Rethinking serializable multiversion concurrency control

Jose M. Faleiro
Yale University
jose.faleiro@yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

Multi-versioned database systems have the potential to significantly increase the amount of concurrency in transaction processing systems because they can avoid reads-write conflicts. Unfortunately, these increases in concurrency usually come at the cost of transaction serializability. If a database user requests full serializability, modern multi-versioned systems significantly constrain their read-write concurrency. In main-memory, multi-core settings these additional constraints are so significant, that multi-versioned systems are often *outperformed* by single-version systems.

We propose Bohm, a new concurrency control protocol for main-memory multi-versioned database systems. Bohm guarantees serializable execution while ensuring that reads *never* block writes. In addition, Bohm does not require reads to perform any book-keeping whatsoever, thereby avoiding the overhead of tracking reads via contended writes to shared memory. This leads to excellent scalability and performance in multi-core settings. Bohm has all the above characteristics without performing validation based concurrency control. Instead, it is pessimistic, and is therefore not prone to excessive aborts in the presence of contention. An experimental evaluation shows that Bohm performs well in both high contention and low contention settings, and is able to dramatically outperform single-versioned systems despite maintaining the full set of serializability guarantees.

1. INTRODUCTION

Database systems must choose between two alternatives for handling record updates: (1) overwrite the old data with the new data ("update-in-place systems") or (2) write a new copy of the record with the new data, and delete or reduce the visibility of the old record ("multi-versioned systems"). The primary advantage of multi-versioned systems is that transactions that writes to a particular record can proceed in parallel with transactions that read the same record; read transactions do not block write transactions since they can read older versions until the write transaction has committed. On the other hand, multi-versioned systems must consume additional space to store the extra versions, and incurs additional complexity to maintain them. As space becomes increasingly "cheap" in modern hardware configurations, the balance is shifting, and the majority of recently architected database systems are choosing the multi-versioned approach.

While it is well-known how to implement concurrency control techniques that guarantee serializability in database systems that use locking to preclude write-write and read-write conflicts, it is much harder to guarantee serializability in multi-versioned systems that enable reads and writes of the same record to occur concurrently. One popular option that achieves a level of isolation very close to full serializability is "snapshot isolation" [6]. Snapshot isolation guarantees that all transactions read the database state resulting from all transactions that committed before that transaction began, while also not reading any state that was produced by transactions running concurrently with it. Snapshot isolation comes very close to fully guaranteeing serializability, and indeed, highly successful commercial database systems (such as older versions of Oracle) implement snapshot isolation when the user requests the "serializable" isolation level [18]. However, snapshot isolation is vulnerable to serializability violations [6, 15]. For instance, the famous write-skew anomaly can occur when two transactions have an overlapping read-set and disjoint write-set, where the write-set (of each transaction) includes elements from the shared read-set [6]. Processing such transactions using snapshot isolation can result in a final state that cannot be produced if the transactions are processed serially.

There has been a significant amount of work on how to make multi-versioned systems serializable, either by avoiding the write-skew anomaly in snapshot isolation systems [13, 14], or by using alternative concurrency control protocols to snapshot isolation [10, 21]. However, these solutions either introduce new read-write conflicts (despite the fact that the ability to avoid read-write conflicts is a major reason why multi-valued systems are popular) or they require more coordination and book-keeping which reduce the performance of the system in other ways.

In this paper, we start from scratch, proposing a new concurrency control protocol for multi-versioned database systems, which we call Bohm, that guarantees full serializability while still ensuring that reads do not block writes. Furthermore, Bohm does not require the additional coordination and book-keeping introduced by other methods for achieving serializability in multi-versioned systems. The final result is perhaps the most scalable (across multiple cores) concurrency control protocol ever proposed — there is no centralized lock manager, almost all data structures are thread-local, no coordination needs to occur across threads except at the end of a large batch of transactions, and the need for latching or any kind of atomic instructions is therefore minimized.

The major disadvantage of our approach is that entire transactions must be submitted to the database system before the system can begin to process them — traditional cursor-oriented database access, where transactions are submitted to the database in pieces, are therefore not supported. Furthermore, the write-set of a transaction must be deducible before the transaction begins — either through explicit write-set declaration by the program that submits the transaction, or through analysis of the transaction by the database system, or through optimistic techniques that submit a transaction for a trial run to get an initial guess for its write-set, and abort the transaction if the trial run resulted in an incorrect prediction [33, 30].

Although these disadvantages (especially the first one) change the model by which a user submits transactions to a database system, an increasingly large number of performance sensitive applications are already utilizing stored-procedures to submit transactions to database systems in order to avoid paying round-trip communication costs to the database server. These applications can leverage our multiversioned concurrency control technique without any modifications.

Bohm thus presents a new, interesting alternative in the space of concurrency control options — an extremely scalable technique, at the cost of requiring entire transactions with deducible write-sets in advance. Experiments show that Bohm achieves linear scalability up to (at least) 20 million record accesses per second with transactions being processed over dozens of cores.

