Lightweight Locking for Main Memory Database Systems

Kun Ren[†*] kun@cs.yale.edu
Alexander Thomson[*] thomson@cs.yale.edu
Daniel J. Abadi[*] dna@cs.yale.edu
[†]Northwestern Polytechnical University, China [*]Yale University

ABSTRACT

Locking is widely used as a concurrency control mechanism in database systems. As more OLTP databases are stored mostly or entirely in memory, transactional throughput is less and less limited by disk IO, and lock managers increasingly become performance bottlenecks.

In this paper, we introduce *very lightweight locking* (VLL), an alternative approach to pessimistic concurrency control for main-memory database systems that avoids almost all overhead associated with traditional lock manager operations. We also propose a protocol called *selective contention analysis* (SCA), which enables systems implementing VLL to achieve high transactional throughput under high contention workloads. We implement these protocols both in a traditional single-machine multi-core database server setting and in a distributed database where data is partitioned across many commodity machines in a shared-nothing cluster. Our experiments show that VLL dramatically reduces locking overhead and thereby increases transactional throughput in both settings.

1. INTRODUCTION

As the price of main memory continues to drop, increasingly many transaction processing applications keep the bulk (or even all) of their active datasets in main memory at all times. This has greatly improved performance of OLTP database systems, since disk IO is eliminated as a bottleneck.

As a rule, when one bottleneck is removed, others appear. In the case of main memory database systems, one common bottleneck is the lock manager, especially under workloads with high contention. One study reported that 16-25% of transaction time is spent interacting with the lock manager in a main memory DBMS [8]. However, these experiments were run on a single core machine with no physical contention for lock data structures. Other studies show even larger amounts of lock manager overhead when there are transactions running on multiple cores competing for access to the lock manager [9, 18, 14]. As the number of cores per machine continues to grow, lock managers will become even more of a performance bottleneck.

Although locking protocols are not implemented in a uniform way across all database systems, the most common way to implement a lock manager is as a hash table that maps each lockable record's primary key to a linked list of lock requests for that record [4, 2, 3, 20, 7]. This list is typically preceded by a lock head that tracks the current lock state for that item. For thread safety, the lock head generally stores a mutex object, which is acquired before lock requests and releases to ensure that adding or removing elements from the linked list always occurs within a critical section. Every lock release also invokes a traversal of the linked list for the purpose of determining what lock request should inherit the lock next.

These hash table lookups, latch acquisitions, and linked list operations are main memory operations, and would therefore be a negligible component of the cost of executing any transaction that accesses data on disk. In main memory database systems, however, these operations are not negligible. The additional memory accesses, cache misses, CPU cycles, and critical sections invoked by lock manager operations can approach or exceed the costs of executing the actual transaction logic. Furthermore, as the increase in cores and processors per server leads to an increase in concurrency (and therefore lock contention), the size of the linked list of transaction requests per lock increases—along with the associated cost to traverse this list upon each lock release.

We argue that it is therefore necessary to revisit the design of the lock manager in modern main memory database systems. In this paper, we explore two major changes to the lock manager. First, we move all lock information away from a central locking data structure, instead co-locating lock information with the raw data being locked (as suggested in the past [5]). For example, a tuple in a main memory database is supplemented with additional (hidden) attributes that contain information about the row-level lock information about that tuple. Therefore, a single memory access retrieves both the data and lock information in a single cache line, potentially removing additional cache misses.

Second, we remove all information about which transactions have outstanding requests for particular locks from the lock data structures. Therefore, instead of a linked list of requests per lock, we use a simple semaphore containing the number of outstanding requests for that lock (alternatively, two semaphores—one for read requests and one for write-requests). After removing the bulk of the lock manager's

Proceedings of the VLDB Endowment, Vol. 6, No. 2

main data structure, it is no longer trivial to determine which transaction should inherit a lock upon its release by a previous owner. One key contribution of our work is therefore a solution to this problem. Our basic technique is to force all locks to be requested by a transaction at once, and order the transactions by the order in which they request their locks. We use this global transaction order to figure out which transaction should be unblocked and allowed to run as a consequence of the most recent lock release.

The combination of these two techniques—which we call *very lightweight locking* (VLL)—incurs far less overhead than maintaining a traditional lock manager, but it also tracks less total information about contention between transactions. Under high-contention workloads, this can result in reduced concurrency and poor CPU utilization. To ameliorate this problem, we also propose an optimization called *selective contention analysis* (SCA), which—only when needed—efficiently computes the most useful subset of the contention information that is tracked in full by traditional lock managers at all times.

Our experiments show that VLL dramatically reduces lock management overhead, both in the context of a traditional database system running on a single (multi-core) server, and when used in a distributed database system that partitions data across machines in a shared-nothing cluster. In such partitioned systems, the distributed commit protocol (typically two-phase commit) is often the primary bottleneck, rather than the lock manager. However, recent work on deterministic database systems such as Calvin [22] have shown how two-phase commit can be eliminated for distributed transactions, increasing throughput by up to an order of magnitude—and consequently reintroducing the lock manager as a major bottleneck. Fortunately, deterministic database systems like Calvin lock all data for a transaction at the very start of executing the transaction. Since this element of Calvin's execution protocol satisfies VLL's lock request ordering requirement, VLL fits naturally into the design of deterministic systems. When we compare VLL (implemented within the Calvin framework) against Calvin's native lock manager, which uses the traditional design of a hash table of request queues, we find that VLL enables an even greater throughput advantage than that which Calvin has already achieved over traditional nondeterministic execution schemes in the presence of distributed transactions.

2. VERY LIGHTWEIGHT LOCKING

The category of "main memory database systems" encompasses many different database architectures, including single-server (multi-processor) architectures and a plethora of emerging partitioned system designs. The VLL protocol is designed to be as general as possible, with specific optimizations for the following architectures:

- Multiple threads execute transactions on a single-server, shared memory system.
- Data is partitioned across *processors* (possibly spanning multiple independent servers). At each partition, a single thread executes transactions serially.
- Data is partitioned arbitrarily (e.g. across multiple machines in a cluster); within each partition, multiple worker threads operate on data.

The third architecture (multiple partitions, each running multiple worker threads) is the most general case; the first two architectures are in fact special cases of the third. In the first, the number of partitions is one, and in the second, each partition limits its pool of worker threads to just one. For the sake of generality, we introduce VLL in the context of the most general case in the upcoming sections, but we also point out the advantages and tradeoffs of running VLL in the other two architectures.

2.1 The VLL algorithm

The biggest difference between very lightweight locking and traditional lock manager implementations is that VLL stores each record's "lock table entry" not as a linked list in a separate lock table, but rather as a pair of integer values (C_X, C_S) immediately preceding the record's value in storage, which represent the *number* of transactions requesting exclusive and shared locks on the record, respectively. When no transaction is accessing a record, its C_X and C_S values are both 0.

In addition, a global queue of transaction requests (called `TxnQueue`) is kept at each partition, tracking all active transactions in the order in which they requested their locks.

When a transaction arrives at a partition, it attempts to request locks on all records at that partition that it will access in its lifetime. Each lock request takes the form of incrementing the corresponding record's C_X or C_S value, depending whether an exclusive or shared lock is needed. Exclusive locks are considered to be "granted" to the requesting transaction if $C_X = 1$ and $C_S = 0$ after the request, since this means that no other shared or exclusive locks are currently held on the record. Similarly, a transaction is considered to have acquired a shared lock if $C_X = 0$, since that means that no exclusive locks are held on the record.

