The VLDB Journal (2015) 24:681–705
DOI 10.1007/s00778-014-0377-7

SPECIAL ISSUE PAPER

VLL: a lock manager redesign for main memory database systems

Kun Ren · Alexander Thomson · Daniel J. Abadi

Received: 14 February 2014 / Revised: 9 October 2014 / Accepted: 18 December 2014 / Published online: 4 January 2015

Abstract Lock managers are increasingly becoming a bottleneck in database systems that use pessimistic concurrency control. In this paper, we introduce *very lightweight locking* (VLL), an alternative approach to pessimistic concurrency control for main memory database systems, which 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 are partitioned across many commodity machines in a shared-nothing cluster. Furthermore, we show how VLL and SCA can be extended to enable range locking. Our experiments show that VLL dramatically reduces locking overhead and thereby increases transactional throughput in both settings.

Keywords Lightweight locking · Main memory · Lock manager · Deterministic · Contention · Scalability

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 [12]. 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 [14,22,29]. 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 [2,4,5,11,34]. 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

K. Ren (✉) · D. J. Abadi
Yale University, New Haven, CT, USA
e-mail: kun@cs.yale.edu

D. J. Abadi
e-mail: dna@cs.yale.edu

A. Thomson
Google, New York, NY, USA
e-mail: agt@google.com

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 [8]). 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 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 [36,37] has 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.

We also propose an extension of VLL (called VLLR) that locks ranges of rows rather than individual rows. Experiments show that VLLR outperforms two traditional range locking mechanisms.

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 are partitioned across *processors* (possibly spanning multiple independent servers). At each partition, a single thread executes transactions serially.
- Data are 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 trade-offs of running VLL in the other two architectures.

2.1 The VLL algorithm

The biggest difference between VLL 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 represents 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 Sect. 2.4.

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`

[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.

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 Sect. 2.4). 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 Sect. 2.6.

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 Sect. 2.6). 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 (for simplicity, this pseudocode only considers local transactions). Each worker thread in the system executes the `VLLMainLoop` function. Figure 2 depicts an example execution trace for a sequence of transactions.

```
// 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);
```

Fig. 1 Pseudocode for the VLL algorithm

2.2 Arrayed VLL

As discussed above, VLL usually prefers to colocate the C_X and C_S values for each record with the record itself. Unlike traditional lock management data structures, which are traditionally implemented as linked lists, C_X and C_S are both simple integers and are therefore much easier to integrate into the data records themselves. This leads to improved cache/memory bandwidth utilization, as a single request from memory brings both the record and the lock information about the record into cache.

On the other hand, one disadvantage of this approach is that it spreads out lock information across the entire dataset. If there is code that only accesses lock information (without accessing the data itself), it is better to have all the lock information stored together, rather than interspersed with the data. One example of this is the deterministic lock manager of Calvin. In order to implement the deterministic locking algorithm described in previous work [35], Calvin sometimes puts its lock manager in a separate thread that is dedicated to just lock management. Since this thread never touches the raw data, it would be preferable to keep all lock management data together in order to ensure that the cache local to the core that this lock manager thread is running is filled with lock data and not polluted with record data as well.

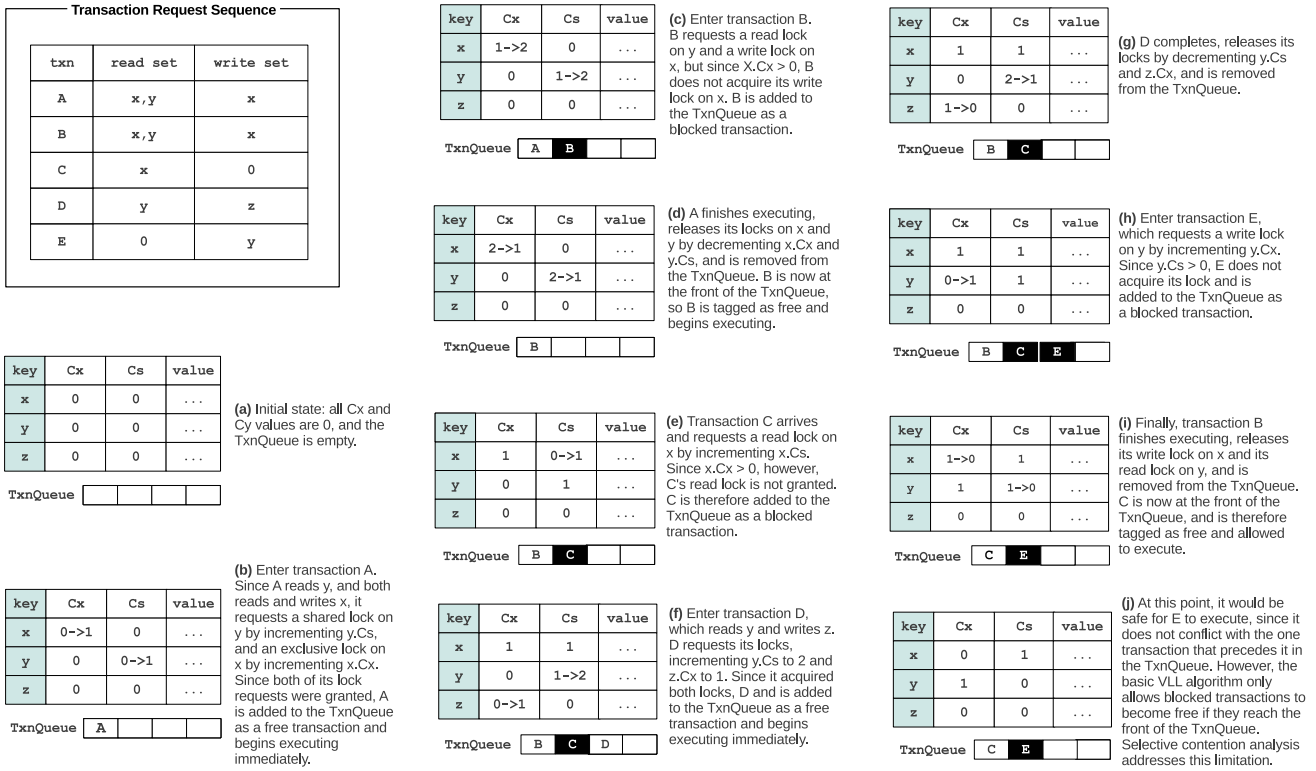


Fig. 2 Example execution of a sequence of transactions $\{A, B, C, D, E\}$ using VLL. Each transaction's read and write set is shown in the *top left box*. Free transactions are shown with *white* backgrounds in the `TxnQueue`, and blocked transactions are shown as *black*. Transaction logic and record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

We therefore designed another implementation of VLL, which we call "arrayed VLL", which uses a vector or array data structure to store integer lock information consecutively. For example in C++, we store all C_X information in one vector and all C_s information in a second vector to represent read–write locks and read locks, respectively. The ith element of each vector corresponds to the C_X and C_s values for the ith record.

The overhead of arrayed VLL is a little larger than the colocated version of VLL, since data and lock information are accessed in two separate requests; however, when the lock manager is running in a separate thread, this implementation is preferable, since the memory request for the lock is often in the cache of the core running the lock manager (especially when the number of records is small, or when the records which are accessed are skewed).

We will further compare the performance differences between arrayed VLL and colocated VLL in Sect. 3.

2.3 Single-threaded VLL

Thus far, we have 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 [33], where data are 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 was implemented simply by running only one thread (on each partition) according to the previous specification, the result would be a serial execution of transactions (within each partition). This results in wasted resources, since when the thread needs to sleep, waiting for a message from a distributed node, and no other progress can be made within that partition.

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.

In other words, while there is still only one thread per partition, this thread now has the capability to work on more than one transaction at once—switching over to a different transaction (instead of sleeping) when waiting for messages from distributed nodes. Since there is still only one thread per partition, this implementation shares the H-Store advantage of not requiring latches/critical sections around lock acquisition. Figure 3 shows the pseudocode for the distributed single-threaded VLL algorithm (note the lack of critical sections relative to Fig. 1).

2.4 Impediments to acquiring all locks at once

As discussed in Sect. 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 sets 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 Figs. 1 and 3. After performing these exploratory reads, it enters the critical section and requests those locks that it discovered it would likely need. Once the

```
// Requests exclusive locks on all records in T's WriteSet
// and shared locks for T's ReadSet, and sets T's status.
function BeginTransaction(Txn T)
    T.Type = Free;
    foreach key in T.ReadSet // Request T's read locks
        if key is in local storage
            data[key].Cs++;
            if (data[key].Cx > 0)
                T.Type = Blocked;
    foreach key in T.WriteSet // Request T's write locks
        if key is in local storage
            data[key].Cx++;
            if (data[key].Cx > 1 OR data[key].Cs > 0)
                T.Type = Blocked;
    if T is a distributed transaction And T.Type == Free
        T.Type = Waiting

// Releases T's locks and removes T from TxnQueue.
function FinishTransaction(Txn T)
    foreach key in T.ReadSet
        if key is in local storage
            data[key].Cs−−;
    foreach key in T.WriteSet
        if key is in local storage
            data[key].Cx−−;

// Transaction execution thread main loop.
function VLLMainLoop()
    while (true)
        // Select a transaction to run next...
        // First choice: Resume a "waiting" transaction
        if (Received remote message for transaction T)
            HandleReadResult(T, message);
            if ReadyToExecute(T);
                Execute(T);
                FinishTransaction(T);
                TxnQueue.Remove(T);
        // Second choice: a previously-blocked txn
        // that now does not conflict with older txns.
        else if (TxnQueue.front().Type == Blocked)
            Txn T = TxnQueue.front();
            if T is a distributed transaction
                T.Type = Waiting;
                Send local reads to other participating nodes
            else
                T.Type = Free;
                Execute(T);
                FinishTransaction(T);
                TxnQueue.Remove(T);
        // Third 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);
            else if (T.Type == Waiting)
                Send local reads to other participating nodes
                TxnQueue.Enqueue(T);
            else if (T.Type == Blocked)
                TxnQueue.Enqueue(T);
```

Fig. 3 Pseudocode for the single-threaded VLL algorithm

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.

As will be discussed further in Sect. 3, we tried both approaches and found that for high-contention workloads, the first solution is problematic, since the overhead of handling and detecting distributed deadlock completely negates the VLL advantage of reducing the overhead of lock management. Meanwhile, the second approach, while adding nontrivial coordination overhead, is still able yield improved performance. For low-contention workloads, both approaches are possibilities.