In addition to contributions around multi-versioned serializability and multi-core scalability, a third important contribution of Bohm is a clean, modular design. Whereas traditional database systems use a monolithic approach, with the currency control and transaction processing components of the systems heavily cross-dependent and intertwined, Bohm completely separates these system components, with entirely separate threads performing concurrency control and transaction processing. This modular design is made possible by Bohm's philosophy of planning transaction execution in advance, so that when control is handed over to the execution threads, they can proceed without any concern for other concurrently executing transactions. This architecture greatly improves database engine code maintainability and reduces database administrator complexity.

This paper proceeds as follows. In Section 2 we discuss related work to our approach. In Section 3 we go into more detail into particular areas of related work that motivate our design. In Section 4 we describe the design of Bohm, and in Section 5 we experimentally evaluate it. We conclude in Section 6.

2. RELATED WORK

2.1 Concurrency Control Protocols

Timestamp ordering is a concurrency control technique in which the serialization order of transactions is chosen a-priori [7, 8]. The serialization order is decided by assigning each transaction a monotonically increasing timestamp. In order to commit, conflicting transactions must execute in an order that is consistent with their timestamps. If a transaction is inconsistently ordered with respect to other conflicting transactions, it must abort. Reed designed a multiversion concurrency control protocol based on timestamp ordering [28]. As in single-version timestamp ordering, transactions are assigned timestamps a-priori. Unlike single-version timestamp ordering, reads are always successful. However, readers may cause writers to abort, and the database needs to track the timestamp of each read in order to abort writers. In contrast to timestamp ordering based concurrency control, Bohm's design avoids any kind of tracking when a transaction reads the value of a record in the database. By eliminating writes to shared-memory, Bohm greatly improves multi-core scalability.

Snapshot Isolation (SI) is a multiversion concurrency control protocol which guarantees that transactions that read the value of a particular record never block transactions that write the record, and vice-versa [6]. SI, therefore, allows transactions to execute with more concurrency than protocols such as timestamp ordering and two-phase locking. This increase in concurrency, however, comes at the cost of serializable execution. Effectively, SI does not track anti-dependencies between transactions [1], and is thus susceptible to serializability violations. Unlike SI, Bohm guarantees serializability. Bohm guarantees that reader transactions never block writer transactions, but, unlike SI, *does not* guarantee the converse.

2.2 Multi-versioned Systems

Serializable Snapshot Isolation (SSI) is a modification to snapshot isolation that guarantees serializability [10]. Cahill et al. added support for SSI to BerkeleyDB [26]. SSI is based on the following fact about serializability violations in the context of snapshot isolation: if a serializability violation occurs, then the corresponding serialization graph of transactions contains two consecutive anti-dependency edges [14]. SSI, therefore, tracks anti-dependencies among concurrent transactions on the fly; whenever SSI finds a sequence of two consecutive anti-dependency edges, it aborts a transaction in order to break the sequence. In contrast, Bohm ensures that the serialization graph does not contain cycles *prior* to actually executing transactions; consequently, Bohm does not need to track any anti-dependency edges, and does not impose restrictions on the number of consecutive anti-dependency edges in the serialization graph of transactions. This greatly reduces Bohm's runtime overhead and improves its scalability.

Larson et al. propose techniques for optimistic and pessimistic multiversion concurrency control in the context of main-memory databases [21]. Their techniques address several limitations of traditional systems; their optimistic validation technique does not require the use of a global critical section, while their pessimistic system avoids the use of a centralized lock-manager. However, their design uses a global counter to generate timestamps (that is accessible to many different threads), and thus inherits the scalability bottlenecks associated with contended global data-structures. Furthermore, despite the improvements in design, their multiversioned system is significantly outperformed by single-version locking on many OLTP workloads. Bohm makes additional scalability optimizations, and is thus able to outperform (or at least perform comparably) to single-version locking on those same workloads.

2.3 Multi-core Systems

Silo is a database system designed for main-memory, multicore machines [34]. Silo implements a variant of optimistic concurrency control, and uses a timestamp based technique to validate transactions at commit time. Silo uses a decentralized technique for assigning timestamps to transactions and avoids all writes to shared memory for transactions' reads. Bohm shares some of Silo's design principles: it uses a low contention technique to generate timestamps to decide the relative ordering of conflicting transactions, and does not require any writes to shared memory for records in a transaction's readset. Unlike Silo, however, Bohm does not use optimistic concurrency control. Therefore, it is able to perform much better on high-contention workloads for which optimistic concurrency control leads to many aborts and performs poorly.

Pandis et al. propose a data-oriented architecture (DORA) in order to eliminate the impact of contended accesses to shared memory by transaction execution threads [27]. DORA partitions a database among several physical cores of a multicore system. DORA executes a disjoint subset of each tranasctions's logic on multiple threads, effectively using a form of intra-transaction parallelism. Bohm uses intra-transaction parallelism to decide the *order* in which transactions must execute. However, the execution of a transaction's logic still occurs on a single thread.