Once a transaction has requested its locks, it is added to the `TxnQueue`. Both the requesting of the locks and the adding of the transaction to the queue happen inside the same critical section (so that only one transaction at a time within a partition can go through this step). In order to reduce the size of the critical section, the transaction attempts to figure out its entire read set and write set in advance of entering this critical section. This process is not always trivial and may require some exploratory actions. Furthermore, multi-partition transaction lock requests have to be coordinated. This process is discussed further in Section 3.1.

Upon leaving the critical section, VLL decides how to proceed based on two factors:

- Whether or not the transaction is **local** or **distributed**. A local transaction is one whose read- and write-sets include records that all reside on the same partition; distributed transactions may access a set of records spanning multiple data partitions.
- Whether or not the transaction successfully acquired all of its locks immediately upon requesting them. Transactions that acquire all locks immediately are termed **free**. Those which fail to acquire at least one lock are termed **blocked**.

VLL handles each transactions differently based on whether they are free or blocked:

- **Free transactions** are immediately executed. Once completed, the transaction releases its locks (i.e. it

decrements every C_X or C_S value that it originally incremented) and removes itself from the `TxnQueue`[1]. Note, however, that if the free transaction is distributed then it may have to wait for remote read results, and therefore may not complete immediately.

- **Blocked transactions** cannot execute fully, since not all locks have been acquired. Instead, these are tagged in the `TxnQueue` as blocked. Blocked transactions are not allowed to begin executing until they are explicitly unblocked by the VLL algorithm.

In short, all transactions—free and blocked, local and distributed—are placed in the `TxnQueue`, but only free transactions begin execution immediately.

Since there is no lock management data structure to record which transactions are waiting for data locked by other transactions, there is no way for a transaction to hand over its locks directly to another transaction when it finishes. An alternative mechanism is therefore needed to determine when blocked transactions can be unblocked and executed. One possible way to accomplish this is for a background thread to examine each blocked transaction in the `TxnQueue` and examine the C_X and C_S values of each data item for which the transaction requested a lock. If the transaction incremented C_X for a particular item, and now C_X is down to 1 and C_S is 0 for that item (indicating that no other active transactions have locked that item), then the transaction clearly has an exclusive lock on it. Similarly, if the transaction incremented C_S and now C_X is down to 0, the transaction has a shared lock on the item. If all data items that it requested are now available, the transaction can be unblocked and executed.

The problem with this approach is that if another transaction entered the `TxnQueue` and incremented C_X for the same data item that a transaction blocked in the `TxnQueue` already incremented, then both transactions will be blocked forever since C_X will always be at least 2.

Fortunately, this situation can be resolved by a simple observation: a blocked transaction that reaches the front of the `TxnQueue` will always be able to be unblocked and executed—no matter how large C_X and C_S are for the data items it accesses. To see why this is the case, note that each transaction requests all locks and enters the queue all within the same critical section. Therefore if a transaction makes it to the front of the queue, this means that all transactions that requested their locks before it have now completed. Furthermore, all transactions that requested their locks after it will be blocked if their read and write set conflict.

Since the front of the `TxnQueue` can always be unblocked and run to completion, every transaction in the `TxnQueue` will eventually be able to be unblocked. Therefore, in addition to reducing lock manager overhead, this technique also guarantees that there will be no deadlock within a partition. (We explain how distributed deadlock is avoided in Section 3.1.) Note that a blocked transaction now has two ways to become unblocked: either it makes it to the front of the queue (meaning that all transactions that requested locks before it have finished completely), or it becomes the only transaction remaining in the queue that requested locks on each of the keys in its read-set and write-set. We discuss a more sophisticated technique for unblocking transactions in Section 2.5.

[1]The transaction is not required to be at the front of the `TxnQueue` when it is removed. In this sense, `TxnQueue` is not, strictly speaking, a queue.

```
// Requests exclusive locks on all records in T's
// WriteSet and shared locks on all records in T's ReadSet.
// Tags T as free iff ALL locks requested were
// successfully acquired.
function BeginTransaction(Txn T)
    <begin critical section>
    T.Type = Free;
    // Request read locks for T.
    foreach key in T.ReadSet
        data[key].Cs++;
        // Note whether lock was acquired.
        if (data[key].Cx > 0)
            T.Type = Blocked;
    // Request write locks for T.
    foreach key in T.WriteSet
        data[key].Cx++;
        // Note whether lock was acquired.
        if (data[key].Cx > 1 OR data[key].Cs > 0)
            T.Type = Blocked;
    TxnQueue.Enqueue(T);
    <end critical section>

// Releases T's locks and removes T from TxnQueue.
function FinishTransaction(Txn T)
    <begin critical section>
    foreach key in T.ReadSet
        data[key].Cs−−;
    foreach key in T.WriteSet
        data[key].Cx−−;
    TxnQueue.Remove(T);
    <end critical section>

// Transaction execution thread main loop.
function VLLMainLoop()
    while (true)
        // Select a transaction to run next...
        // First choice: a previously-blocked txn
        // that now does not conflict with older txns.
        if (TxnQueue.front().Type == Blocked)
            Txn T = TxnQueue.front();
            T.Type = Free;
            Execute(T);
            FinishTransaction(T);
        // 2nd choice: Start on a new txn request.
        else if (TxnQueue is not full)
            Txn T = GetNewTxnRequest();
            BeginTransaction(T);
            if (T.Type == Free)
                Execute(T);
                FinishTransaction(T);
```

Figure 1: Pseudocode for the VLL algorithm.

One problem that VLL sometimes faces is that as the `TxnQueue` grows in size, the probability of a new transaction being able to immediately acquire all its locks decreases, since the transaction can only acquire its locks if it does not conflict with any transaction in the entire `TxnQueue`.

We therefore artificially limit the number of transactions that may enter the `TxnQueue`—if the size exceeds a threshold, the system temporarily ceases to process new transactions, and shifts its processing resources to finding transactions in the `TxnQueue` that can be unblocked (see Sec-