However, recent work on deterministic database systems [35–37] shows that the coordination overhead of the second approach can be reduced by performing it 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 [35], the integration of VLL and deterministic database systems seems to be a particularly good match. We therefore used the second approach with coordination happening before transaction execution for our implementation in most experiments of this paper.

2.5 Trade-offs of VLL

VLL's primary strength lies in its extremely low overhead in comparison with 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	Write set
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			Standard	
Key	Cx	Cs	Key	Request queue
x	2	0	x	A, C
y	1	0	y	B
z	2	0	z	C, D
TxnQueue				
A, B, C, D				

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

VLL			Standard	
Key	Cx	Cs	Key	Request queue
x	1	0	x	C
y	1	0	y	B
z	2	0	z	C, D
TxnQueue				
B, 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.

2.6 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 Sect. 2.5 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 *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 ith transaction in the `TxnQueue` can only conflict now with up to $(i - 1)$ prior transactions, whereas it previously had to contend with (up to) the number of `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 100 kB (so that both will easily fit inside an L2 cache of size 256 kB) 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[hash(key)] = 0 for all keys in T_{next}'s read set
- D_X[hash(key)] = 0 for all keys in T_{next}'s write set
- D_S[hash(key)] = 0 for all keys in T_{next}'s write set

```
// 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;
```

Fig. 4 SCA pseudocode

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 Fig. 4.

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

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

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 100 kB bitstring, but hashing every key for each transaction as we iterate through the `TxnQueue` can be expensive. We therefore cache (inside the transaction state) the results of the hash function the first time SCA encounters a transaction. 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 appropriate bits in the SCA bitstring, rather having to re-hash each key.

2.7 Very lightweight locking of ranges (VLLR)

For workloads that frequently read, write, or delete many consecutive rows within the same transaction, it can be useful to lock ranges of rows rather than many individual "point" rows. This technique is particularly useful in avoiding phantoms [26] and in the common use case of deleting an entity whose data rows all share a common prefix of their primary key. Some database systems, such as Spanner, use range locks exclusively in place of hash table-based "point" locks [7]. This section describes an extension of VLL (called VLLR for brevity) that locks ranges of rows rather than individual rows.

VLLR works by locking bitstring prefixes. When locking a range of primary keys, the range is represented first as a range R of (lexicographically sorted) bitstrings, and then a set P of bitstrings is generated such that for each bitstring K of the range, there exists an element $p \in P$ that is a prefix of K. One simple way to generate P is to choose the set consisting of the longest common prefix of the minimum and maximum bitstrings in R.[3]

Given a set P of prefixes to lock, VLLR proceeds in a manner similar to hierarchical locking [10]. In traditional hierarchical locking, *intention locks* are acquired from course to fine granularity before the actual lock is acquired on the target key—for example, intention locks may be acquired on database, then table, then page before an actual lock for a row is requested. VLLR's approach is analogous in that it requests intention locks on all nonempty prefixes of each element in P.

	C_x	C_s	I_x	I_s
C_x	X	X	X	X
C_s	X		X	
I_x	X	X		
I_s	X			

Fig. 5 Lock mode conflicts for VLLR (Xs indicate conflict)

To manage lock states, VLLR uses four counters per key: C_x, C_s, I_x, and I_s. When requesting a lock on a prefix $p \in P$, $C_x[p]$ or $C_s[p]$ is incremented (depending whether the request is for an exclusive or shared lock) and for each nonempty strict prefix p_j of p, $I_x[p_j]$ or $I_s[p_j]$ is incremented, respectively. For each incremented counter, all conflicting counters are checked (see Fig. 5 for a table of which pairs of counters conflict) to see whether the lock is acquired.

Note that the lock management overhead of VLLR is bounded to one increment/decrement for each bit in the combined elements of the union of P sets for each transaction. Overlapping lock ranges incur no extra lock management work as they do in traditional range locking mechanisms where ranges may have to be split.

The primary difference between VLLR and VLL is the maintenance of two extra counters, corresponding to I_x and I_s. Therefore, it is straightforward to apply SCA to VLLR in an analogous way to how it is applied for VLL—SCA simply has to add two extra bitmaps corresponding to I_x and I_s (note, however, that SCA for VLLR will have to update more bits within the bitstrings per transaction to account for the additional prefixes that must be locked). Figure 6 shows the pseudocode for VLLR. Figure 7 depicts an example execution trace for a sequence of transactions.

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 describe our experimental setup in Sect. 3.1, and the remainder of this section presents our experimental results.

3.1 Experimental setup

In this section, we describe our experimental setup. In order to fully evaluate the VLL protocol, we implement three different versions of VLL (one single-machine version and two distributed versions) and compare each version to systems of corresponding designs using traditional locking methods. In

[3] This can sometimes result in locking a much larger part of the key space than needed. For example, in an 8-bit key space, locking the range R = `[00111100,01000010]` with this method results in locking all keys with prefix `0xxxxxxx`—half the key space rather than the necessary 7/256 of the key space. In this particular case, P could instead consist of the prefixes `001111xx`, `0100000x`, and `01000010`, thus locking exactly the rows in R. Or P could consist of `001111xx` and `010000xx`, locking the rows in R, plus one additional row (`010000011`). Under some workloads, locking larger ranges than necessary may incur expensive and unnecessary contention costs, justifying the costs of finer-grained $R \rightarrow P$ transformations. Under other workloads, coarser-grained prefix locking may be acceptable and users may profit from the reduced locking overhead.

```
function bool RequestLock(Range range, uint64 txn,
bool exclusive)
    TxnQueue.Enqueue(txn);
    vector<Key> prefixes = range.ToPrefixes();
    bool success = true;
    foreach key in prefixes
        success &= RequestPrefixLock(key, exclusive);
    return success;

function void ReleaseLock(Range range, uint64 txn,
bool exclusive)
    vector<Key> prefixes = range.ToPrefixes();
    foreach key in prefixes)
        ReleasePrefixLock(key, exclusive);
    TxnQueue.Remove(txn);

function bool RequestPrefixLock(Key key,
bool exclusive)
    // Check if this or descendant node is X- or S-locked.
    uint64 k = key.hash();
    bool success = true;
    if (exclusive)
        success &= (Cs[k] == 0);
    success &= (Cx[k] == 0);
    if (exclusive)
        success &= (Is[k] == 0);
    success &= (Ix[k] == 0);
    // Check if any ancestor node is X- or S-locked.
    for (int i = 1; i < key.size(); i++)
        uint64 h = key.prefix(i).hash();
        success &= (Cx[h] == 0);
        if (exclusive)
            success &= (Cs[h] == 0);
    // Increment S/X counter for node.
    if (exclusive)
        Cx[k]++;
    else
        Cs[k]++;
    // Increment descendant sums for all ancestor nodes.
    for (int i = 1; i < key.size(); i++)
        uint64 h = key.prefix(i).hash();
        if (exclusive)
            Ix[h]++;
        else
            Is[h]++;
    return success;

function void ReleasePrefixLock(Key key,
bool exclusive)
    uint64 k = key.hash();
    if (exclusive)
        Cx[k]–;
    else
        Cs[k]–;
    for (int i = 1; i < key.size(); i++)
        uint64 h = key.prefix(i).hash();
        if (exclusive)
            Ix[h]–;
        else
            Is[h]–;
```

Fig. 6 Pseudocode for the VLLR algorithm

total, we implemented and evaluated nine systems. Section 3.1.1 describes each of these nine systems. After describing these systems, we describe the benchmarks used to evaluate these systems in Sect. 3.1.2.

3.1.1 Compared systems

We separate our experiments into two groups: single-machine experiments and experiments in which data are partitioned across multiple machines in a shared-nothing cluster.

In our single-machine experiments, we ran VLL (exactly as described in Sect. 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.

For our distributed, shared-nothing cluster experiments, we considered two different partitioning implementations: one where each multicore *machine* in the cluster contains a partition, and the multi-threaded version of VLL runs on each machine (as described in Sect. 2.4); and one "H-Store-style" implementation where each *core* on each machine has its own separate partition, and each partition runs all transactions in a single worker thread (as described in Sect. 2.3).

As discussed earlier, distributed versions of VLL are susceptible to entering distributed deadlock states. We presented two potential solutions to this problem: (a) detecting deadlocks by analyzing transaction dependencies and aborting transactions to break cycles and (b) avoiding deadlocks via global coordination of lock acquisition order. We built and experimented with both options.

We implement the first solution in the context of a traditional System-R* design [27], using two-phase commit (2PC) for distributed transactions, but with the lock manager on each machine being replaced by our VLL implementation from Sect. 2.1. As a comparison point, we compare against the same exact traditional System-R* design, with the same mechanisms for detecting and resolving distributed deadlock (discussed later in this section), but using a traditional hash table-based lock manager implementing the two-phase locking protocol instead of VLL.

The second option can leverage the Calvin design (which, as described in Sect. 2.4, also avoids distributed deadlock by creating a global order of all transactions). Since globally ordering transactions also has the nice property of enabling deterministic transaction execution and 2PC is not necessary in deterministic systems [37], we allow our implementation of the second option also to take advantage of this property and eschew 2PC. Therefore, we compare this implemen-