Jung et al. propose techniques for improving the scalability of lock-managers [20]. Their design includes the pervasive use of the read-after-write pattern [5] in order to avoid repeatedly "bouncing" cache-lines due to cache-coherence [4, 24]. In addition, to avoid the cost of reference counting locks, they use a technique to *lazily* de-allocate locks in batches. Bohm similarly refrains from the use of reference counters to garbage collect versions of records that are no longer visible to transactions.

Johnson et al. identified contention for updating lock manager state of hot locks as a scalability bottleneck in multicore databases [19]. Updating the state of a hot lock involves calling the lock manager's *acquire* or *release* method, which in turn acquires a contended latch. The performance of a contended latch degrades as the number of threads contending for access to the latch increases. They proposed Speculative Lock Inheritance (SLI), a technique to reduce the number of contended latch acquisitions. SLI passes *hot* locks from transaction to transaction without requiring calls to the lock manager, consequently, reducing the number of contended latch acquisitions. SLI effectively *amortizes* the cost of contended latch acquisitions across a batch transactions that pass locks to each other. Bohm similarly amortizes synchronization across batches of transactions in order to scale concurrency control.

Very lightweight locking (VLL) reduces lock-manager overhead by co-locating concurrency control related meta-data with records [29]. VLL is not designed for systems with large number of cores because every transaction must execute a global critical section before it can execute. The design of Bohm's partitioned concurrency control layer could be adapted to improve VLL's scalability.

2.4 Deterministic Systems

In our own previous work, we describe a technique for *lazily* evaluating transactions in the context of deterministic database systems [12]. This lazy database design separates concurrency control from transaction execution — a design element that is shared by Bohm. However, Bohm does not process transactions lazily, and is far more scalable due to its use of intra-transaction parallelism, and avoiding writes to shared memory on reads. Furthermore, Bohm is designed to be a generic multi-versioned concurrency control technique, and is motivated by existing limitations in multiversion concurrency control systems.

Calvin [32] and H-Store [31] are both deterministic database systems that execute transactions according to a pre-defined total order. Calvin uses deterministic transaction ordering to reduce the impact of distributed transactions on scalability. Although similar to Bohm with its focus on scalability, Calvin is a single-versioned system and uses locking to avoid read-write and write-write conflicts, while Bohm is multiversioned and ensures that reads do not block writes. Furthermore, Calvin is focused on horizontal shared-nothing scalability, while Bohm is focused on multi-core scalability.

H-Store [31] uses a shared-nothing architecture consisting of single-threaded partitions in order reduce the impact of lock-manager overhead [17], and logging overhead [22]. However, performance degrades rapidly if a workload contains multi-partition transactions. Furthermore, suboptimal performance results if some partitions have more work to do than others. Bohm achieves scalability without doing a hard-partitioning of the data — it is thus less susceptible to skew problems and does not suffer from the multi-partition transaction problem.

3. MOTIVATION

As compared to their single version counterparts, multiversion database systems can execute transactions with greater concurrency. A transaction, T_r, which reads record x need not block a concurrent transaction, T_w, which performs a write operation on record x. In order to avoid blocking T_w, T_r can read a version of x, x_{old} that exists *prior* to the version produced by T_w's write, x_{new}. More generally, multiversioning allows transactions with conflicting read and write sets to execute without blocking each other. Unfortunately, if conflicting transactions are processed without restraint, the resulting execution may not be serializable. In our example, if T_r is allowed to read x_{old}, then it must be ordered before T_w in the serialization order.

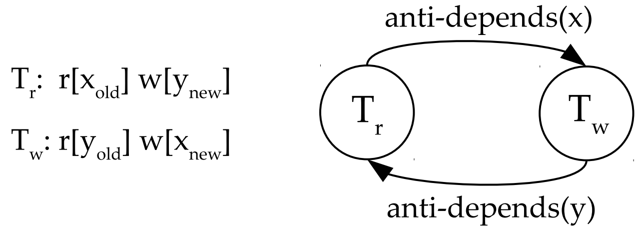

Figure 1: Non-serializable interleaving, and corresponding serialization graph of T_r and T_w

In the formalism of Adya et al. [1], the serialization graph corresponding to the above execution contains an *anti-dependency edge* from T_r to T_w. In order for an execution of transactions to be serializable, the serialization graph corresponding to the trace of the execution cannot contain cycles. If T_r were to write record y, and T_w read y (in addition to T_r's read of x and T_w's write of x), then the relative order of T_r and T_w with respect to their operations on record y must be the same as their order with respect to operations on record x. In particular, T_w cannot read the version of y prior to T_r's write, y_{old}. If T_w were to read y_{old}, then the serialization graph would contain an anti-dependency edge from T_w to T_r.

Figure 1 shows the the interleaved execution of transactions T_r and T_w, and the corresponding serialization graph. $r[x_1]$ denotes to a *read* of version 1 of record x, correspondingly, $w[x_1]$ denotes a *write* to record x, which produces version 1. A record's subscript corresponds to the version read or written by the transaction. As Figure 1 shows, the serialization graph of T_r and T_w contains two anti-dependency edges, one from T_r to T_w, and the other from T_w to T_r; these two edges form a cycle, implying that the interleaving of T_w and T_r as described above is not serializable. This example is a variant of Snapshot Isolation's well known *write-skew* anomaly [6].