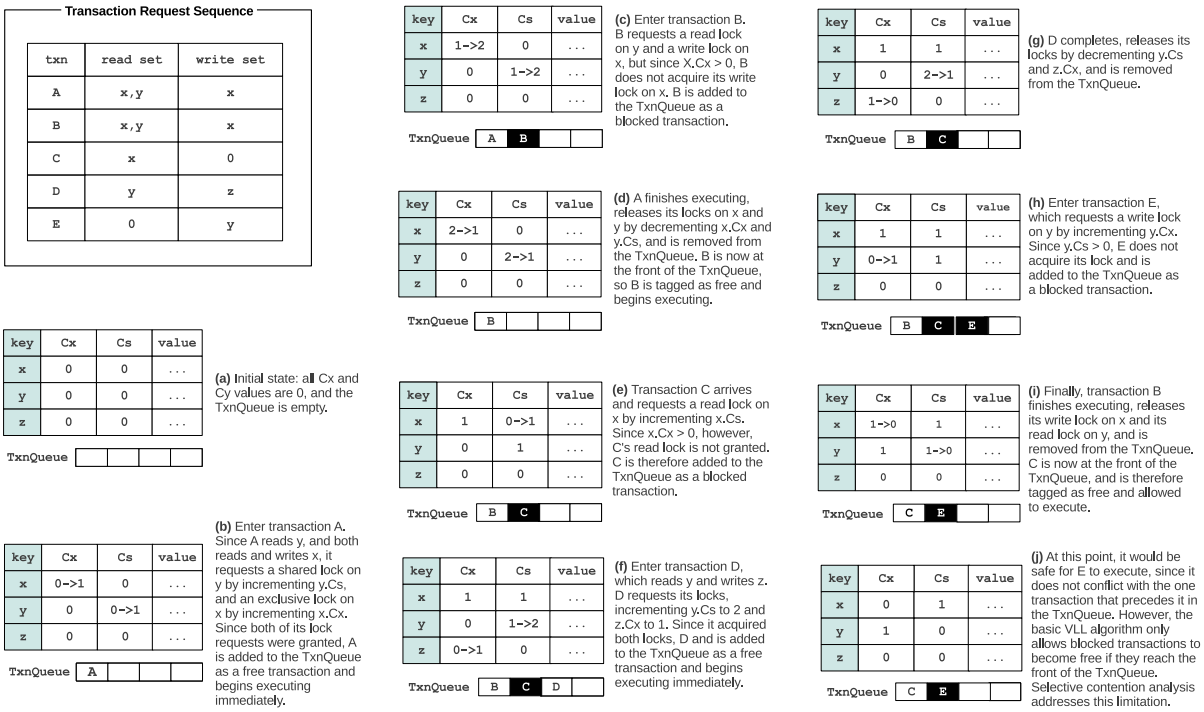

Figure 2: Example execution of a sequence of transactions $\{A, B, C, D, E\}$ using VLL. Each transaction's read and write set is shown in the top left box. Free transactions are shown with white backgrounds in the `TxnQueue`, and blocked transactions are shown as black. Transaction logic and record values are omitted, since VLL depends only on the keys of the records in transactions' read and write sets.

tion 2.5). In practice we have found that this threshold should be tuned depending on the contention ratio of the workload. High contention workloads run best with smaller `TxnQueue` size limits since the probability of a new transaction not conflicting with any element in the `TxnQueue` is smaller. A longer `TxnQueue` is acceptable for lower contention workloads. In order to automatically account for this tuning parameter, we set the threshold not by the size of the `TxnQueue`, but rather by the number of blocked transactions in the `TxnQueue`, since high contention workloads will reach this threshold sooner than low contention workloads.

Figure 1 shows the pseudocode for the basic VLL algorithm. Each worker thread in the system executes the `VLLMainLoop` function. Figure 2 depicts an example execution trace for a sequence of transactions.

2.2 Single-threaded VLL

In the previous section, we discussed the most general version of VLL, in which multiple threads may process different transactions simultaneously within each partition. It is also possible to run VLL in single-threaded mode. Such an approach would be useful in H-Store style settings [19], where data is partitioned across cores within a machine (or within a cluster of machines) and there is only one thread assigned to each partition. These partitions execute independently of one another unless a transaction spans multiple partitions, in which case the partitions need to coordinate processing.

In the general version of VLL described above, once a thread begins executing a transaction, it does nothing else until the transaction is complete. For distributed transactions that perform remote reads, this may involve sleeping for some period of time while waiting for another partition to send the read results over the network. If single-threaded VLL were implemented simply by running only one thread according to the previous specification, the result would be a serial execution of transactions, since no transaction would ever start until the previous transaction completed, and therefore no transaction could ever block on any other.

In order to improve concurrency in single-threaded VLL implementations, we allow transactions to enter a third state (in addition to "blocked" and "free"). This third state, "waiting", indicates that a transaction was previously executing but could not complete without the result of an outstanding remote read request. When a transaction triggers this condition and enters the "waiting" state, the main execution thread puts it aside and searches for a new transaction to execute. Conversely, when the main thread is looking for a transaction to execute, in addition to considering the first transaction on the `TxnQueue` and any new transaction requests, it may resume execution of any "waiting" transaction which has received remote read results since entering the "waiting" state.

There is also no need for critical sections when running in single-threaded mode, since only one transaction at a time attempts to acquire locks at any particular partition.

2.3 Impediments to acquiring all locks at once

As discussed in Section 2.1, in order to guarantee that the

head of the `TxnQueue` is always eligible to run (which has the added benefit of eliminating deadlocks), VLL requires that all locks for a transaction be acquired together in a critical section. There are two possibilities that make this nontrivial:

- The read and write set of a transaction may not be known before running the transaction. An example of this is a transaction that updates a tuple that is accessed through a secondary index lookup. Without first doing the lookup, it is hard to predict what records the transaction will access—and therefore what records it must lock.
- Since each partition has its own `TxnQueue`, and the critical section in which it is modified is local to a partition, different partitions may not begin processing transactions in the same order. This could lead to distributed deadlock, where one partition gets all its locks and activates a transaction, while that transaction is "blocked" in the `TxnQueue` of another partition.

In order to overcome the first problem, before the transaction enters the critical section, we allow the transaction to perform whatever reads it needs to (at no isolation) for it to figure out what data it will access (for example, it performs the secondary index lookups). This can be done in the `GetNewTxnRequest` function that is called in the pseudocode shown in Figure 1. After performing these exploratory reads, it enters the critical section and requests those locks that it discovered it would likely need. Once the transaction gets its locks and is handed off to an execution thread, the transaction runs as normal unless it discovers that it does not have a lock for something it needs to access (this could happen if, for example, the secondary index was updated immediately after the exploratory lookup was performed and now returns a different value. In such a scenario, the transaction aborts, releases its locks, and submits itself to the database system to be restarted as a completely new transaction.

There are two possible solutions to the second problem. The first is simply to allow distributed deadlocks to occur and to run a deadlock detection protocol that aborts deadlocked transactions. The second approach is to coordinate across partitions to ensure that multi-partition transactions are added to the `TxnQueue` in the same order on each partition.

We choose the second approach for our implementation in this paper. The reason is that multi-partition transactions are typically bottlenecked by the commit protocol (e.g. two-phase commit) that are run in order to ensure ACID-compliance of transactional execution. Reducing the lock manager overhead in such a scenario is therefore unhelpful in the face of a larger bottleneck. However, recent work on deterministic database systems [21, 22] shows that the commit protocol may be eliminated by performing all multi-partition coordination before beginning transactional execution. In short, deterministic database systems such as Calvin order all transactions across partitions, and this order can be leveraged by VLL to avoid distributed deadlock. Furthermore, since deterministic systems have been shown to be a particularly promising approach in main memory database systems [21], the integration of VLL and deterministic database systems seems to be a particularly good match.

2.4 Tradeoffs of VLL

VLL's primary strength lies in its extremely low overhead in comparison to that of traditional lock management approaches. VLL essentially "compresses" a standard lock manager's linked list of lock requests into two integers. Furthermore, by placing these integers inside the tuple itself, both the lock information and the data itself can be retrieved with a single memory access, minimizing total cache misses.

The main disadvantage of VLL is a potential loss in concurrency. Traditional lock managers use the information contained in lock request queues to figure out whether a lock can be granted to a particular transaction. Since VLL does not have these lock queues, it can only test more selective predicates on the state: (a) whether this is the *only* lock in the queue, or (b) whether it is *so* old that it is impossible for any other transaction to precede it in any lock queue.