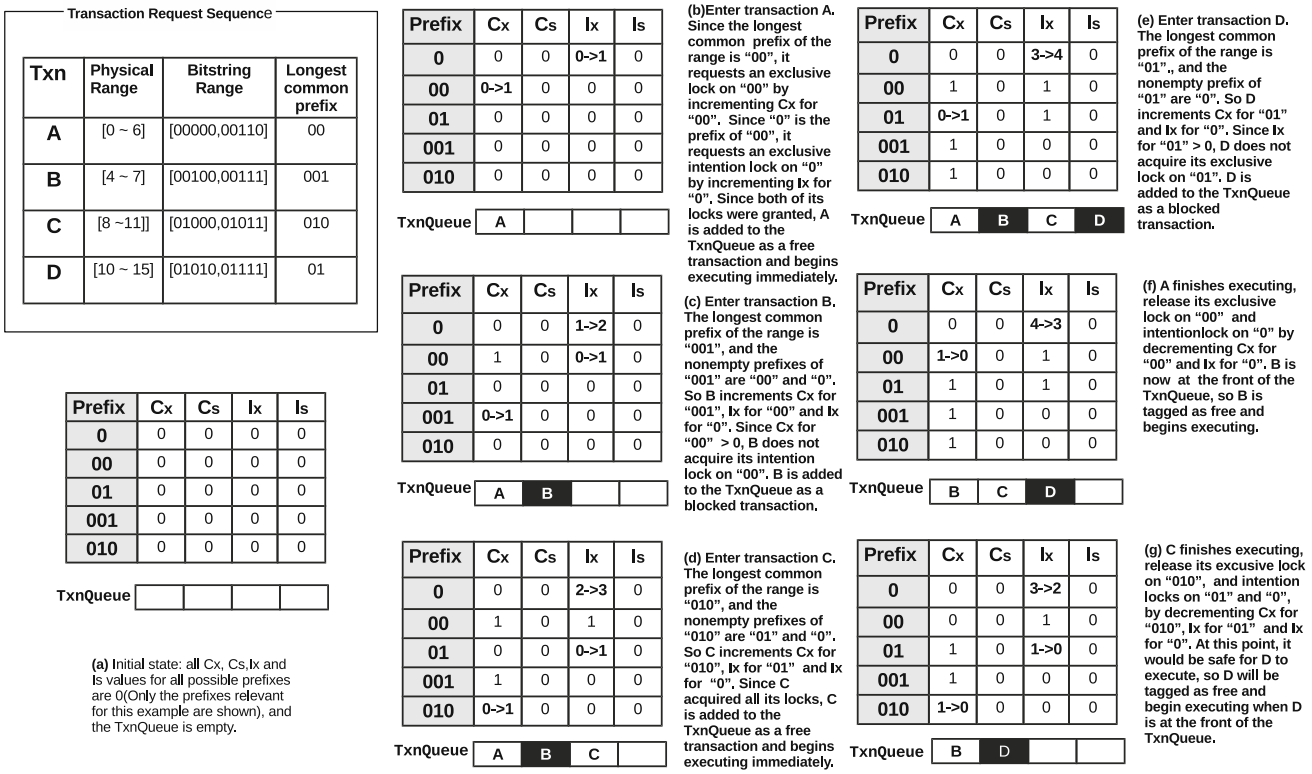


Fig. 7 Example execution of a sequence of transactions $\{A, B, C, D\}$ using the VLLR algorithm. Each transaction requests an exclusive lock on a range. Currently executing transactions are shown with *white* backgrounds in the `TxnQueue`, and blocked transactions are shown as *black*

tation with the original version of Calvin. The two implementations use the exact same code for everything but lock management—including the code for choosing the global order of transactions, the code for deterministic transactional execution without two-phase commit, and code for scheduling transactions to run in worker threads once all locks have been successfully acquired. The only difference is that we replaced Calvin's hash table-based lock manager with our VLL implementation. Since Calvin runs all lock management operations in a separate thread from the worker threads that execute transaction logic, we use the array-based implementation of VLL described in Sect. 2.2.

In summary, we compare four different distributed systems that contain multiple threads per partition: a traditional R* design (1) with and (2) without VLL, and a Calvin design (3) with and (4) without VLL. The traditional R* design uses 2PC and implements deadlock detection and resolution (using a waits-for graph—see below), whereas the Calvin design avoid deadlocks and processes distributed transactions without 2PC.

For the distributed "H-Store"-style design where each core on each machine has its own separate partition and each partition contains just one thread, we implement and compare three different implementations. First, we implement VLL (exactly as described in Sect. 2.3) directly into Calvin codebase. Each partition, running on each core, runs the single-threaded version of VLL whose pseudocode was given in Fig. 3.

We compare our single-threaded VLL implementation against two similar alternatives. First, we compare against an H-Store implementation[4] [33] that also partitions data across cores (so that an 8-server cluster, where each server has five cores devoted to processing transactions, is partitioned into 40 partitions) and executes all transactions at each partition serially within a single thread, removing the need for locking or latching of shared data structures. H-Store therefore has the advantage of not requiring any overhead for lock management, but the disadvantage of not allowing any concurrency within a partition (and for distributed transactions must sit and wait for remote messages instead of working on other transactions in the interim).

Second, we compare against a less extreme version of H-Store that uses a traditional lock manager inside each partition to enable concurrent transaction execution within a partition, so that the system does not need to wait and do nothing while waiting for remote messages as part of dis-

[4] 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.

tributed transactions. This implementation should therefore see improved performance for workloads that contain many distributed transactions, at the cost of increased overhead for lock management.

Since we built each of these three "H-Store-style" implementations inside the Calvin code base, they are all deterministic, and all can avoid 2PC for distributed transactions (just like the latter two implementations for the multi-threaded experiments). The only difference is lock management. Regular H-Store uses no locks and serializes transactions within each partition, lock manager-based H-Store uses a traditional lock manager in each partition, and VLL-based H-Store uses the single-threaded VLL implementation from Sect. 2.3.

We spent a lot of time investigating multiple different deadlock detection and resolution protocols. Although we used a timeouts-based protocol in an earlier version of this paper [30], we found that the waits-for graphs implementation from Gray [9] and Stonebraker [32] performs better in practice. We therefore carefully implemented the waits-for graphs protocol for deadlock detection and also optimized the "blocked transactions threshold" in our system in order to reduce deadlock by limiting the number of blocked transactions allowed in the system. We therefore managed to substantially improve the performance of our prototype for those implementations in which deadlock is possible relative to our previous work [30].

Our prototype is implemented in C++. All the experiments measuring throughput were conducted on Amazon EC2 using Double Extra Large instances (m3.2xlarge), which promise 30 GB of memory and 26 EC2 Computer Units—eight virtual cores with 3.25 EC2 Compute Units each (where each EC2 Compute Unit provides the roughly the CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor. Experiments were run on a shared-nothing cluster of eight of these Double Extra Large EC2 instances, unless stated otherwise.

In order to minimize the effects of irrelevant components of the database system on our results, we devote three out of eight 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 five 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.

For single-threaded systems implementations, we leveraged these five virtual cores by creating five data partitions per machine, so every execution engine used one core to handle transactions associated with its partition.

For configurations where data were not partition within a machine, we devoted all five virtual cores to worker threads when running the distributed 2PL protocol with deadlock detection. In the Calvin-based deadlock-free mechanisms, dedicated one core entirely to the lock manager thread, leaving four cores for worker threads.

3.1.2 Benchmarks

The first set of experiments we present in this paper use the same microbenchmark experimented with in several recently published papers [30,37]. 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 1,000, 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.

The transactions in this microbenchmark are short and simple—reading and updating 10 records can be performed very quickly. A relatively high percentage of transaction time is spent acquiring locks, since 10 separate data items must be locked, and once the lock is acquired, very little time is spent performing the required actions on the locked item (a simple read and update). We therefore refer to this transactional workload as our "short" microbenchmark.

In order to experiment with workloads where a smaller percentage of transaction time is spent acquiring locks, we introduce an altered version of the microbenchmark where 10 ms of CPU computation must be performed on each data item (in addition to the read and update). We call this version of the microbenchmark the "long" microbenchmark, and each transaction in the benchmark takes approximately 150 ms in total (which is similar to the New Order transaction in the TPC-C benchmark). Transactions in the "long" microbenchmark take approximately three times longer than "short" microbenchmark transactions (150 vs. 50 ms).

To expand our experiments beyond our simple microbenchmarks, we also implement the full TPC-C benchmark. The TPC-C benchmark models the OLTP workload of an eCommerce order processing system. TPC-C consists of a mix of five transactions (New Order 45 %, Payment 43 %, Order Status 4 %, Stock Level 4 %, and Delivery 4 %) with different properties, and it contains both read–write heavy transactions and read-only transactions. Since the transaction logic

of TPC-C is complex and the contention index is high, our TPC-C experiments are most similar to our experiments on the "long" microbenchmark, under high contention.

3.2 Multi-core, single-server experiments

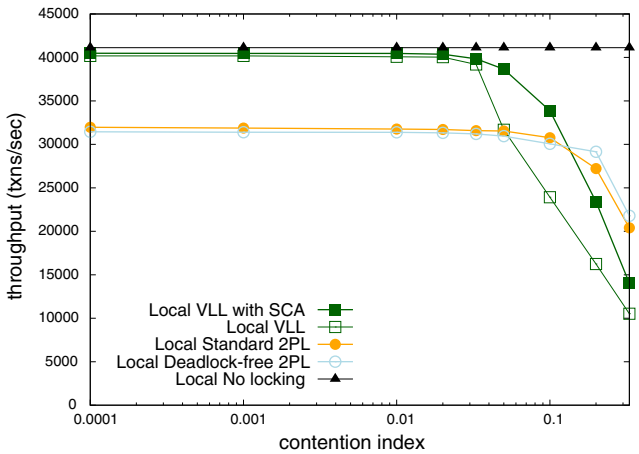
This section compares the performance of VLL against two-phase locking. For VLL, we analyze performance with and without the SCA optimization. We also implement two versions of 2PL: a "traditional" implementation that detects deadlocks and aborts deadlocked transactions, and a deadlock-free variant of 2PL in which a transaction places all of its lock requests in a single atomic step (where the data that must be locked are determined in an identical way as VLL, as described in Sect. 2.4). However, this modified version of 2PL still differs from VLL in that it uses a traditional lock management data structure.

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.

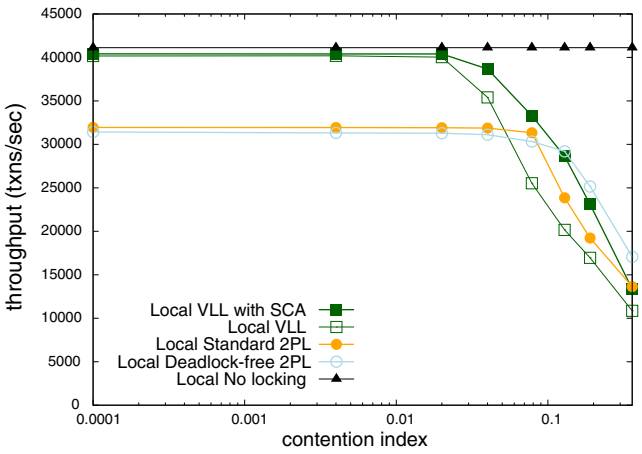
We benchmark our system using both the "long" microbenchmark transactions and "short" microbenchmark transactions described in Sect. 3.1.2.

Figure 8a shows the transactional throughput the system is able to achieve under the four alternative locking schemes for "long" microbenchmark transactions, and Fig. 8c shows the throughput for "short" microbenchmark transactions.