In order to avoid non-serializable executions such as the one described above, multiversioned database systems need to account for anti-dependencies among transactions whose read and write sets conflict. These systems account for anti-dependencies in one of two ways:

- **Track Reads** Whenever a transaction reads a record, they track the fact that the transaction performed the read by associating some metadata with each record in the database. The read metadata associated with records in the database system is then used to decide on the order of transactions. For instance, the pessimistic version of Hekaton's multiversion concurrency control algorithm associates a counter with every record in the database [21]. The counter reflects the number of in-flight transactions that have read the record. As another example, Cahill et al. modify BerkeleyDB's lock manager to track anti-dependency edges to and from a particular transaction [10].

- **Validate Reads** A transaction locally keeps track of the version of each record it observed. When the transaction is ready to commit, it validates that the reads it observed are consistent with a serial order. This technique is used technique is used by Hekaton's optimistic concurrency control protocol [21], and Multiversion General Validation [2].

While both approaches above ensure that all executions are serializable, they come at a cost. Concurrency control protocols track reads in order to *constrain* the execution of concurrent readers and writers. For instance, Hekaton's pessimistic concurrency control protocol does not allow a writer to commit until all concurrent readers have either committed or aborted [21]. In addition to the reduction in concurrency due to the concurrency control protocol itself, tracking reads entails writes to shared memory. If a record is popular, then many different threads may attempt to update the same memory words concurrently, leading to contention for access to internal data structures, and cache coherence slow-downs. Since *reads* are tracked, this contention is present even if the workload is read-only.

The "Validate Reads" approach does not suffer from this problem of requiring reads to write to shared memory. However, validation protocols reduce concurrency among readers and writers by aborting readers. Such a situation runs counter to the original intention of multi-version concurrency control, because allowing multiple versions of a record is supposed to allow for greater concurrency among readers and writers.

In order to address the limitations above, we designed Bohm's concurrency control protocol with the following goals in mind: (1) a transaction, T_r, which reads the value of a particular record should *never* block or abort a concurrent transaction that writes the record (Bohm does not require T_r to be read-only transaction), (2) reading the value of a record should not require any writes to shared memory.

It should be noted that snapshot isolation ensures both (1) and (2), and in addition, also guarantees the converse of (1): a transaction that writes a particular record never blocks a concurrent transaction that reads the value of the record. Thus, Snapshot Isolation is able to execute transactions with greater concurrency than Bohm. However, Snapshot Isolation does not guarantee serializable execution of transactions, whereas Bohm always executes transactions in a serializable fashion.

4. DESIGN

Bohm's design philosophy is to eliminate or reduce coordination among database threads due to synchronization based on writes to shared memory. Bohm ensures that threads either make decisions based on local state, or amortize the cost of coordination across several transactions. Bohm achieves this goal by separating concurrency control logic from the transaction execution logic. This separation is reflected in Bohm's architecture: a transaction is processed by two different sets of threads in two phases: (1) a concurrency control phase which determines the proper serialization order and creates a data structure that will enable the second phase to process transactions without concern for concurrently executing transactions, and (2) an execution phase, which actually executes transactions's logic.

While the separation of concurrency control logic and transaction execution logic allows Bohm to avoid scalability bottlenecks, it comes at the cost of extra requirements. In particular, in order to plan execution correctly, the concurrency control phase needs to know in advance which records a transaction is going to write. This requirement is not unique to Bohm — several prior systems exploit a priori information about transactions's read- and/or write-sets [3, 12, 33, 25]. These previous papers have shown that even though they need transactions' write- (and sometimes also read-) sets in advance, it is not necessary for transactions to predeclare these read-/write-sets. For example, Calvin proposes a speculative technique which predicts transactions's read-/write-sets on the fly [33]. Bohm can make use of this technique if transactions' write-sets are not available (or derivable) in advance. However, either way, there is a requirement that the entire transaction be submitted to the system at once. Thus, cursor-oriented transaction models that a submit a transaction to the system in pieces cannot be supported.

4.1 Concurrency Control

The concurrency control layer is responsible for (1) determining the serialization order for transactions (2) creating a safe environment in which the execution phase can run transactions without concern for other transactions running concurrently.

First, it assigns each transaction a unique, monotonically increasing timestamp. This is done in a thread-local manner to avoid thread contention for a shared counter. Second, for every record in a transaction's writeset, it writes out a new version of the record v. A version created by the concurrency control phase does not (yet) contain a value, instead, the version contains a reference to the transaction which must be executed to produce the value.