As a result, it is common for scenarios to arise under VLL where a transaction cannot run even though it "should" be able to run (and would be able to run under a standard lock manager design). Consider, for example, the sequence of transactions:

txn	writeset
A	x
B	y
C	x, z
D	z

Suppose A and B are both running in executor threads (and are therefore still in the `TxnQueue`) when C and D come along. Since transaction C conflicts with A on record x and D conflicts with C on z, both are put on the `TxnQueue` in blocked mode. VLL's "lock table state" would then look like the following (as compared to the state of a standard lock table implementation):

VLL

key	Cx	Cs
x	2	0
y	1	0
z	2	0

TxnQueue: A, B, C, D

Standard

key	request queue
x	A, C
y	B
z	C, D

Next, suppose that A completes and releases its locks. The lock tables would then appear as follows:

VLL

key	Cx	Cs
x	1	0
y	1	0
z	2	0

TxnQueue: B, C, D

Standard

key	request queue
x	C
y	B
z	C,D

Since C appears at the head of all its request queues, a standard implementation would know that C could safely be run, whereas VLL is not able to determine that.

When contention is low, this inability of VLL to immediately determine possible transactions that could potentially

be unblocked is not costly. However, under higher contention workloads, and especially when there are distributed transactions in the workload, VLL's resource utilization suffers, and additional optimizations are necessary. We discuss such optimizations in the next section.

2.5 Selective contention analysis (SCA)

For high contention and high percentage multi-partition workloads, VLL spends a growing percentage of CPU cycles in the state described in Section 2.4 above, where no transaction can be found that is known to be safe to execute—whereas a standard lock manager would have been able to find one. In order to maximize CPU resource utilization, we introduce the idea of *selective contention analysis* (SCA).

SCA simulates the standard lock manager's ability to detect which transactions should inherit released locks. It does this by spending work examining contention—but *only* when CPUs would otherwise be sitting idle (i.e., `TxnQueue` is full and there are no obviously unblockable transactions). SCA therefore enables VLL to selectively increase its lock management overhead when (and only when) it is beneficial to do so.

Any transaction in the `TxnQueue` that is in the 'blocked' state, conflicted with one of the transactions that preceded it in the queue at the time that it was added. Since then, however, the transaction(s) that caused it to become blocked may have completed and released their locks. As the transaction gets closer and closer to the head of the queue, it therefore becomes much less likely to be "actually" blocked.

In general, the i^{th} transaction in the `TxnQueue` can only conflict now with up to $(i-1)$ prior transactions, whereas it previously had to contend with (up to) `TxnQueueSizeLimit` prior transactions. Therefore, SCA starts at the front of the queue, and works its way through the queue looking for a transaction to execute. The whole while, it keeps two bit-arrays, D_X and D_S, each of size 100kB (so that both will easily fit inside an L2 cache of size 256kB) and initialized to all 0s. SCA then maintains the invariant that after scanning the first i transactions:

- $D_X[j] = 1$ iff an element of one of the scanned transactions' write-sets hashes to j
- $D_S[k] = 1$ iff an element of one of the scanned transactions' read-sets hashes to k

Therefore, if at any point the next transaction scanned (let's call it T_{next}) has the properties

- $D_X[\text{hash(key)}] = 0$ for all keys in T_{next}'s read-set
- $D_X[\text{hash(key)}] = 0$ for all keys in T_{next}'s write-set
- $D_S[\text{hash(key)}] = 0$ for all keys in T_{next}'s write-set

then T_{next} does not conflict with any of the prior scanned transactions and can safely be run[2].

In other words, SCA traverses the `TxnQueue` starting with the oldest transactions and looking for a transaction that is ready to run and does not conflict with any older transaction. Pseudocode for SCA is provided in Figure 3.

[2]Although there may be some false negatives (in which an "actually" runable transaction is still perceived as blocked) due to the need to hash the entire keyspace into a 100kB bitstring, this algorithm gives no false positives.

```
// SCA returns a transaction that can safely be run (or null
// if none exists). It is called only when TxnQueue is full.
function SCA()
    // Create our 100kB bit arrays.
    bit Dx[819200] = {0};
    bit Ds[819200] = {0};
    foreach Txn T in TxnQueue
        // Check whether the Blocked transaction
        // can safely be run
        if (T.state() == BLOCKED)
            bool success = true;
            // Check for conflicts in ReadSet.
            foreach key in T.ReadSet
                // Check if a write lock is held by any
                // earlier transaction.
                int j = hash(key);
                if (Dx[j] == 1)
                    success = false;
                Ds[j] = 1;
            // Check for conflicts in WriteSet.
            foreach key in T.WriteSet
                // Check if a read or write lock is held
                // by any earlier transaction.
                int j = hash(key);
                if (Dx[j] == 1 OR Ds[j] == 1)
                    success = false;
                Dx[j] = 1;
            if (success)
                return T;
        // If the transaction is free, just mark the bit-arrays.
        else
            foreach key in T.ReadSet
                int j = hash(key);
                Ds[j] = 1;
            foreach key in T.WriteSet
                int j = hash(key);
                Dx[j] = 1;
    return NULL;
```

Figure 3: SCA pseudocode.

SCA is actually "selective" in two different ways. First, it only gets activated when it is really needed (in contrast to traditional lock manager overhead which always pays the cost of tracking lock contention even when this information will not end up being used). Second, rather than doing an expensive all-to-all conflict analysis between active transactions (which is what traditional lock managers track at all times), SCA is able to limit its analysis to those transactions that are (a) most likely to be able to run immediately and (b) least expensive to check.

In order to improve the performance of our implementation of SCA, we include a minor optimization that reduces the CPU overhead of running SCA. Each key needs to be hashed into the 100kB bitstring, but hashing every key for each transaction as we iterate through the `TxnQueue` can be expensive. We therefore cache the results of the hash function the first time SCA encounters a transaction inside the transaction state. If that transaction is still in the `TxnQueue` the next time SCA iterates through the queue, the algorithm may then use the saved list of offsets that corresponds to the keys read and written by that transaction to set the appro-

priate bits in the SCA bitstring, rather having to re-hash each key.

3. EXPERIMENTAL EVALUATION

To evaluate VLL and SCA, we ran several experiments comparing VLL (with and without SCA) against alternative schemes in a number of contexts. We separate our experiments into two groups: single-machine experiments, and experiments in which data is partitioned across multiple machines in a shared-nothing cluster.

In our single-machine experiments, we ran VLL (exactly as described in Section 2.1) in a multi-threaded environment on a multi-processor machine. As a comparison point, we implemented a traditional two-phase locking (2PL) protocol inside the same main-memory database system prototype. This allows for an apples-to-apples comparison in which the only difference is the locking protocol.

Our second implementation of VLL is designed to run in a shared-nothing (multi-server) configuration. As described in Section 3.1, the cost of running any locking protocol is usually dwarfed by the contention cost of distributed commit protocols such as two-phase commit (2PC) that are used to guarantee ACID for multi-partition transactions. We therefore implemented VLL inside Calvin, a deterministic database system that does *not* require distributed agreement protocols to commit distributed transactions [22]. We compare our deterministic VLL implementation against (1) Calvin's default concurrency control scheme (which uses standard lock management data structures to track what data is locked), (2) a traditional distributed database implementation which uses 2PL (with a standard lock manager on each partition that tracks locks of data in that partition) and 2PC, and (3) an H-Store implementation[3] [19] that partitions data across threads (so that an 8-server cluster, where each server has 8 hardware threads, is partitioned into 64 partitions) and executes all transactions at each partition within a single thread, removing the need for locking or latching of shared data structures.

Our prototype is implemented in C++. All the experiments measuring throughput were conducted on a Linux cluster of a 2.6 GHz quad-core Intel Xeon X5550 machines with 8 CPU threads and 12G RAM, connected by a single gigabit Ethernet switch.