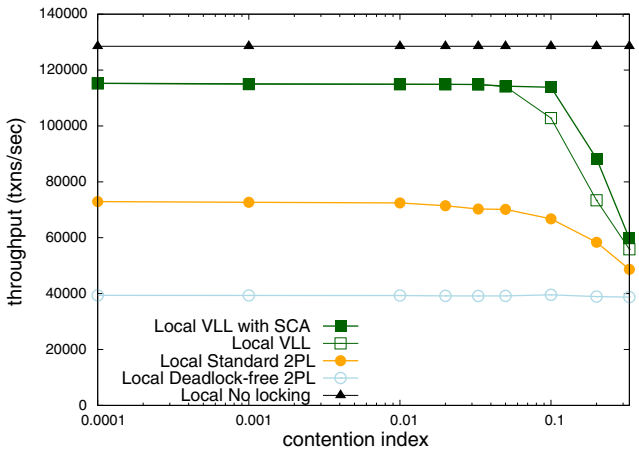
When contention is low (below 0.02), VLL (with and without SCA) yields near-optimal throughput. As contention



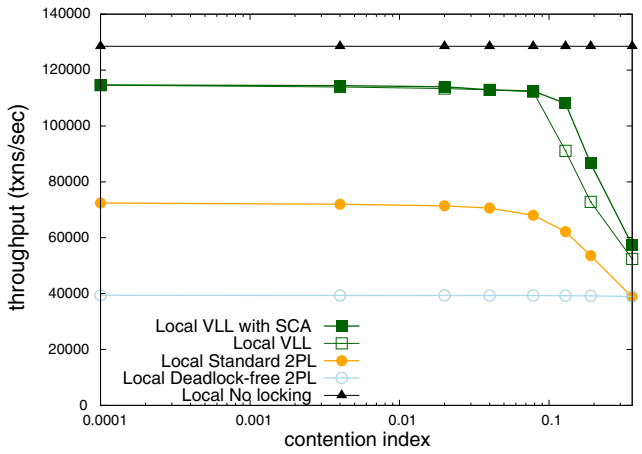
(a) "Long" transactions under a deadlock-free workload.



(b) "Long" transactions under a workload in which deadlocks are possible.



(c) "Short" transactions under a deadlock-free workload.



(d) "Short" transactions under a workload in which deadlocks are possible.

Fig. 8 Transactional throughput versus contention under various microbenchmark workloads

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. This effect is much stronger for the "long" transaction microbenchmark (where SCA boosts VLL's performance by up to 41 %) than the "short" transaction microbenchmark, where the benefit of SCA is more muted. This is explained by the fact that transactions in the "short" microbenchmark are being executed so fast that SCA's ability to remove transactions from the TxnQueue slightly earlier than they would otherwise be removed when reaching the head of the queue yields a very small improvement.

Interestingly, although SCA is able to improve throughput relative to basic VLL at high contention levels, at extremely high contention levels SCA once again approaches basic VLL in performance (this effect is seen for both "long" transactions and "short" transactions). The is because at extremely high contention, almost every transactions conflicts with every other transaction, and SCA becomes unable to find transactions that can be removed early from the TxnQueue. Therefore, the shape of the SCA graph relative to the basic VLL graph is a bubble, with close to identical performance at low and extremely high contentions, and improvement only observable at medium-to-high 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. Therefore, the difference between the "no locking" throughput and the throughput of each of the other implementations can be completely explained by the overhead of acquiring locks in each implementation. For the standard 2PL implementation, for "long" microbenchmark transactions, we see that the locking overhead of 2PL is about 22 %. This number is consistent with previous measurements of locking overhead in main memory databases [12,22]. However, the overhead reaches 43 % for the 2PL implementation for "short" microbenchmark transactions. This is because, as described in Sect. 3.1.2, for the "short" microbenchmark, a greater percentage of transaction time is spent acquiring locks. (The "long" transaction benchmark is more similar to real-world workloads with which the above-cited studies experimented.)

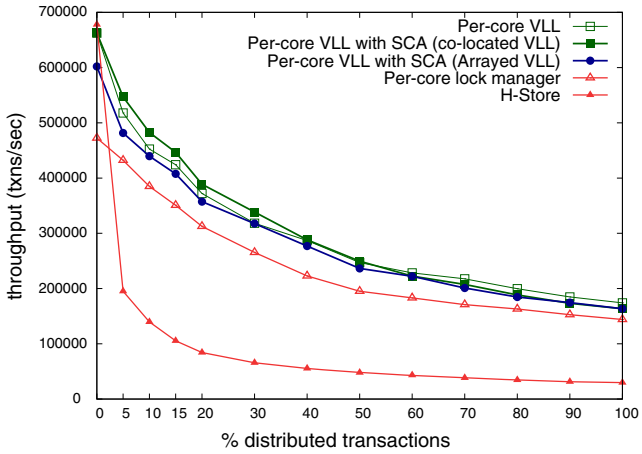
Meanwhile, the overhead of VLL is only 1.5 % for the "long" microbenchmark transactions and 10.2 % for the "short" microbenchmark transactions, providing evidence of VLL's lighter-weight lock manager implementation relative to 2PL. By colocating lock information with data to increase cache locality and by representing lock information in only two integers per record, VLL is able to lock data with extremely low overhead. However, the multi-threaded version of VLL still must acquire and release locks in a critical section, and this caused a bottleneck which become visible for the "short" microbenchmark transactions.

The deadlock-free 2PL implementation also acquires and releases locks in a critical section. However, the critical section of the deadlock-free 2PL implementation is more expensive than the corresponding critical section of VLL since the process of acquiring or releasing a lock in the deadlock-free 2PL implementation uses the much heavier-weight traditional hash-based lock manager data structure. For the "short" transaction microbenchmark, this critical section becomes such a bottleneck, that this implementation consistently yielded extremely poor performance, and was unaffected by the contention index.

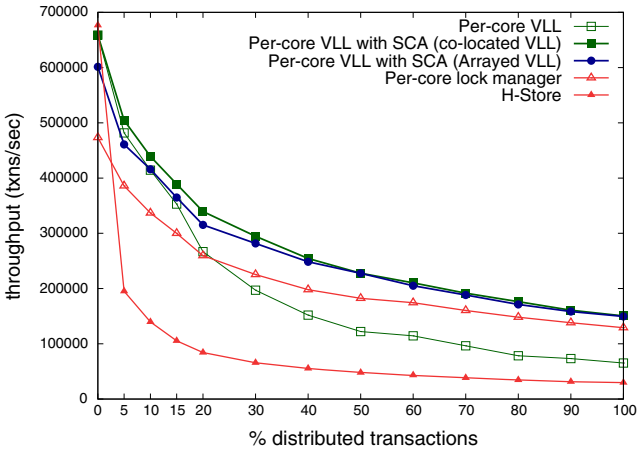
As contention increases, the throughput of both VLL and 2PL decreases (since the system is "clogged" by blocked transactions), and the two schemes approach one another in performance as the additional information that the 2PL scheme keeps in the lock manager becomes increasingly useful (as more transactions block). However, even at high contention, the performance of VLL with SCA is similar to 2PL, since SCA can quickly construct the relevant part of transactional data dependencies on the fly. 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.

Since this workload involved locking only one hot item per transaction, there is (approximately) no risk of transactions deadlocking.[5] 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 and present the results in Fig. 8b, d. Although the presence of deadlocks reduces the performance of 2PL-based locking implementations at high contention indexes (when deadlock becomes likely), our careful optimizations of the deadlock detection and resolution protocols in our prototype kept the effect of deadlock relatively small compared to the previous experiments. Therefore, Fig. 8b is similar to Fig. 8a, and Fig. 8d is similar to Fig. 8c. However, it is clear that the modified workload that allows deadlocks does reduce the performance of 2PL, but VLL is unaffected by the change in workload.

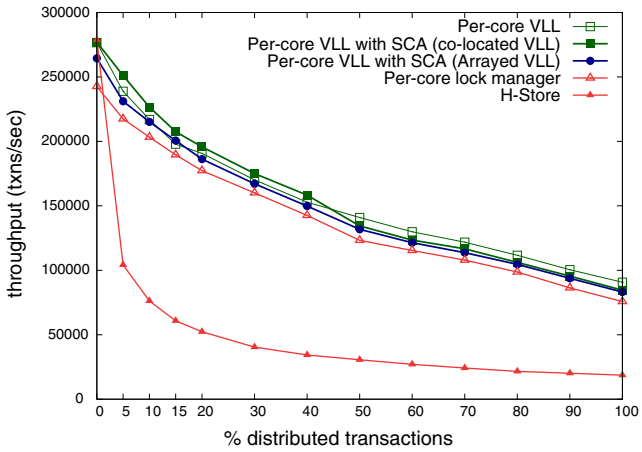
[5] The exception is that deadlock is possible in this workload if transactions conflict on cold items. This was rare enough, however, that we observed no deadlocks in our experiments.



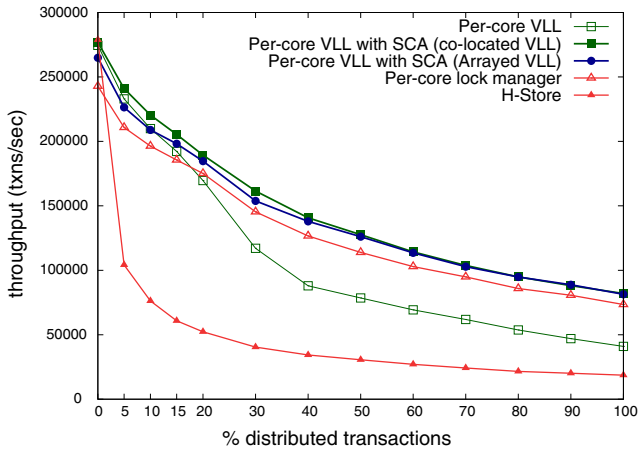
(a) "Short" transactions under low contention.



(b) "Short" transactions under high contention.



(c) "Long" transactions under low contention.



(d) "Long" transactions under high contention.

Fig. 9 Microbenchmark throughput of partition-per-core systems, varying how many transactions span multiple partitions

3.3 Distributed database experiments

In this section, we examine the performance of VLL in a distributed (shared-nothing) database architecture. We start with some experiments on the microbenchmark and then analyze performance on TPC-C.

As described in Sect. 3.1.1, we implemented VLL both for systems with one partition per machine and for systems with one partition per core (in the style of H-Store). Section 3.3.1 presents our experiments with the partition-per-core systems, and Sect. 3.3.2 presents our experiments with the partition-per-machine systems. Section 3.3.3 then presents our experiments on TPC-C.

3.3.1 Partition-per-core experiments

In our first set of distributed experiments, we used the microbenchmark (both "long" transactions and "short" transactions) and varied both lock contention and the percentage of multi-partition transactions.

These experiments ran on eight machines, each of which had five cores devoted to transaction processing, so the partition-per-core systems (H-Store, per-core VLL, per-core lock manager) split data across 40 partitions, while the partition-per-machine systems (Calvin, 2PL + 2PC, Nondeterministic VLL + 2PC, Deterministic VLL) split data across eight partitions. For the low-contention test, each partition contained 1,000,000 records of which 10,000 were hot, yielding a contention index of 0.0001. For the high-contention tests, we reduced the number of hot records to 100 per partition, resulting in a contention index of 0.01.