4.1.1 Timestamp Assignment

The concurrency control layer assigns each transaction that enters the system a unique timestamp. Because the concurrency control layer is separated from execution, and is run by a completely different group of threads, Bohm is able to use a single thread dedicated solely to the purpose of assigning transactions their timestamps. The thread assigns each transaction a new timestamp based on the value of a local counter; when a new transaction enters the system, the thread increments the value of the counter and assigns the incremented value to the new transaction. There is no contention for the value of the counter, since it is only ever accessed by a single thread. This is an example of the philosophy of our design — Bohm avoids writing and reading from shared data-structures as much as possible.

Several prior multiversion concurrency control mechanisms assign each transaction, T, two timestamps, t_{begin} and t_{end} [2, 6, 10, 21]. t_{begin} determines which versions of pre-existing records are visible to T, while t_{end} determines the logical time at which T's writes become visible to other transactions, and to validate whether T can commit. The time between t_{begin} and t_{end} determines the logical interval of time during which T executes. If another transaction's logical interval overlaps with that of T, then the database system needs to ensure that the transactions do not conflict with each other (what exactly constitutes a conflict depends on the isolation level desired). In contrast, Bohm assigns each transaction a single timestamp, ts. Intuitively, ts "squashes" t_{begin} and t_{end} together; ts determines both the logical time at which T performs its reads, and the logical time at which T's writes are visible to other transactions. As a consequence, each transaction appears to execute atomically at time ts.

4.1.2 Inserting Placeholders

Once a transaction's timestamp has been determined, the concurrency control layer inserts a new version for every record in the transaction's writeset. The version inserted by the concurrency control layer contains a placeholder for the value of the version, but the value is uninitialized. The actual value of the version is only produced once the corresponding transaction's logic is executed by the execution layer (Section 4.2).

Several threads contribute to the processing of a single transaction's writeset. Bohm partitions the responsibility for each record of a table across the set of concurrency control threads. When the concurrency control layer receives a transaction, *every* concurrency control thread examines T's writeset in order to determine whether any records belong to the partition for which it is responsible.

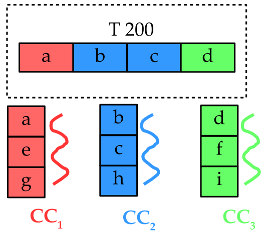

Figure 2: Intra-transaction parallelism. Transaction 200 which writes 4 records is shown in the upper rectangle. The logical partitioning of concurrency control thread responsibility is shown below.

Figure 2 illustrates how several threads cooperatively process each transaction. The transaction is assigned a timestamp of 200, and its writeset consists of records a, b, c, and d. The concurrency control layer partitions records among three threads, CC_1, CC_2, and CC_3. CC_1's partition contains record a, CC_2's partition contains records b and c, and CC_3's partition contains record d. CC_1 thus inserts a new version for record a, CC_2 does the same for records b and c, and CC_3 for d. Bohm uses several threads to process a *single* transaction, a form of *intra-transaction* parallelism.

Every concurrency control thread must check whether a transaction's writeset contains records that belong to its partition. For instance, if record d belonged to CC_1's partition instead of CC_3's, CC_3 would still have to check the transaction's writeset in order to determine that no records in the transaction's writeset map to its partition.

This design is consistent with our philosophy that concurrency control threads should *never* need to coordinate with each other in order to process a transaction. Each record is *always* processed by the same thread; two concurrency control threads will never try to process the same record, even across transaction boundaries. The decision of which records of a transaction's writeset to process is a purely *thread local* decision; a concurrency control thread, CC, will process a particular record only if the record's key resides in CC's partition.

Not only does this lead to reduced cache coherence traffic, but it also leads to multi-core scalability. As we dedicate more concurrency control threads to processing transactions, throughput increases for two reasons. First, each transaction is processed by a greater number of concurrency control threads, which leads to an increase in intra-transaction parallelism. Since the concurrency control threads do not need to coordinate with each other, there is little downside to adding additional threads as long as there enough processing resources on which they can run. Second, for a fixed database size, the number of keys assigned to each thread's partition *decreases*. As a consequence, each concurrency control thread will have a smaller cache footprint.

One impediment to scalability is the fact that every concurrency control thread must examine every transaction that enters the system. This is logic which is effectively executed serially, since every concurrency control thread runs the same piece of logic. Increasing the number of concurrency control threads beyond a certain point will therefore yield a diminishing increase in throughput due to Amdahl's law. Although we have not encountered this scalability bottleneck in our experimental evaluation, a straightforward mechanism around the issue is to "route" each transaction to the appropriate concurrency control threads by pre-analyzing its write-set. This routing layer would increase latency, but would remove this scalability bottleneck.