In order to minimize the effects of irrelevant components of the database system on our results, we devote 3 out of 8 cores on every machine to those components that are completely independent of the locking scheme (e.g. client load generation, performance monitoring instrumentation, intra-process communications, input logging, etc.), and devote the remaining 5 cores to worker threads and lock management threads. For all techniques that we compare in the experiments, we tuned the worker thread pool size by hand by increasing the number of worker threads until throughput decreased due to too much thread contention.

3.1 Multi-core, single-server experiments

This section compares the performance of VLL against two-phase locking. For VLL, we analyze performance with and without the SCA optimization. We also implement two versions of 2PL: a "traditional" implementation that detects deadlocks via timeouts and aborts deadlocked transactions, and a deadlock-free variant of 2PL in which a transaction places all of its lock requests in a single atomic step (where the data that must be locked is determined in an identical way as VLL, as described in Section). However, this modified version of 2PL still differs from VLL in that it uses a traditional lock management data structure.

Our first set of single-machine experiments uses the same microbenchmark as was used in [22]. Each microbenchmark transaction reads 10 records and updates a value at each record. Of the 10 records accessed, one is chosen from a small set of 'hot' records, and the rest are chosen from a larger set of 'cold' records. Contention levels between transactions can be finely tuned by varying the size of the set of hot records. In the subsequent discussion, we use the term *contention index* to refer to the probability of any two transactions conflicting with one another. Therefore, for this microbenchmark, if the number of the hot records is 1000, the contention index would be 0.001. If there is only one hot record (which would then be modified by every single transaction) the contention index would be 1. The set of cold records is always large enough such that transactions are extremely unlikely to conflict due to accessing the same cold record.

As a baseline for all four systems, we include a "no locking" scheme, which represents the performance of the same system with all locking completely removed (and any isolation guarantees completely forgone). This allows us to clearly see the overhead of acquiring and releasing the locks, maintaining lock management data structures for each scheme, and waiting for blocked transactions when there is contention.

Figure 4 shows the transactional throughput the system is able to achieve under the four alternative locking schemes.

When contention is low (below 0.02), VLL (with and without SCA) yields near-optimal throughput. As contention increases, however, the `TxnQueue` starts to fill up with blocked transactions, and the SCA optimization is able to improve performance relative to the basic VLL scheme by selectively unblocking transactions and "unclogging" the execution engine. In this experiment, SCA boosts VLL's performance by up to 76% under the highest contention levels.

At the very left-hand side of the figure, where contention is very low, transactions are always able to acquire all their locks and run immediately. Looking at the difference between "no locking" and 2PL at low contention, we can see that the locking overhead of 2PL is about 21%. This number is consistent with previous measurements of locking overhead in main memory databases [8, 14]. However, the overhead of VLL is only 2%, demonstrating VLL's lightweight nature. By co-locating lock information with data to increase cache locality, and by representing lock information in only two integers per record, VLL is able to lock data with extremely low overhead.

As contention increases, the throughput of both VLL and 2PL decrease (since the system is "clogged" by blocked transactions), and the two schemes approach one another in performance as the additional information that the 2PL scheme keeps in the lock manager becomes increasingly useful (as more transactions block). However, it is interesting to note that 2PL never overtakes the performance of VLL with SCA, since SCA can quickly construct the relevant part of trans-

[3]Although we refer to this system as "H-Store" in the discussion that follows, we actually implemented the H-Store protocol within the Calvin framework in order to provide as fair a comparison as possible.

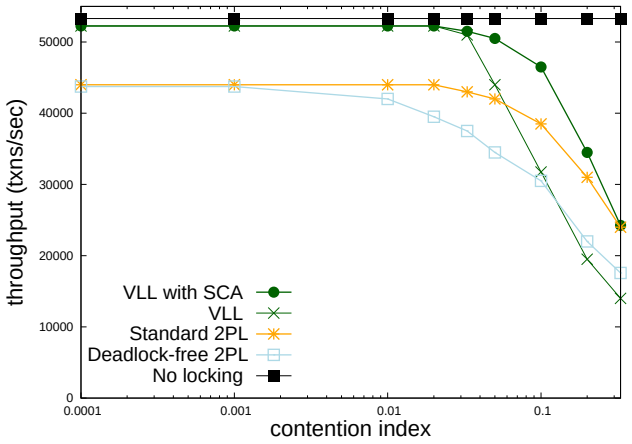

Figure 4: Transactional throughput vs. contention under a deadlock-free workload.

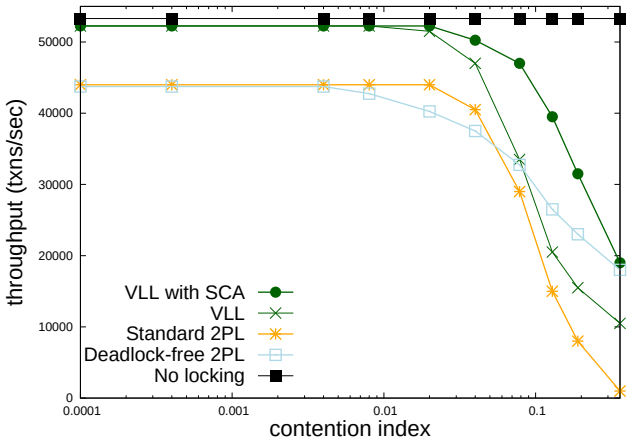

Figure 5: Transactional throughput vs. contention under a workload in which deadlocks are possible.

actional data dependencies on the fly. In fact, it appears that the overhead of repeatedly running SCA at very high contention is approximately equal to the overhead of maintaining a full lock manager. Therefore, VLL with the SCA optimization has the advantage that it eliminates the lock manager overhead when possible, while still reconstructing the information stored in standard lock tables when this is needed to make progress on transaction execution.

With increasing contention, the deadlock-free 2PL implementation saw a greater throughput reduction than the traditional 2PL implementation. This is because in traditional 2PL, transactions delay requesting locks until actually reading or writing the corresponding record, thereby holding locks for shorter durations than the deadlock-free version in which transactions request all locks up front.

Since this workload involved locking only one hot item per transaction, there is (approximately) no risk of transactions deadlocking[4]. This hides a major disadvantage of 2PL relative to VLL, since 2PL must detect and resolve deadlocks, while VLL does not, since VLL is always deadlock-free regardless of the transactional workload (due to the way it orders its lock acquisitions). In order to model other types of workloads where multiple records per transaction can be contested (which can lead to deadlock for the traditional lock schemes), we increase the number of hot records per transaction in our second experiment. Figure 5 shows the resulting throughput as we vary contention index. Frequent deadlocks causes throughput for traditional 2PL to drop dramatically, which results in VLL with SCA outperforming 2PL by at least 163% in most cases, and as much as 18X in the extreme high contention case (the right-most part of the figure).

Although traditional 2PL performance changes significantly between Figures 4 and 5, the deadlock-free 2PL variant is not affected (as expected). However, since it still uses a traditional lock manager data structure, it has higher overhead than VLL at low contention levels. Furthermore, the critical section in which all locks are acquired is more expensive than the corresponding critical section for VLL (since it includes the more expensive hash table and queue operations associated with standard lock table implementations instead of VLL's simple integer operations) and therefore continues to be outperformed by VLL even at higher contention levels.

[4]The exception is that deadlock is possible in this workload if transactions conflict on cold items. This was rare enough, however, that we observed no deadlocks in our experiments.

3.2 Distributed Database Experiments

In this section, we examine the performance of VLL in a distributed (shared-nothing) database architecture. We compare VLL against three other concurrency control designs for distributed main memory databases: traditional distributed 2PL, H-Store's lock-free protocol, and Calvin's deterministic locking protocol.