Figure 9 presents the results of these experiments. When comparing the performance of VLL with and without the SCA optimization, it is clear that for both "short" transactions and "long" transactions, SCA is extremely important under high contention, and VLL's throughput is poor without it. SCA optimization outperforms basic VLL by up to about 100 % with "long" transactions and about 130 % with "short" transactions. The benefit of SCA is thus much larger for these distributed system experiments than the single-machine

experiments presented in Sect. 3.2. This is because for the single-machine experiments, the head of the `TxnQueue` can always run (the only reason why a transaction will enter the `TxnQueue` is if it blocks waiting for locks and once a transaction reaches the head of the queue, it is guaranteed to be able to acquire all of its locks), so progress can always be made by running transactions from the head of the queue. For the distributed database experiments, the head of the queue can be stalled, waiting for a remote message. Without SCA, the entire `TxnQueue` must wait, waiting for this remote message. However, SCA is able to find other transactions in the `TxnQueue` to run, while the first transaction is waiting for remote data.

Under low contention, however, the SCA optimization actually hinders performance slightly when there are more than 60 % multi-partition transactions. We observe 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 very 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`. Overall, VLL with the SCA optimization performs similarly to regular VLL under low contention and up to 130 % faster under high contention.

When comparing colocated (where the lock information is stored inside the records themselves) and arrayed VLL (where the lock information is stored in separate vectors as described in Sect. 2.2), we see that with fewer multi-partition transactions, the colocated version VLL outperforms arrayed VLL since arrayed VLL's extra memory accesses represent approximately 10 % of the cost of each "short" microbenchmark transaction (and approximately 3–4 % overhead for each "long" microbenchmark transaction) when the transaction executes entirely locally. However, as distributed transactions are added into the mix, the overhead of cross-partition communication and coordination begins to dominate execution costs, and the benefit of colocating lock counters with data becomes much less visible.

Although the difference between arrayed VLL and colocated VLL is small, the difference between either VLL scheme and the non-VLL schemes is much larger. When comparing to the per-core lock manager scheme, the VLL scheme can improve performance by 10–30 % when there are few distributed transactions, but the difference becomes smaller as more distributed transactions are added to the workload. Both the per-core lock manager implementation and VLL implementation use locking to enable a transaction processing thread to work on other transactions while waiting for remote messages during a distributed transaction. Therefore, the only difference between these schemes is the locking overhead. This locking overhead difference is a greater percentage of total transaction execution time for local transactions and "short" transactions, but becomes less noticeable as transactions become longer or distributed.

The H-Store (serial execution) implementation performs extremely poorly as more multi-partition transactions are added to the workload. This is because H-Store partitions have no mechanism for concurrent transaction execution within a partition, 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 observation further highlights the extremely low overhead of lock management with VLL.

Since Fig. 9 measured throughput at only two different contention levels, Fig. 10 presents the throughput of these systems when contention is varied more directly. We fix the percentage of multi-partition transactions to be 20 % for this experiment.

When comparing VLL with and without the SCA optimization, it is clear that SCA becomes increasingly impor-

[6] Subsequent versions of H-Store proposed to speculatively execute transactions during this idle time [16], 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.

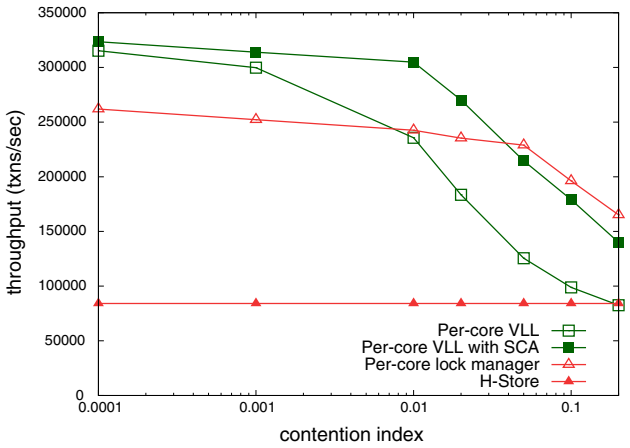


Fig. 10 Throughput of partition-per-core systems while varying contention

tant as the contention increases. However, SCA always outperforms basic VLL, even at low contention. Interestingly, we do not see the decrease in effectiveness of the SCA optimization at extremely high contention levels that we saw for the single-machine experiments in Fig. 8. This is because, as explained above, in the presence of distributed transactions, the head of the `TxnQueue` may stall waiting for remote messages, and SCA's ability to find other transactions to work on while the head of the queue is stalled is extremely important.

As contention increases, the throughput difference between VLL/SCA and the per-core lock manager becomes smaller, and under very high contention, the per-core lock manager eventually slightly outperforms VLL/SCA. This inflection point exemplifies the conditions under which the benefits of fully tracking contention data at all times outweighs the costs of maintaining standard lock queues. To the right of this point, information contained in the lock manager can be used to unblock transactions much more quickly than VLL—and this advantage outweighs the additional locking costs.

H-Store is unaffected by contention, since it processes transactions serially.

3.3.2 Partition-per-machine experiments

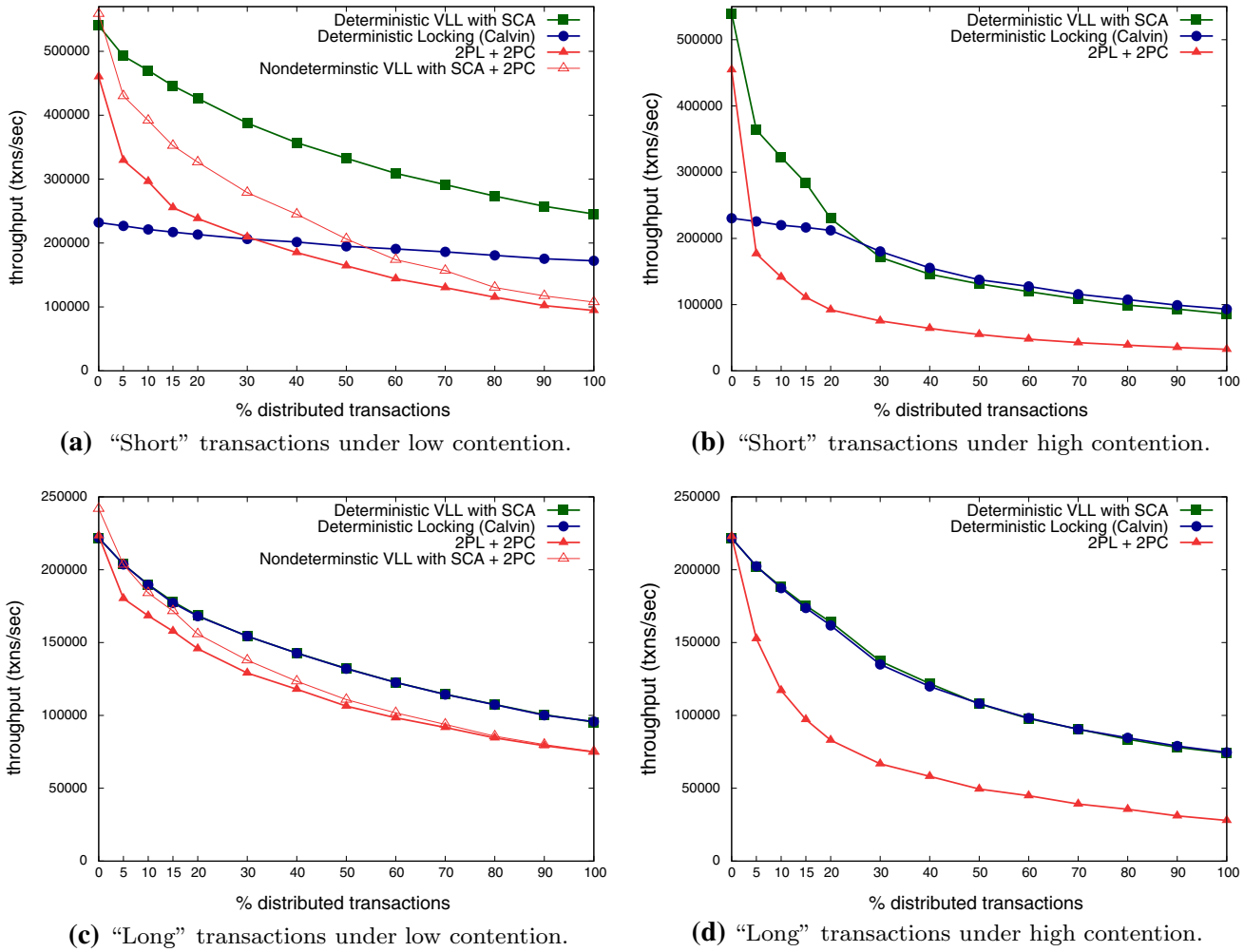
Figure 11 shows the results of our partition-per-machine experiments. As described in Sect. 3.1.1, we experiment with four systems: (1) a traditional System-R* design for a distributed database system that uses two-phase locking (with distributed deadlock detection and resolution) for concurrency control, and two-phase commit for committing distributed transactions—we call this system "2PL + 2PC"; (2) an identical version of the System-R* implementation with the traditional hash-based lock manager replaced with VLL (and the SCA optimization)—we call this "Nondeterministic VLL with SCA + 2PC"; (3) an implementation of Calvin that uses determinism to avoid deadlock and 2PC for distributed transactions—we call this "Deterministic Locking (Calvin)"; and (4) an identical version of Calvin with its traditional hash-based lock manager replaced with VLL (with the SCA optimization)—which we call "Deterministic VLL with SCA".

When comparing the nondeterministic System-R* design systems ("2PL + 2PC" vs. "Nondeterministic VLL with SCA + 2PC"), it is clear that VLL provides a greater benefit for "short" transactions than "long" transactions. This is because, as explained above, "short" transactions spend a greater percentage of transaction time acquiring and releasing locks, so reducing the overhead of the lock manager yields greater benefits for "short" transactions. This difference is only visible at low contention and low percentage of distributed transactions. At high contention, both nondeterministic systems have problems with distributed deadlock and perform similarly poorly. At high percentage of distributed transactions, both systems are limited by 2PC and costs of generating and processing network messages, and again the locking overhead is less of a bottleneck.