4.1.3 Processing a single transaction's read/write set

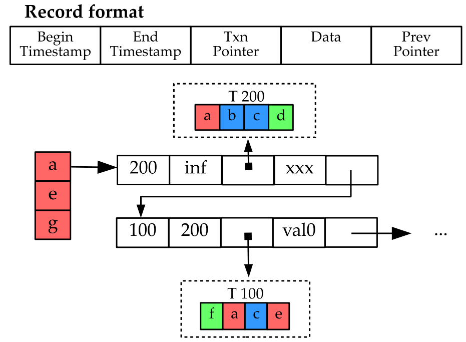

Figure 3: Inserting a new version

For each record in a transaction's **write-set**, the concurrency control phase produces a new version to hold the transaction's write. Figure 3 shows the format of a record's version. Each version consists of the following fields:

- **Begin Timestamp.** The timestamp of the transaction that created the record.
- **End Timestamp.** The timestamp of the transaction that invalidated the record.
- **Txn Pointer.** A reference to the transaction that must be executed in order to obtain the value of the record.
- **Data.** The actual value of the record.
- **Prev Pointer.** A reference to the version of the record that precedes the current version.

When inserting a new version of a record, the concurrency control thread sets the version's fields as follows: (1) the version's start timestamp is set to the timestamp of the transaction that creates the version, (2) the version's end timestamp is set to infinity, (3) the version's txn pointer is set to the transaction that creates the version, (4) the version's data is left uninitialized, (5) the version's prev pointer is set to the preceeding version of the record.

Figure 3 shows the thread CC_1 inserting a new version of record a, which is produced by transaction T_{200}. CC_1 sets the new version's begin timestamp to 200, and its end timestamp to infinity. The version's txn pointer is set to T_{200} (since T_{200} produces the new version). At this point, the version's data has not yet been produced; Bohm needs to execute T_{200} in order to obtain the value of the version.

While inserting a new version of record a, CC_1 finds that a previous version of the record exists. The older version of a was produced by transaction T_{100}. CC_1 sets the new version's prev pointer to the old version, and sets the old version's end timestamp to 200, the timestamp of the transaction that created the new version.

In order to create a new version of a record, Bohm does not need to synchronize concurrency control threads. Bohm partitions the database among concurrency control threads such that a record is *always* processed by the same thread, even across transaction boundaries (Section 4.1.2). One consequence of this design is that there is no contention in the concurrency control phase. The maintenance of the pointers to the current version of every record can be done in a thread-local data-structure; thus the lookup needed to populate the prev pointer in the new versions is thread-local. Furthermore, if multiple transactions update the same hot record, the corresponding new versions of the record are written by the *same* concurrency control thread, thereby avoiding cache coherence slow-downs.

For each element in the transaction's **read-set**, Bohm needs to identify the corresponding version that the transaction will read. In general, the concurrency control phase does not need to get involved in processing a transaction's read-set. When an execution thread that is processing a transaction with timestamp ts wants to read a record in the database, it can find the correct version of the record to read by starting at the latest version of the record, and following the chain of linked versions (via the prev pointer field) until it finds a version whose begin and end timestamps are of the form $t_{begin} \leq ts$ and $t_{end} \geq ts$. If no such version exists, then the record is not visible to the transaction.

While the above-described technique to find the right version of a record to read is correct, the cost of the traversal of prev pointers may be non-trivial if the linked list of versions is very long. Such a situation may arise if a record is popular and updated often. An optimization to eliminate this cost is possible if the concurrency control phase has advanced knowledge of the read-sets of transactions (in addition to the write-set knowledge it already requires). In this case, for every record a transaction will read, concurrency control threads annotate the transaction with a reference to the correct version of the record to read. This is a low-cost operation for the concurrency control threads since the correct version is simply the most recent version at the time the concurrency control thread is running[1]. In particular, if a record in a transaction's readset resides on a concurrency control thread's logical partition, the thread looks up the latest version of the record and writes a reference to the latest version in a memory word reserved in advance within the transaction. The concurrency control thread *does not* track the read in the database, it merely gives the transaction a reference to the latest version of the record as of the transaction's timestamp.

A consequence of Bohm's design is that a transaction's reads do not require any writes to shared memory. Even for the optimization mentioned above, the write containing the correct version reference for a read is to pre-allocated space for that record within a transaction, and is uncontended since no other concurrency control thread is responsible for that record. In contrast, pessimistic multiversion systems such as Hekaton [21], and Serializable Snapshot Isolation [10] need to coordinate a transaction's reads with concurrent transaction's writes in order to avoid anti-dependency induced serializability violations.

We now sketch an argument for why Bohm's concurrency control protocol guarantees serializable execution of transactions. Given any pair of transactions T_0 and T_1 with timestamps ts_0 and ts_1, Bohm ensures the following invariant on the resulting transaction serialization graphs[2]: if $ts_0 < ts_1$, then the serialization graph contains no dependencies from T_1 to T_0. Effectively, this invariant implies that the timestamp order of transactions is equivalent to their serialization order.

Serialization graphs may contain three kinds of dependencies among transactions: write-write (ww) dependencies, write-read (wr) dependencies, and read-write (rw) dependencies [1]. Bohm guarantees that the invariant holds for each kind of dependency:

- **ww dependencies.** In order for a ww dependency to occur from T_1 to T_0, their writesets must overlap, and T_1's write must precede T_0's write. For each record r in the overlapping part of the write-sets, a single concurrency control thread will write out versions of record r (Section 4.1.2). In addition, concurrency control threads always process transactions in timestamp order. Since $ts_0 < ts_1$, the appropriate concurrency control thread thread will always write out T_0's version before T_1's version. Therefore, T_1's write can never precede T_0's. Thus there exist no ww dependencies from T_1 to T_0.