The first comparison point we use is traditional distributed 2PL, which uses a traditional lock manager (exactly as in the single-server experiments) on each partition, and uses the two-phase commit (2PC) protocol to commit distributed transactions.

The H-Store implementation [19] eliminates all lock manager overhead by removing locking entirely and executing transactions serially. In order to make use of multiple cores (and multiple machines), H-Store partitions data across cores and runs transactions serially in a single thread at each partition. If a transaction touches data across multiple partitions, each participating partition blocks until *all* participants reach and start to process that transaction. The main disadvantage of this approach is that, in general, partitions are not synchronized in their execution of transactions, so partitions participating in a distributed transaction may have to wait for other, slower participants to reach the same point in execution to complete the transaction (and since transactions are executed serially, the waiting participant cannot apply idle CPU resources to other useful tasks while waiting). Furthermore, even after each partition starts on the same transaction, there can be significant idle time while waiting for messages to be passed between partitions. Therefore, H-Store performs extremely well on "embarrassingly partitionable" workloads—where every transaction can be executed within a single partition—but its performance quickly degrades as distributed transactions enter the mix.

The third comparison point is Calvin, a deterministic database system prototype that employs a traditional hash-table-of-request-queues structure in its lock manager. Previous work has shown that Calvin performs similarly to traditional database systems when transactions are short and only access memory-resident data, worse when there are long-running transactions that access disk (due to the additional inflexibility associated with the determinism guarantee), and better when there are distributed transactions (since the determinism guarantee allows the elimination of two-phase commit) [21, 22]. For these experiments, all transactions are completely in-memory (so that Calvin is expected to perform approximately as well as possible as a comparison point), and we vary the percentage of transactions that span multiple partitions.

Since VLL acquires locks for each transaction all at once, it is possible for it to also be used in a deterministic system. We simply require that each transaction acquires its locks in the same order as the deterministic transaction sequence. This is more restrictive than the general VLL algorithm—which allows transactions to (atomically) request their locks in any order—but allows for a more direct comparison with Calvin, since by satisfying Calvin's determinism invariant, VLL too can use Calvin's determinism invariant to eschew two-phase commit. To further improve the quality of comparison, we allow H-Store to omit the two-phase commit protocol as well (even though the original H-Store papers implement two-phase commit).

For these experiments, we implement the single-threaded version of VLL as described in Section 2.2, since, as explained in that section, this allows for locks to be acquired outside of critical sections. The main disadvantage of single-threaded VLL is that in order to utilize all the CPU resources on a multi-core server, data must be partitioned across each core (with a single thread in charge of processing transactions for that core), which leads to the overhead of dealing with multi-partition transactions for transactions that touch data controlled by threads on different cores. However, since this "distributed" set of experiments has to deal with multi-partition transactions anyway (for transactions that span data stored on different physical machines), the infrastructure cost of this extra overhead has already been incurred. Therefore, we use the multi-threaded version of VLL for the single-server (multi-core) experiments, and the single-threaded version of VLL for the distributed database experiments.

Since we are running a distributed database, we dedicate one of the virtual cores on each database server node to handle communication with other nodes (in addition to the three other virtual cores previously reserved for database components described above), leaving 4 virtual cores dedicated to transaction processing. For H-Store and both VLL implementations, we leveraged these 4 virtual cores by creating 4 data partitions per machine, so every execution engine used one core to handle transactions associated with its partition, and for distributed 2PL, we left 4 virtual cores dedicated to worker threads[5]. Similarly to distributed 2PL, Calvin does not partition data within a node. We found

[5]For 2PL, we hand-tuned the number of worker threads for each experiment, since workloads with more multi-partition transactions require more worker threads to keep the CPU occupied (because many worker threads are sleeping, waiting for remote data). With enough worker threads, the extended contention footprint of 2PC eventually becomes the bottleneck that limits total throughput.

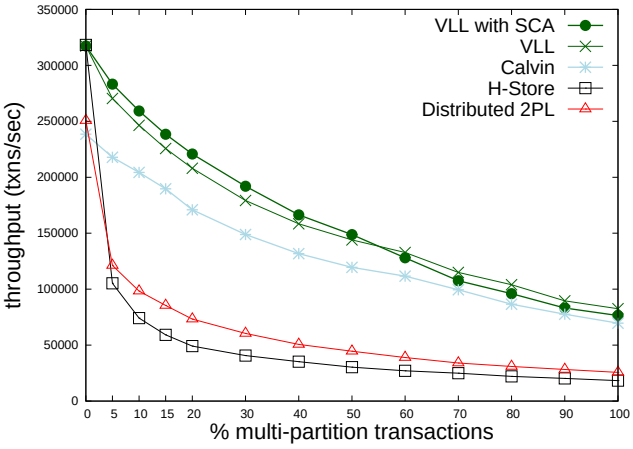

Figure 6: Microbenchmark throughput with low contention, varying how many transactions span multiple partitions.

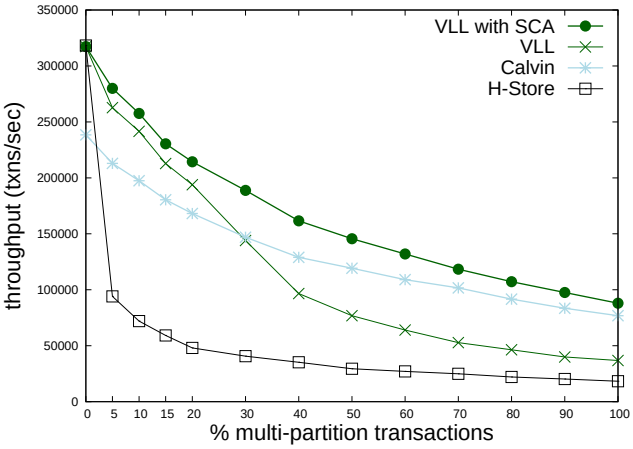

Figure 7: Microbenchmark throughput with high contention, varying how many transactions span multiple partitions (distributed 2PL results omitted due to large amounts of distributed deadlock).

that Calvin's scheduler (which includes one thread that performs all lock manager operations) fully utilized one of the four cores that were devoted to transaction processing.

3.2.1 *Microbenchmark experiments*

In our first set of distributed experiments, we used the same microbenchmark from Section 3.1. For these experiments, we vary both lock contention and the percentage of multi-partition transactions.

We ran this experiment on 8 machines, so H-Store and VLL split data across 32 partitions, while the Calvin and distributed 2PL deployments used a single partition per machine for a total of 8 partitions (each of which was 4 times the size of one H-Store/VLL partition). For the low contention test (Figure 6), each H-Store/VLL partition con-

tained 1,000,000 records of which 10,000 were hot, and each Calvin partition contained 4,000,000 records of which 40,000 were hot. Since VLL and H-Store had a smaller number of hot records *per partition*, they ended up having a slightly larger contention index than Calvin and distributed 2PL (0.0001 for VLL/H-Store vs. 0.000025 for Calvin/2PL). Calvin and distributed 2PL therefore had a slight advantage in the experiments relative to VLL and H-Store. Similarly, for the high contention tests (Figure 7), we used 100 hot records for each H-Store/VLL partition and 400 hot records for each Calvin partition, resulting in contention indexes of 0.01 for VLL/H-Store and 0.0025 for Calvin.