When comparing the deterministic systems to each other ("Deterministic Locking (Calvin)" vs. "Deterministic VLL"), the benefit of VLL is much greater for "short" transactions. This is because Calvin devotes an entire core to lock management and serializes lock acquisitions within one thread running on this core (to ensure the deterministic guarantees). For short transactions, the worker threads running on the other cores overwhelm the lock manager thread with lock requests, and this thread is unable to keep up with demand, becoming a massive bottleneck. Therefore, Calvin is almost totally unaffected by the percentage of distributed transactions in Fig. 11a—the lock manager is the bottleneck. By eliminating this bottleneck, VLL results in up to a factor of 2.3 performance improvement. However, for long transactions, the lock manager thread in Calvin is never a bottleneck. Therefore, the performance of Calvin and VLL is identical for "long" transactions (although the core running the lock manager thread is significantly under-utilized for the VLL implementation).

At high-contention and high-percentage of multi-partition transactions, the performance of "Deterministic Locking (Calvin)" and "Deterministic VLL" become more similar, since throughput is limited by contention on data instead of lock manager overhead.

Although this paper does not focus on comparing deterministic and nondeterministic database systems, it is nonetheless interesting to note that the deterministic systems generally perform better than the nondeterministic systems—especially when there are large numbers of distributed transactions, since the deterministic systems avoid two-phase commit and distributed deadlock. One exception to this rule is for short transactions at low contention, where Calvin is



(a) "Short" transactions under low contention.

(b) "Short" transactions under high contention.

(c) "Long" transactions under low contention.

(d) "Long" transactions under high contention.

Fig. 11 Microbenchmark throughput of partition-per-machine systems, varying how many transactions span multiple partitions

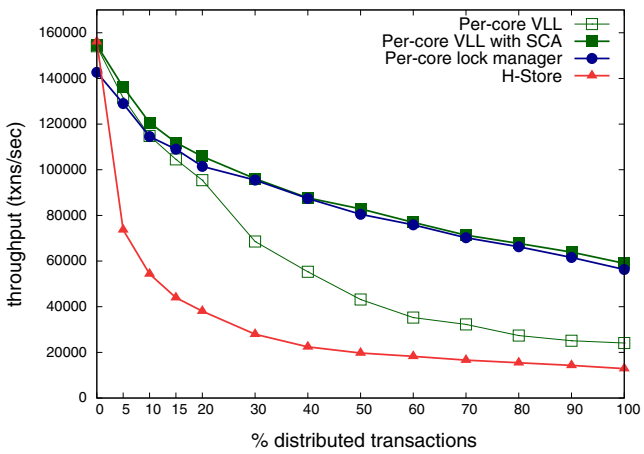
significantly outperformed by traditional 2PL + 2PC because of its bottleneck in the lock manager. However, VLL completely removes this bottleneck and enables Calvin to outperform the nondeterministic system at almost every data point. Although VLL is a very good fit for Calvin's deterministic locking scheme, it clearly improves performance of nondeterministic designs as well. Even at high contention levels, VLL is not outperformed by the traditional hash-based lock management schemes due to the SCA optimization.

3.3.3 TPC-C experiments

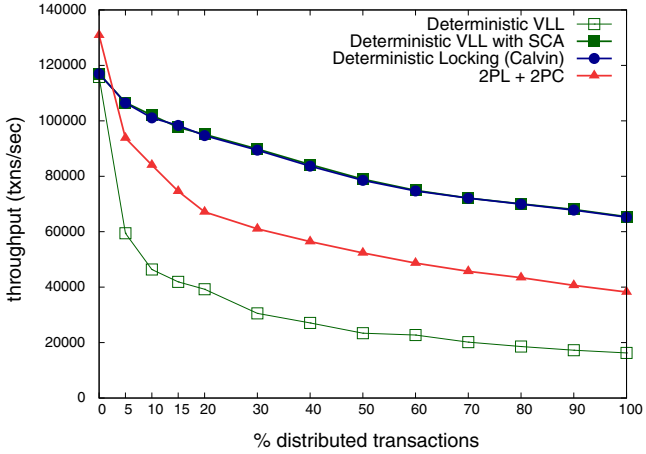
Although our microbenchmark allowed us to carefully vary the length, complexity, contention, and amount of multi-partition transactions in a transaction processing workload, we now present results on the better-known TPC-C benchmark. We keep the length, complexity, and contention levels identical to the TPC-C benchmark specifications, but vary the percentage of multi-partition transactions from 0 to 100 % (the TPC-C specification is for 10 % of transactions to access remote warehouses, and not every remote warehouse is in a different partition, so in practice, the actual percentage of distributed transactions in TPC-C is less than 10 %).

Just like the microbenchmark experiments, we experiment with partition-per-core and partition-per-machine systems. The partition-per-core systems partition the TPC-C data across 40 10-warehouse partitions (each machine contains five partitions, and each partition has 10 warehouses), while partition-per-machine systems partitioned the data across eight 20-warehouse partitions (20 warehouses per machine). The average contention index for the partition-per-core setup is approximately 0.02, and for the partition-per-machine setup is approximately 0.01.

Figure 12a shows the throughput results for the partition-per-core systems. At first glance, the shape and relative performance of each implementation in this figure is very similar to the same implementations in Fig. 9d. This is because TPC-C transactions are similar in terms of length to the "long" transactions of our microbenchmark (TPC-C New Order transactions are slightly longer), and similar in terms



(a) Throughput of "partition-per-core" systems



(b) Throughput of "partition-per-machine" systems

Fig. 12 TPC-C experiments

of contention index to the .01 contention index used in Fig. 9d. Therefore, the results (and analysis) are similar to "long", high-contention microbenchmark.

There are two important results to reiterate. First, the SCA optimization improves performance relative to regular VLL by 40–145 % when there are many multi-partition transactions. This is because, as described in Sect. 3.3.1, SCA finds other transactions to unblock when the head of the `TxnQueue` is stalled, waiting for a remote message, and this function is critical when there are many multi-partition transactions in the workload.

Second, since TPC-C transactions are slightly longer than "long" microbenchmark transactions, the performance of VLL (with SCA) and the per-core lock manager implementation (which uses a traditional hash table lock manager) are closer to each other in Fig. 12a than in Fig. 9d. In general, when contention is high and there are many distributed transactions, the lock manager is not a bottleneck (for the reasons discussed in Sect. 3.3.1). Therefore, switching from a hash-based lock manager to VLL does not lead to large improvement in throughput (and without the SCA optimization, leads to a large decrease in throughput).

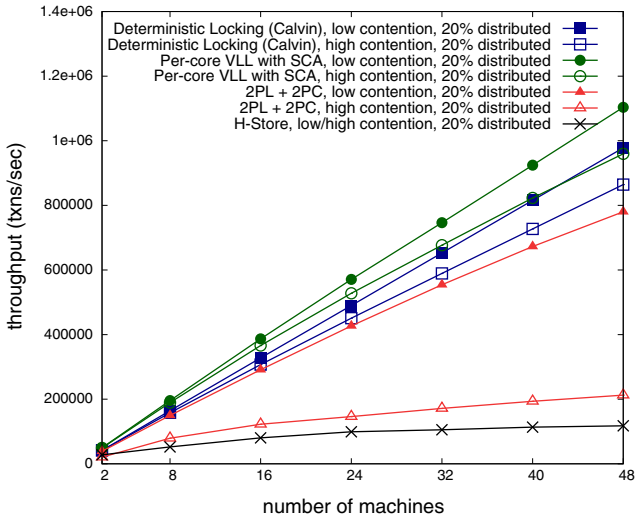
However, one should not conclude from these results that VLL is not beneficial for TPC-C. As mentioned above, the actual TPC-C specification for percentage of distributed transactions is less than 10 %. At that point in Fig. 12a, VLL (with SCA) outperforms the hash-based locking scheme by 8 %.

Figure 12b shows the throughput results for the partition-per-machine systems. Once again, the shape and relative performance of the implementations in this figure is similar to the "long", high-contention microbenchmark (in this case, Fig. 11d for the partition-per-machine microbenchmark). One difference between TPC-C and the microbenchmark experiment is that the throughput drops less significantly as the percentage of distributed transactions increases. This is because in our microbenchmarks, each distributed transaction touches one "hot" key per partition, whereas in TPC-C, each distributed transaction in TPC-C touches one "hot" key total (not per partition). Therefore, with more distributed transactions, the contention index actually decreases.

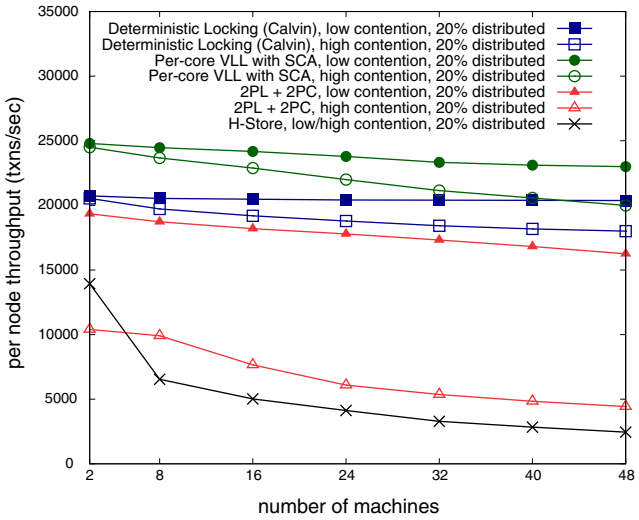
When there are no distributed transactions, the 2PL + 2PC implementation (the System-R*-design) outperforms the deterministic systems. This is because, as mentioned above, TPC-C transactions are comparable to the "long" transactions of our microbenchmark, which, as described in Sect. 3.3.2, leads to a relatively small number of lock requests per second, and thus reduced work for the lock acquisition thread. Therefore, the deterministic design of devoting an entire core to lock acquisition (see Sect. 3.3.2) is costly—this core is significantly under-utilized, and potential CPU resources are therefore wasted. In contrast, the 2PL + 2PC implementation is able to fully utilize all CPU resources. However, as there are more distributed transactions, the disadvantages of the System-R* design around 2PC and distributed deadlock present themselves, and the deterministic systems begin to outperform the 2PL + 2PC system.

3.3.4 Scalability experiments

Figure 13 shows the results of an experiment in which we test the scalability of the different schemes when there are 20 % multi-partition transactions. We scale from two to 48 machines in the cluster. We used the microbenchmark with "long" transactions under both low contention and high contention. We experimented with both partition-



(a) Scalability total throughput.



(b) Scalability per-node throughput.

Fig. 13 Scalability experiments (both partition-per-machine and partition-per-core systems are included)

per-core and partition-per-machine systems; in particular, we experimented with the H-Store and per-core VLL (with SCA) schemes from the partition-per-core systems, and with the 2PL + 2PC and Calvin schemes from the partition-per-machine systems.