- **wr dependencies.** There exists a wr dependency from T_1 to T_0 if T_0 observes the effects of a write by T_1. A version of a record whose respective begin and end timestamps are t_{begin} and t_{end}, is visible to T_0 if $t_{begin} \leq ts_0$ and $t_{end} \geq ts_0$.

  When T_1 updates a record, Bohm changes t_{end} from *infinity* to ts_1, while t_{begin} is unaffected. Prior to T_1's update, $t_{end} > ts_0$ (since t_{end} is *infinity*). After T_1's update, $t_{end} > ts_0$ because $ts_1 > ts_0$. T_1's update does not affect T_0's visibility of the record because $t_{end} > ts_0$ both before and after T_1's update. The argument for inserts and deletes follows along similar lines.

- **rw dependencies.** In order for an rw dependency to occur from T_1 to T_0, T_1 must read a record r that T_0 writes, and T_0's write must occur *after* T_1's read. This implies that T_0 creates a new version of r such that the version's begin timestamp, $t_{begin} > ts_1$. When T_0 creates a new version of a record, it sets the version's t_{begin} to ts_0. Since $ts_0 < ts_1$, $t_{begin} < ts_1$; which contradicts the requirement that $t_{begin} > ts_1$ in order for T_1's read to precede T_0's write.

4.1.4 Batching

Only after a transaction T has been processed by all appropriate concurrency control threads can it be handed off to the transaction execution layer. One naive way of performing this handoff is for the concurrency control threads to notify each other after having processed a transaction by using synchronization barriers. After processing T, each concurrency control thread enters a global barrier in order to wait for all other threads to finish processing T. After all threads have entered this barrier, each concurrency control thread can begin processing the next transaction.

Unfortunately, processing transactions in this fashion is extremely inefficient. Threads need to synchronize with each other on every transaction, which has the effect of forcing concurrency control threads to effectively executes in lock step. Another issue is that some concurrency control threads are needlessly involved in this synchronization process. Consider a scenario where none of the records in a T's writeset belong to a concurrency control thread, CC's, partition. CC has to wait for every thread in order to move on to the next transaction despite the fact that CC "contributes" nothing to T's processing.

In order to avoid expensive global coordination on every transaction, Bohm *amortizes* the cost of coordination across large batches of transactions. The concurrency control thread responsible for allotting each transaction a timestamp accumulates transactions in a batch. Instead of processing transactions in lock-step, concurrency control threads receive an ordered batch of transactions, b, as input. Each concurrency control thread processes every transaction in b independently, without coordinating with any other threads (Sections 4.1.2, 4.1.3). Once a thread has finished processing every transaction in b, it enters a global barrier, where it waits until all concurrency control threads have finished processing b. As a consequence, the cost of coordination is amortized across a batch of transactions.

Coordinating at the granularity of batches means that some threads may outpace others in processing a batch; a particular thread could be processing the 100^{th} transaction in the batch while another is still processing the 50^{th} transaction. Allowing certain concurrency control threads to outpace others is safe for the same reason that intra-transaction parallelism is safe (Section 4.1.2): Bohm parti-

[1]This is true since concurrency control threads process transactions sequentially (threads derive concurrency by exploiting intra-transaction parallelism).

[2]See Section 3 for a definition of serialization graphs.

tions the database among concurrency control threads such that a record is *always* processed by the same thread, even across transaction boundaries.

4.2 Transaction Execution

After having gone through the concurrency control phase, a batch of transactions is handed to the transaction execution layer. The execution layer performs two main functions: it executes transactions' logic, and incrementally garbage collects versions which are no longer visible due to more recent updates.

4.2.1 Evaluating Transaction Logic

The concurrency control layer inserts a new version for every record in each transaction's writeset. However, the *data* within the version cannot yet be read because the transaction responsible for producing the data has not yet executed; concurrency control threads merely insert placeholders for the data within each record. Each version inserted by the concurrency control layer contains a reference to the transaction that needs to be evaluated in order to obtain the data of the version.

Read Dependencies. Consider a transaction T, whose readset consists of $r_1, r_2, ..., r_n$. T needs to read the correct version of each record in its readset using the process described in Section 4.1.3. However, the data stored inside one or more of these correct versions may not yet have been produced because the corresponding transaction has not yet been executed. Therefore, an execution thread may not be able to complete the execution of T until the transaction upon which T depends has finished executing.