Before comparing VLL to the other schemes, we examine VLL with and without the SCA optimization. Under high contention, SCA is extremely important, and VLL's throughput is poor without it. Under low contention however, the SCA optimization actually hinders performance slightly when there are more than 60% multi-partition transactions. There are three reasons for this effect. First, under low contention, very few transactions are blocked waiting for locks, so SCA has a smaller number of potential transactions to unblock.

Second, since multi-partition transactions take longer than single-partition transactions, the `TxnQueue` typically contains many multi-partition transactions waiting for read results from other partitions. The higher the percentage of multi-partition transactions, the longer the `TxnQueue` tends to be (recall that the queue length is limited by the number of blocked transactions, not the total number of transactions). Since SCA iterates through the `TxnQueue` each time that it is called, the overhead of each SCA run therefore increases with multi-partition percentage.

Third, since low contention workloads typically have more multi-partition transactions per blocked transaction in the `TxnQueue`, each blocked transaction has a higher probability of being blocked behind a multi-partition transaction. This further reduces the effectiveness of SCA's ability to find transactions to unblock, since multi-partition transactions are slower to finish, and there is nothing that SCA can do to accelerate a multi-partition transaction.

Despite all these reasons for the reduced effectiveness of SCA for low contention workloads, SCA is only slower than regular VLL by a small amount. This is because SCA runs are only ever initiated when the CPU would otherwise be idle, so SCA only comes with a cost if the CPU would have left its idle state before SCA finishes its iteration through the `TxnQueue`.

VLL with SCA always outperforms Calvin in both experiments. With few multi-partition transactions, the lock manager overhead is clearly visible in these plots. Calvin keeps one of its four dedicated transaction processing cores (or 25% of its available CPU resources) saturated running the lock manager thread. VLL is therefore able to outperform Calvin by up to 33% by eliminating the lock manager thread and devoting that extra core to query execution instead. As the number of multi-partition transactions increases, the lock management overhead becomes less of a bottleneck as throughput becomes limited by communication delays necessary to process multi-partition transactions. Therefore, the performance of Calvin and VLL (with SCA) become more similar.

As in the single-server experiments, the full contention data tracked by the standard lock manager becomes useful at higher contention workloads, since information contained in the lock manager can be used to unblock transactions, so Calvin outperforms the default version of VLL under high contention (especially when there are more multi-partition transactions that fill the `TxnQueue` and prevent blocked transactions from reaching the head of the queue). However, the SCA optimization is able to reconstruct these transactional dependencies when needed, nullifying Calvin's advantage.

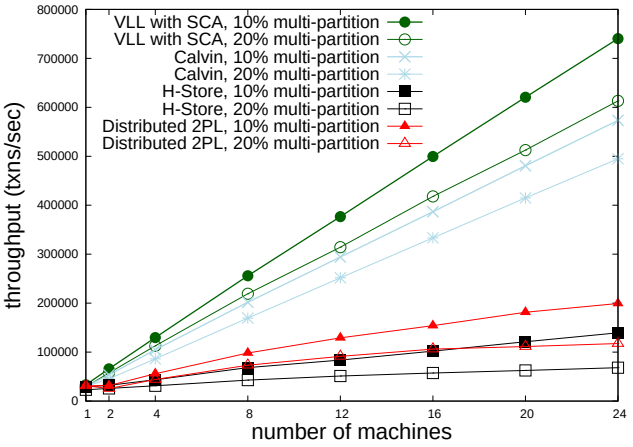

Figure 8: Scalability.

Both VLL and Calvin significantly outperform the H-Store (serial execution) scheme as more multi-partition transactions are added to the workload. This is because H-Store partitions have no mechanism for concurrent transaction execution, and so must sit idle any time it depends on an outstanding remote read result to complete a multi-partition transaction[6]. Since H-Store is designed for partitionable workloads (fewer than 10% multi-partition transactions), it is most interesting to compare H-Store and VLL at the left-hand side of the graph. Even at 0% multi-partition transactions, H-Store is unable to significantly outperform VLL, despite the fact that VLL acquires locks before executing a transaction, while H-Store does not. This further demonstrates the extremely low overhead of lock acquisition in VLL.

Both VLL and Calvin also significantly outperform the traditional distributed 2PL scheme (usually by about 3X). This is largely due to the fact that the distributed 2PL scheme pays the contention cost of running 2PC while holding locks (which Calvin and VLL do not). Furthermore, while Calvin, VLL, and H-Store all avoid deadlock, traditional distributed databases that use 2PL and 2PC must detect and resolve deadlocks. We found that under our high contention benchmark, distributed deadlock rates for the

[6]Subsequent versions of H-Store proposed to speculatively execute transactions during this idle time [10], but this can lead to cascading aborts and wasted work if there would have been a conflict. We do not implement speculative execution in our H-Store prototype since H-Store is designed for workloads with small percentages of multi-partition transactions (when speculative execution is not necessary), and our purpose for including it as a comparison point is only to analyze how it compares to VLL on these target single-partition workloads.

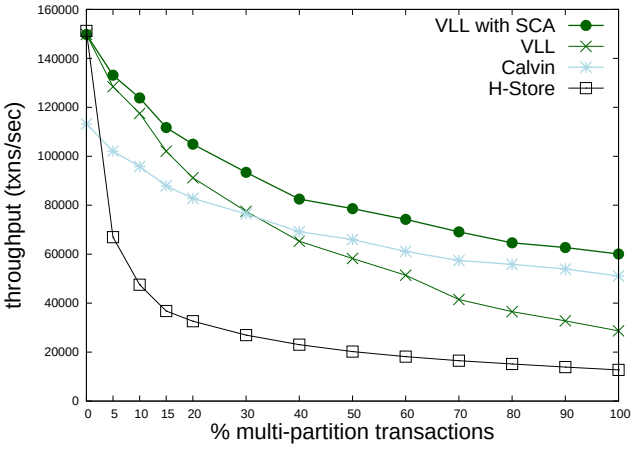

Figure 9: TPC-C throughput.

distributed 2PL scheme were so high that throughput approached 0. We therefore omitted the distributed 2PL line from Figure 7.

Figure 8 shows the results of an experiment in which we test the scalability of the different schemes at low contention when there are 10% and 20% multi-partition transactions. We scale from 1 to 24 machines in the cluster. This figure shows that VLL is able to achieve similar linear scalability as Calvin, and is therefore able to maintain (and extend) its performance advantage at scale. Meanwhile, traditional distributed 2PL does not scale as well as VLL and Calvin, again due to the contention cost of 2PC. We observed very similar behavior under the higher contention benchmark (plot omitted due to space constraints).

3.2.2 *TPC-C experiments*

To expand our benchmarking efforts beyond our simple microbenchmark, we present in this section benchmarks of VLL on TPC-C. The TPC-C benchmark models the OLTP workload of an eCommerce order processing system. TPC-C consists of a mix of five transactions with different properties. In order to vary the percentage of multi-partition transactions in TPC-C, we vary the percentage of New Order transactions that access a remote warehouse.

For this experiment, we divided 96 TPC-C warehouses across the same 8-machine cluster described in the previous section. As with the microbenchmark experiments, VLL and H-Store partitioned the TPC-C data across 32 three-warehouse partitions (four per machine), while Calvin partitioned it across 8 twelve-warehouse partitions (one per machine).

Figure 9 shows the throughput results for VLL, Calvin, and H-Store. (Again, we omit the distributed 2PL scheme because high amounts of distributed deadlocks caused near-zero throughput). Overall, the relative performance of the different locking schemes is very similar to the high-contention microbenchmark results. The high contention in TPC-C results in the SCA optimization improving performance relative to regular VLL by 35% to 109% when multi-partition transactions are numerous.