Figure 13a shows total throughput measurements, and Fig. 13b shows per-node throughput measurements. These figures show that VLL is able to achieve similar linear scalability as Calvin and is therefore able to maintain (and extend) its performance advantage at scale. However, both the VLL and Calvin schemes did not achieve perfect linear scalability under high contention. This is because both systems experienced increased *execution progress skew* with an increased number of machines. Each machine occasionally falls behind briefly in execution due to random variation in workloads, random fluctuations in RPC latency, etc., slowing down other machines; the more machines there are, the more likely that at least one machine is experiencing this at any time. Under higher contention and with more distributed transactions, the more sensitivity there is to other machines falling behind and being slow to serve those reads, so the effects of execution progress skew are more pronounced in these scenarios, resulting in performance degradation as more machines are added.

The 2PL + 2PC scheme scales as gracefully as the VLL and Calvin schemes under low contention. However, performance degrades more steeply under high contention. In addition to being affected by execution progress skew, the 2PL + 2PC scheme was vulnerable to an increase in distributed deadlocks with increasing contention. These deadlocks further increase the contention and significantly degrade the scalability of the system.

H-Store also scales poorly, since all distributed transactions need to be executed serially.

3.4 VLLR evaluation

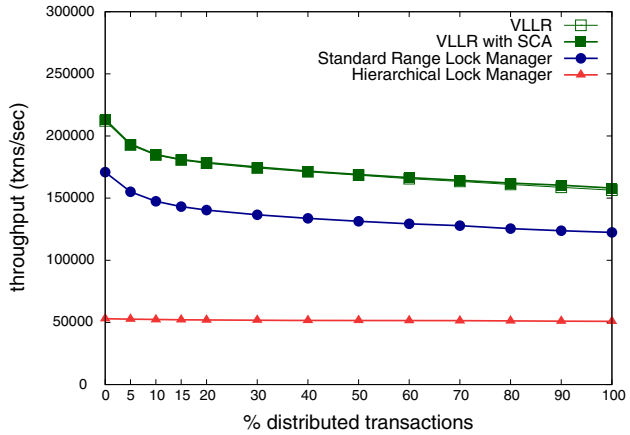
In this section, we evaluate the performance VLLR (as described in Sect. 2.7), with and without SCA. We first compare VLLR to traditional mechanisms for locking ranges. Then, we compare VLLR to the basic VLL algorithm that requests one lock per element in the range.

3.4.1 VLLR versus traditional mechanisms for locking ranges

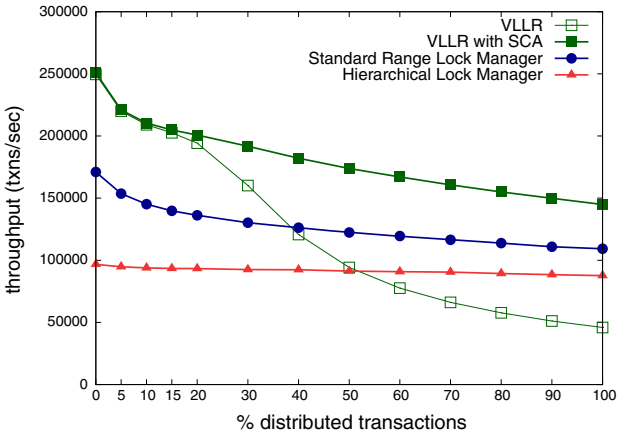
This section compares VLLR against two traditional mechanisms for locking ranges. Our first baseline ("Standard Range Lock Manager") explicitly maps key ranges to lock request queues—fragmenting ranges when needed if new ranges are inserted that partially overlap existing ranges. Since this technique requires ranges to be sorted, the underlying data structure uses a `std::map` (which internally implements a red-black tree). Our second baseline ("Hierarchical Lock Manager") is a hash table-based lock manager that also does the same bitwise-prefix hierarchical locking as VLLR, acquiring intention locks for each nonempty prefix of each element of P.

To compare the performance and contention characteristics of these techniques in isolation of other factors,[7] we implemented a single-machine, single-threaded benchmark in which each transaction locked a single range—and then imposed artificial waits on a varying fraction of transactions

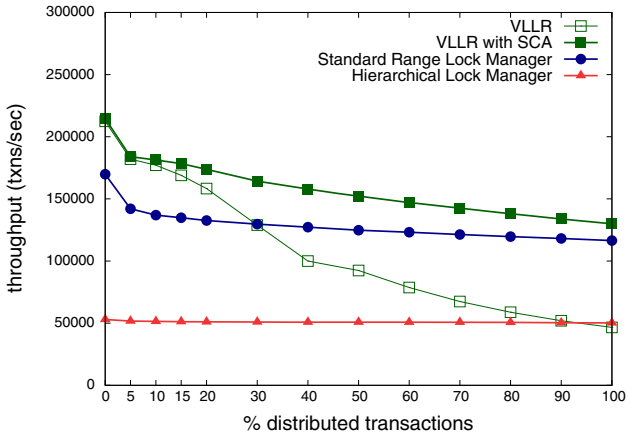
[7] For example, distributed deadlock detection mechanisms differ for Standard Range Locking than for the other mechanisms, because partially overlapping ranges must be split, so the number of lock queues that a transaction's requests are in may increase over time without the transaction requesting new locks.



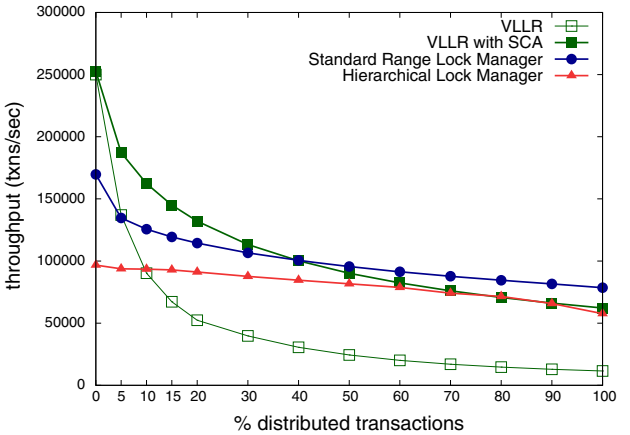
(a) 100-microsecond "distributed transaction" delays, low contention.



(b) 100-microsecond "distributed transaction" delays, high contention.



(c) 500-microsecond "distributed transaction" delays, low contention.



(d) 500-microsecond "distributed transaction" delays, high contention.

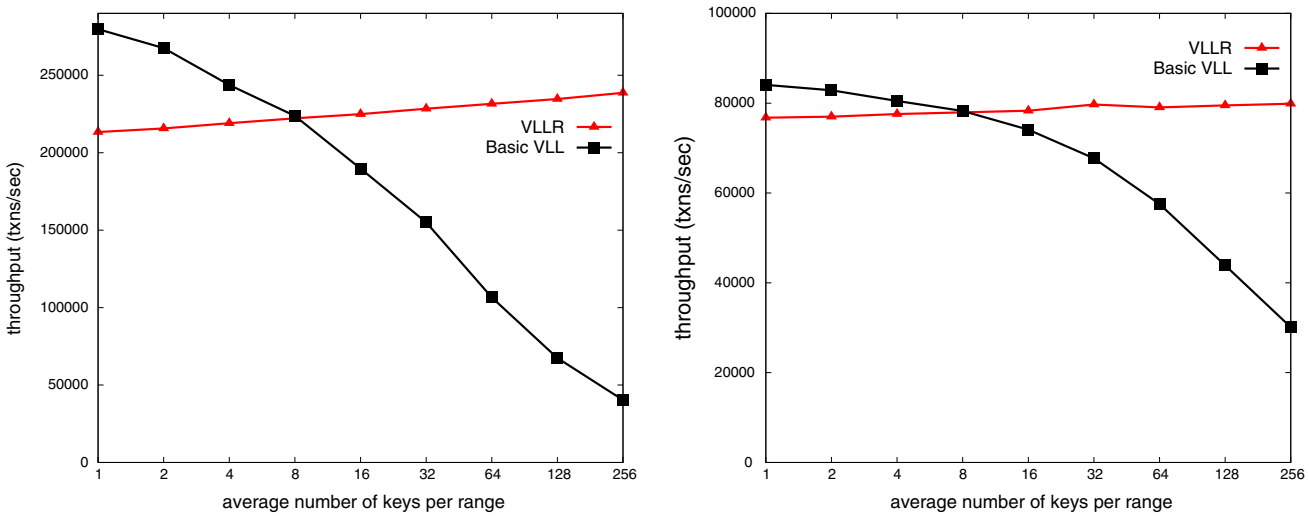
Fig. 14 Range locking throughput with different artificial delays for distributed transactions

to simulate remote reads. To vary contention, we chose ranges whose start and end points, when represented as bitstrings, had shared prefixes of varying length. We chose two scenarios: a low-contention scenario in which the start and limit of each range shared a prefix of on average 15 bits, and a high-contention scenario in which the start and limit of each range shared a prefix of on average 6 bits. For VLLR and the Hierarchical Lock Manager, we translated each key range to a singleton prefix set consisting of only the longest shared prefix between the start and limit. As a result, the Standard Range Lock Manager observed contention rates of 0.0001 and 0.0125, respectively, while VLLR and the Hierarchical Lock Manager observed significantly higher contention rates: 0.0002 and 0.0178. However, under high contention, VLLR and the Hierarchical Lock Manager required significantly fewer intention locks, which resulted in lower CPU overhead. Figure 14 shows our measured results with 100-ms (Fig. 14a, b) and 500-ms (Fig. 14c, d) delays, and with low (Fig. 14a, c) and high (Fig. 14b, d) contention.

Before comparing VLLR to the other schemes, we examine VLLR with and without SCA. Under high contention, SCA proves similarly critical to VLLR as to VLL; it costs little and VLLR's throughput drops significantly without it when there are distributed transaction delays. As with VLL, VLLR benefits little from SCA when transaction stalls are short or infrequent.

The Standard Range Lock Manager incurred higher CPU overhead than VLLR due to red-black tree operations and range splitting, but its performance declined more gracefully when distributed transaction delays were simulated because it experienced lower contention than the prefix-based VLLR implementation (recall that it locks exact key ranges rather than prefixes that represent conservative estimates of key ranges).

The Hierarchical Lock Manager, on the other hand, incurred very high CPU overhead when enqueuing and dequeuing requests for many, many intention locks, significantly limiting throughput. This CPU cost is so severe that it



(a) Short Transactions. **(b)** Longer Transactions.

Fig. 15 VLLR versus basic VLL algorithm

remains the sole throughput bottleneck in Figs. 14a–c, even when 100 % of transactions incur delays during execution. Only under high contention *and* frequent long delays (exactly when the Standard Range Lock Manager outperforms VLLR, illustrating extreme contention costs) is throughput limited by contention.

