalidated and cannot be used; the longest transactions have virtually no hot locks because they acquire so many that relatively little time per transaction goes to any one request. We also note that mixing multiple transaction types increases the number of locks which are invalidated, and also increases the number of useless locks which transactions eventually discard. However, as we will see in the next section, the locks which are successfully inherited are also the ones responsible for most of the lock manager bottleneck.

7.2 Performance Impact of SLI

Figure 10 shows the work breakdown of transactions when SLI is active. Significantly, none of the transactions has a large contribution from lock manager contention any more. This indicates that SLI is effective in identifying and passing the locks which cause most lock manager contention. We also note that SLI has very low overhead — even in the worst case it adds only 5% overhead, usually with a corresponding decrease in lock manager overhead. For example, locks which are inherited but never used must still be released, and that overhead counts toward SLI, not the lock manager. For most transaction profiles contention in the lock manager was replaced by useful work, suggesting a significant performance improvement. However, the New Order transaction sees a shift of

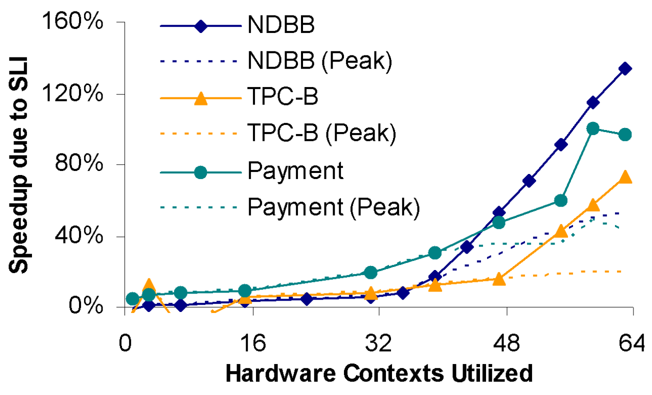

Figure 11. Performance improvement due to SLI

contention from the lock manager to other areas (mostly in Shore's free space manager). The two large TPC-C transactions are virtually unchanged by SLI (as expected) because they did not have a significant lock manager component to begin with.

Overall, the transactions spend 75% or more of the time doing useful work with SLI active, even though the system is now fully loaded (in contrast with Figure 6). For example, NDBB+SLI exhibits lower contention at 95% utilization than the baseline does at 60% utilization. SLI gives large speedup for this workload because it not only eliminates contention for existing load, but allows load to increase without the contention returning. We also note that SLI never reduces performance for any of the workloads.

Figure 11 compares performance of the baseline system with SLI. As expected, the short transactions benefit the most. Some, especially the larger transactions, see little or no performance improvement. Comparing against Figure 6 we can see that large transactions did not have lock manager contention component to start with, explaining their lack of improvement.

Some of the smaller transactions, such as New Order, which saw no speedup, had lock manager contention replaced by contention from other sources, indicating that bottlenecks other than the lock manager limit performance. If there were no other scalability bottlenecks in the system, we would expect SLI to improve performance even more than measured here.

8. CONCLUSIONS

In this paper we identify an important scalability challenge in database engines: hierarchical locking, which is vital for high application concurrency, forms a growing bottleneck, especially for small transactions such as those found in telecom and banking. We observe that the compatible nature of high-level intent locks means they can be safely retained across transactions in order to reduce pressure on the lock manager. We then show how Speculative Lock Inheritance yields throughput improvements of 10-40% for today's hardware, while fundamentally limiting the number of threads which are likely to request the same high-level lock simultaneously. Thus, even as core counts continue to grow we expect that lock inheritance will keep the lock manager off the critical path for transaction processing. Speculative Lock inheritance is an example of a low-effort, high-impact modification which significantly improves scalability within the lock manager.

9. REFERENCES

[1] Ailamaki, A., DeWitt, D. J., and Hill, M. D. Walking Four Machines By The Shore. In *Proc. CAECW*, 2001.

[2] Anon, et al. A measure of transaction processing power. *Datamation*, April 1, 1985.

[3] Brigde, W., Joshi, A., Keihl, M., Lahiri, T., Loaiza, J., and MacNaughton, N. The Oracle Universal Server Buffer. In *Proc. VLDB'97*, 1997.

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

[5] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, New York, NY, 1993.

[6] Harizopoulos, S., Abadi, D., Madden, S., and Stonebraker, M. OLTP under the looking glass, and what we found there. In *Proc. SIGMOD'08*, Vancouver, Canada, 2008.

[7] P. Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *Proc. CIDR'07*, Asilomar, CA, 2007.

[8] IBM. *IBM DB2 9.5 Information Center for Linux, UNIX, and Windows*. Available online at http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp

[9] IBM. SolidDB Product Family. See http://www-01.ibm.com/software/data/soliddb/

[10] MySQL AB. MySQL Reference Manual v5.0. See http://dev.mysql.com/doc/refman/5.0/

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

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

[13] Jorwekar, S., Fekete, A., Ramamritham, K., and Sudarshan, S. Automating the detection of snapshot isolation anomalies. In *Proc VLDB'07*, Vienna, Austria, 2007.

[14] Joshi, A. Adaptive locking strategies in a multi-node data sharing environment. In Proc. VLDB'91, Barcelona, 1991.

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

[16] Robinson, J. A fast general-purpose hardware synchronization mechanism. In *Proc SIGMOD'85*, Austin, Texas, 1985.

[17] SAP. *SAP Sales and Distribution Benchmark*. Description and results available online at http://www.sap.com/solutions/benchmark/sd.epx

[18] Stonebraker, M., Madden, S., Abadi, D., Harizopoulos, S., Hachem, N., and Helland, P. The End of an Architectural Era (It's Time for a Complete Rewrite). In Proc. VLDB, 2007.

[19] Transaction Processing Performance Council (TPC). *TPC Benchmark B: Standard Specification*. Available online at http://www.tpc.org/tpcb/spec/tpcb_current.pdf.

[20] Transaction Processing Performance Council (TPC). *TPC Benchmark C: Standard Specification*. Available online at http://www.tpc.org/tpcc/spec/tpcc_current.pdf.