The shape of the Calvin and H-Store lines relative to the two versions of VLL is also similar to the high contention microbenchmark experiments. At low percentages of multi-partition transactions, the lock manager overhead of Calvin reduces its performance, but at higher percentages of multi-partition transactions, it is able to use the information in the lock manager to unblock stuck transactions and is able to outperform VLL unless the SCA optimization is used.

3.3 CPU costs

The above experiments show the performance of the different locking schemes by measuring total throughput under different conditions. These results reflect both (a) the CPU overhead of each scheme, and (b) the amount of concurrency achieved by the system.

In order to tease apart these two components of performance, we measured the CPU cost per transaction of requesting and releasing locks for each locking scheme. These experiments were not run in the context of a fully loaded system—rather, we measured the CPU cost of each mechanism in complete isolation.

locking mechanism	per-transaction CPU cost (μs)
Traditional 2PL	20.13
Deterministic Calvin	20.16
Multi-threaded VLL	1.8
Single-threaded VLL	0.71

Figure 10: Locking overhead per transaction.

Figure 10 shows the results of our isolated CPU cost evaluation, which demonstrates the reduced CPU costs that VLL achieves by eliminating the lock manager. We find that the Calvin and 2PL schemes (which both use traditional lock managers) have an order of magnitude higher CPU cost than the VLL schemes. The CPU cost of multi-threaded VLL is a little larger than the cost of single-threaded VLL, since multi-threaded VLL has the additional cost of acquiring and releasing a latch around the acquisition of locks.

4. RELATED WORK

The System R lock manager described in [6] is the lock manager design that has been adopted by most database systems. In order to reduce the cost of locking, there have been several published methods for reducing the number of lock calls [11, 15, 17]. While these schemes may partially mitigate the cost of locking in main-memory systems, but they don't address the root cause of high lock manager overhead—the size and complexity of the data structure used to store lock requests.

[12] presented a Lightweight Intent Lock (LIL), which maintains a set of lightweight counters in a global lock table. However, this proposal doesn't co-locate the counters with the raw data (to improve cache locality), and if the transaction doesn't acquire all of its locks immediately, the thread blocks, waiting to receive a message from another released transaction thread. VLL differs from this approach in using the global transaction order to figure out which transaction should be unlocked and allowed to run as a consequence of the most recent lock release.

The idea of co-locating a record's lock state with the record itself in main memory databases was proposed almost two decades ago [5, 16]. However, this proposal involved maintaining a linked list of "Lock Request Blocks" (LCBs) for each record, significantly complicating the underlying data structures used to store records, whereas VLL

aims to *simplify* lock tracking structures by compressing all per-record lock state into a simple pair of integers.

Given the high overhead of the lock manager when a database is entirely in main memory [8, 9, 14], some researchers observe that executing transactions serially that without concurrency control can buy significant throughput improvement in main memory database systems [10, 19, 23]. Such an approach works well only when the workload can be partitioned across cores, with very few multi-partition transactions. VLL enjoys some of the advantages of reduced locking overhead, while still performing well for a much larger variety of workloads.

Other attempts to avoid locks in main memory database systems include optimistic concurrency control schemes and multi-version concurrency control schemes [1, 4, 13, 14]. While these schemes eliminate locking overhead, they introduce other sources of overhead. In particular, optimistic schemes can cause overhead due to aborted transactions when the optimistic assumption fails (in addition to data access tracking overhead), and multi-version schemes use additional (expensive) memory resources to store multiple copies of data.

5. CONCLUSION AND FUTURE WORK

We have presented *very lightweight locking* (VLL), a protocol for main memory database systems that avoids the costs of maintaining the data structures kept by traditional lock managers, and therefore yields higher transactional throughput than traditional implementations. VLL co-locates lock information (two simple semaphores) with the raw data, and forces all locks to be acquired by a transaction at once. Although the reduced lock information can complicate answering the question of when to allow blocked transactions to acquire locks and proceed, *selective contention analysis* (SCA), allows transactional dependency information to be created as needed (and only as much information as is needed to unblock transactions). This optimization allows our VLL protocol to achieve high concurrency in transaction execution, even under high contention.

We showed how VLL can be implemented in traditional singer-server multi-core database systems and also in deterministic multi-server database systems. The experiments we presented demonstrate that VLL can outperform standard two-phase locking, deterministic locking, and H-Store style serial execution schemes significantly—without inhibiting scalability, or interfering with other components of the database system. Furthermore, VLL is highly compatible with both standard (non-deterministic) approaches to transaction processing and deterministic database systems like Calvin.

Our focus in this paper was on database systems that update data in place. In future work, we intend to investigate multi-versioned variants of the VLL protocol, and integrate hierarchical locking approaches into VLL.

6. ACKNOWLEDGMENTS

This work was sponsored by the NSF under grants IIS-0845643 and IIS-1249722, and by a Sloan Research Fellowship. Kun Ren is also supported by National Natural Science Foundation of China under Grant 61033007 and National 973 project under Grant 2012CB316203.

7. REFERENCES

[1] D. Agrawal and S. Sengupta. Modular synchronization in distributed, multiversion databases: Version control and concurrency control. *IEEE TKDE*, 5, 1993.

[2] R. Agrawal and M. J. C. M.Livny. Concurrency control performance modeling: Alternatives and implications. volume 12, pages 609–654, 1987.

[3] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] V. Gottemukkala and T. Lehman. Locking and latching in a memory-resident database system. *VLDB*, 1992.

[6] J. Gray. *Notes on database operating systems*. Operating System, An Advanced Course. Springer-Verlag,Berlin, 1979.

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

[8] S. Harizopoulos, D. J. Abadi, S. R. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[9] R. Johnson, I. Pandis, and A. Ailamaki. Improving oltp scalability using speculative lock inheritance. *PVLDB*, 2(1):479–489, 2009.

[10] E. P. C. Jones, D. J. Abadi, and S. R. Madden. Concurrency control for partitioned databases. In *SIGMOD*, 2010.

[11] A. Joshi. Adaptive locking strategies in a multi-node data sharing environment. *VLDB*, 1991.

[12] H. Kimura, G. Graefe, and H. Kuno. Efficient locking techniques for databases on modern hardware. In *Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2012.

[13] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *TODS*, 6(2), 1981.

[14] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory database. *PVLDB*, 5(4):298–309, 2011.

[15] T. Lehman. *Design and Performance Evaluation of a Main Memory Relational Database System*. PhD thesis, University of Wisconsin-Madison, 1986.

[16] T. J. Lehman and V. Gottemukkala. *The Design and Performance Evaluation of a Lock Manager for a Memory-Resident Database System*. Performance of Concurrency Control Mechanisms in Centralised Database System, Addison-Wesley, 1996.

[17] C. Mohan and I. Narang. Recovery and coherency-control protocols for fast inter-system page transfer and fine-granularity locking in shared disks transaction environment. *VLDB*, 1991.

[18] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[19] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, Vienna, Austria, 2007.

[20] A. Thomasian. Two-phase locking performance and its thrashing behavior. *TODS*, 18(4):579–625, 1993.

[21] A. Thomson and D. J. Abadi. The case for determinism in database systems. *VLDB*, 2010.

[22] A. Thomson, T. Diamond, P. Shao, K. Ren, S.-C. Weng, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[23] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In *HPTS*, 1997.