3.4.2 VLLR versus basic VLL algorithm

We now examine the performance of VLLR versus the performance of the basic VLL algorithm that implements range locks by simply acquiring locks on all records inside the range. In this experiment, we use the same VLLR implementation and experimental transaction from Sect. 3.4.1, with 100 % single-partition transactions. The bitstring size we use in this experiment is 16, and we vary the longest common prefix of the bitstring range from 16 to 7. For example, if the shared prefix is 11, then the five least-significant bits are not shared, which enables $2^5 = 32$ unique values within that range. Since the average range is half of the total possible range size, the average size range for a shared prefix of 11 bits is 16 records. Therefore, by varying the longest common prefix of the bitstring range from 16 to 7, we are in effect experimenting with ranges of average size between 1 and 256 records.

Figure 15a shows the results of our experiment for short transactions and Fig. 15b shows the results for long transactions. When ranges are short, basic VLL outperforms VLLR. This is because VLL only has to lock a few records in the range, while VLLR has to acquire all the hierarchical prefix locks. In other words, VLLR must perform more overall lock acquisitions than VLL. However, when the range gets larger, VLL must acquire more individual record locks, and thus its performance degrades with larger ranges. In contrast, VLLR actually has to acquire fewer locks as the range size increases, since longer ranges result in shorter shared prefixes, and thus fewer hierarchical prefix locks that VLLR needs to acquire. For shorter transactions, the cost of the additional locks that VLL must acquire is a larger relative percentage of the transaction costs, and thus the difference between VLL and VLLR for short transactions with large ranges is more pronounced.

In summary, when the ranges are short, especially in the extreme case of a range of size one (i.e., a "point" access), basic VLL performs well. However, VLLR results in significantly larger throughput for longer range accesses.

3.5 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.

Figure 16 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

locking mechanism	per-transaction CPU cost (μs)
Traditional 2PL	21.07
Deterministic Calvin	20.18
Multi-threaded VLL	1.83
Single-threaded VLL	0.69

Fig. 16 Locking overhead per transaction

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 [9] is the lock manager design that has been adopted by most database systems [13]. In order to reduce the cost of locking, there have been several published methods for reducing the number of lock calls [17,24,28]. While these schemes may partially mitigate the cost of locking in main memory systems, but they do not address the root cause of high lock manager overhead—the size and complexity of the data structure used to store lock requests.

Kimura et al. [20] presented a Lightweight Intent Lock (LIL), which maintains a set of lightweight counters in a global lock table. However, this proposal doesn't colocate 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 colocating a record's lock state with the record itself in main memory databases was proposed almost two decades ago [8,25]. 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.

There has been a lot of effort in improving the implementation of the lock manager in database systems. Horikawa [38], Shore-MT [15], and Jung et al. [18] all improve multicore scalability by carefully optimizing the lock manager, removing latching and critical sections wherever possible. However, these systems all keep the basic two-phase locking design and improve performance by removing bottlenecks in the lock manager. This differs from our design that moves the lock data structures outside of the lock manager and fundamentally changes what (and how) lock information is tracked. DORA [29] partitions the lock manager across cores on a multicore machine, which eliminates long chains of lock waits on a centralized lock manager and increases cache affinity. However, it doesn't change the fundamental hash-based lock manager data structure.

Given the high overhead of the lock manager when a database is entirely in main memory [12,14,22], some researchers observe that executing transactions serially that without concurrency control can buy significant throughput improvement in main memory database systems [16,19,33,40]. 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 [1,3,5,21,22,39], and some industry products also use the these concurrency control schemes, for example, HANA uses MVCC [23], Hekaton uses optimistic MVCC [6], and Google F1 uses OCC (in addition to some pessimistic locking) [31]. 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.

Key range locking, as pioneered by Lomet [26], has been shown to be useful in workloads that frequently operate on many consecutive rows within the same transaction. While not all database systems have implemented key range locking, some of them, such as Spanner [7], use the range locking technique exclusively.

5 Conclusion and future work

We have presented *VLL*, a protocol for main memory database systems that avoid the costs of maintaining the data structures kept by traditional lock managers, and therefore yields higher transactional throughput than traditional implementations. VLL colocates 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, *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 (nondeterministic) approaches to transaction processing and deterministic database systems like Calvin.

We also showed that VLL can be easily extended to support range locking by adding two extra counters and a prefix locking algorithm, and we call this technique VLLR. Experiments demonstrate that VLLR outperforms other alternative range locking approaches.

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.

Acknowledgments We would like to thank the anonymous reviewers for their detailed and insightful comments. This work was sponsored by the NSF under grants IIS-0845643 and IIS-1249722, and by a Sloan Research Fellowship.

References

1. Agrawal, D., Sengupta, S.: Modular synchronization in distributed, multiversion databases: version control and concurrency control. IEEE TKDE **5**, 126–137 (1993)
2. Agrawal, R., Livny, M.J.C.M.: Concurrency control performance modeling: alternatives and implications. ACM Trans. Database Syst. **12**, 609–654 (1987)
3. Mahmoud, H.A., Arora, V., Nawab, F., Agrawal, D., Abbadi, A.E.: Maat: effective and scalable coordination of distributed transactions in the cloud. In: Proceedings of PVLDB, vol. 7, no. 5, (2014)
4. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. **13**(2), 185–221 (1981)
5. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
6. Diaconu, C., Freedman, C., Ismert, E., ke Larson, P., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: Sql server's memory-optimized oltp engine. In: SIGMOD (2013)
7. Corbett, J.C., et al.: Spanner: Googles globally-distributed database. In: Proceedings of OSDI 2012 (2012)
8. Gottemukkala, V., Lehman, T.: Locking and latching in a memory-resident database system. In: VLDB (1992)
9. Gray, J.: Notes on Database Operating Systems. Operating System, An Advanced Course. Springer, Berlin (1979)
10. Gray, J.N., Lorie, R.A., Putzolu, G.R., Traiger, I.L.: Granularity of locks and degrees of consistency in a shared database. In: Proceedings of IFIP Working Conference on Modelling of Database Management Systems (1975)
11. Gray, J., Reuter.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, New York (1993)
12. Harizopoulos, S., Abadi, D.J., Madden, S.R., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: SIGMOD (2008)
13. Hellerstein, J.M., Stonebraker, M., Hamilton, J.: Architecture of a database system. Found. Trends Databases **1**(2), 141–259 (2007)
14. Johnson, R., Pandis, I., Ailamaki, A.: Improving oltp scalability using speculative lock inheritance. PVLDB **2**(1), 479–489 (2009)
15. Johnson, R., Pandis, I., Hardavellas, N., Ailamaki, A., Falsafi, B.: Shore-mt:a scalable storage manager for the multicore era. In: Proceedings of EDBT (2009)
16. Jones, E.P.C., Abadi, D.J., Madden, S.R.: Concurrency control for partitioned databases. In: SIGMOD (2010)
17. Joshi, A.: Adaptive locking strategies in a multi-node data sharing environment. In: VLDB (1991)
18. Jung, H., Han, H., Fekete, A.D., Heiser, G., Yeom, H.Y.: A scalable lock manager for multicores. In: Proceedings of SIGMOD (2013)
19. Kemper, A., Neumann, T., Finis, J., Funke, F., Leis, V., Muhe, H., Muhlbauer, T., Rodiger, W.: Transaction processing in the hybrid OLTP/OLAP main-memory database system hyper. IEEE Data Eng. Bull. **36**(2), 41–47 (2013)
20. Kimura, H., Graefe, G., Kuno, H.: Efficient locking techniques for databases on modern hardware. In: Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (2012)
21. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. TODS **6**(2), 213–226 (1981)
22. Larson, P., Blanas, S., Diaconu, C., Freedman, C., Patel, J., Zwilling, M.: High-performance concurrency control mechanisms for main-memory database. In: PVLDB (2011)
23. Lee, J., Muehle, M., May, N., Faerber, F., Sikka1, V., Plattner, H., Krueger, J., Grund, M.: High-performance transaction processing in sap hana. IEEE Data Eng. Bull. **36**(2), 28–33 (2013)
24. Lehman, T.: Design and performance evaluation of a main memory relational database system. Ph.D. thesis, University of Wisconsin-Madison (1986)
25. Lehman, T.J., Gottemukkala, V.: 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, Reading (1996)
26. Lomet, D.: Key range locking strategies for improved concurrency. In: Proceedings of VLDB (1993)
27. Mohan, C., Lindsay, B.G., Obermarck, R.: Transaction management in the r* distributed database management system. ACM Trans. Database Syst. **11**(4), 378–396 (1986)
28. Mohan, C., Narang, I.: Recovery and coherency-control protocols for fast inter-system page transfer and fine-granularity locking in shared disks transaction environment. In: VLDB (1991)
29. Pandis, I., Johnson, R., Hardavellas, N., Ailamaki, A.: Data-oriented transaction execution. PVLDB **3**(1), 928–939 (2010)
30. Ren, K., Thomson, A., Abadi, D.J.: Lightweight locking for main memory database systems. In: PVLDB (2013)
31. Shute, J., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., Oancea, M., Littleeld, K., Menestrina, D., Ellner, S., Cieslewicz, J., Rae, I., Stancescu, T., Apte, H.: F1: a distributed sql database that scales. In: VLDB (2013)
32. Stonebraker, M.: Concurrency control and consistency of multiple copies of data in distributed ingres. IEEE Trans. Softw. Eng. **SE-5**, 188–194 (1979)
33. Stonebraker, M., Madden, S.R., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era (it's time for a complete rewrite). In: VLDB, Vienna, Austria (2007)
34. Thomasian, A.: Two-phase locking performance and its thrashing behavior. TODS **18**(4), 579–625 (1993)
35. Thomson, A., Abadi, D.J.: The case for determinism in database systems. In: VLDB (2010)
36. Thomson, A., Abadi, D.J.: Modularity and scalability in calvin. IEEE Data Eng. Bull. **36**(2), 48–55 (2013)
37. Thomson, A., Diamond, T., Shao, P., Ren, K., Weng, S.-C., Abadi, D.J.: Calvin: Fast distributed transactions for partitioned database systems. In: SIGMOD (2012)
38. Horikawa, T.: Latch-free data structures for dbms: design, implementation, and evaluation. In: Proceedings of SIGMOD (2013)

39. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: Proceedings of SOSP, SOSP '13, pp. 18–32 (2013)

40. Whitney, A., Shasha, D., Apter, S.: High volume transaction processing without concurrency control, two phase commit, SQL or C++. In: HPTS (1997)