Write Dependencies. Every time the value of a particular record is updated, the concurrency control layer creates a new version of the record, stored separately from other versions. Consider two transactions T_1 and T_2, such that (1) neither transactions' logic contain aborts, and (2) T_1 is processed before T_2 by the concurrency control layer. Both transactions' writesets consist of a single record, x, while their readsets do not contain record x. In this scenario, the concurrency control layer will write out two versions corresponding to record x, one each for T_1's and T_2's update. The order of both transaction's updates is already decided by the concurrency control layer, therefore, T_1 and T_2's execution need not be coordinated; in fact, T_2 could execute *before* T_1, despite the fact that T_1 precedes T_2, and their writesets overlap. Note, however, that in the case where T_2 performs a read-modify-write of record x, then T_2 must wait for the version of x produced by T_1 before it can begin proceed with the write (this is a type of read dependency explained above). If T_2 aborts, then it also needs to wait for T_1. The reason is that in this case, the data written to its version of x is equal to that produced by T_1. Thus, T_2 has a read dependency on T_1.

We now describe how several execution threads execute a batch of transactions handed over from the concurrency control layer. The execution layer receives a batch of transactions in an ordered array $< T_0, T_1, ..., T_n >$. The transactions are partitioned among k execution threads such that thread i is responsible for ensuring transactions T_i, T_{i+k}, T_{i+2k}, and so forth are processed. Thread i does not need to directly execute all transactions it is responsible for — other threads are allowed to execute transactions assigned to i, and i is allowed to execute transactions assigned to other threads. However, before moving onto a new batch of transactions, thread i must ensure that all transactions that it is responsible for in the current batch have been executed.

Each transaction can be in one of three states: **Unprocessed**, **Executing**, and **Complete**. All transactions received from the concurrency control layer are in state **Unprocessed** — this state corresponds to transactions whose logic has not yet been evaluated. A transaction is in state **Executing** if an execution thread is in the process of evaluating the transaction. A transaction whose logic has been evaluated is in state **Complete**.

In order to process a transaction, T, an execution thread, E, attempts to atomically change T's state from **Unprocessed** to **Executing**. E's attempt fails if T is already in state **Executing** or **Complete**. If E's attempt is successful, then Bohm can be sure that E has exclusive access to T; subsequent transactions that try to change T's state from **Unprocessed** to **Executing** will fail.

If, upon trying to read a record, E discovers a read dependency on a version that has yet to be produced, E tries to recursively evaluate the transaction T' which must be evaluated to produce the needed version. If E cannot evaluate T' (because another thread is already processing T') then E sets T's state back to **Unprocessed**. T is later picked up by an execution thread (not necessarily E) which attempts once again to execute the transaction.

After completing all reads and writes for T, E sets T's state to **Complete**.

4.2.2 Garbage Collection

Bohm can be optionally configured to automatically garbage collect all versions that are no longer visible to any active or future transactions. Records that have been "garbage collected" can be either deleted or archived. This section describes how Bohm decides when a version can be safely garbage collected.

Consider a scenario where a transaction T updates a record r, whose version preceeding T's update is v_1. T produces a new version v_2. If there exists an unexecuted transaction, T', whose timestamp precedes that of T, then the version of r visible to T' is v_1. We need to keep version v_1 until all transactions that read version v_1 have finished executing. This intuition leads to the following general condition for garbage collecting old versions of records:

Condition 1: Whenever a transaction updates the value of a particular record, we can garbage collect the preceding version of the record when all transactions that read the preceding version have finished executing.

While **Condition 1** is correct, it requires Bohm to track every transaction that reads each version of a record. However, maintaining this meta-data goes against Bohm's design philosophy of avoiding all writes to shared memory on a read. Instead of waiting for precisely the set of transactions that read a particular version to complete, we can wait for all transactions that precede T to finish executing. This set of transactions is a super-set of the transactions that read the preceding version of record r. Therefore, it is more conservative, but still correct. Instead of tracking every transaction that reads the value of a particular version, we now need to only track when every transaction that precedes T has finished executing. This leads to the following, more efficient, general condition for garbage collecting old versions:

Condition 2: Whenever a transaction updates the value of a particular record, we can garbage collect the preceding version of the record when all transactions with lower timestamps have finished executing.

In order to implement garbage collection based on **Condition 2**, the system needs to maintain a global *low-watermark* timestamp. The low-watermark corresponds to the timestamp of transaction, T', such that all transactions prior to T' have finished executing. Maintaining the low-watermark is less expensive than maintaining the meta-data required for **Condition 1**. However, the low-watermark is a shared global variable that is subject to updates by every execution thread — potentially as frequently as once per transaction — which can hinder Bohm's scalability.

As an alternative, note that the transaction execution layer receives transactions in batches. Transactions are naturally ordered across batches; if batch b_0 precedes batch b_1, then *every* transaction in b_0 precedes *every* transaction in b_1. Going back to our example, assume that the transaction T belongs to batch b_0. T updates the value of record r, and produces a new version v_2. r's preceding version is v_1. The timestamp of any transaction in batch b_i, where $i > 0$, will always exceed v_2, T's timestamp. As a consequence, version v_1, which precedes v_2, will *never* be visible to transactions in batches which occur after b_0. Section 4.2.1 explained that a single execution threads always process batches sequentially, that is, each thread will not move onto batch b_{i+1} until the transactions it is assigned in b_i have been executed. Therefore, we can garbage collect v_1 when *every* execution thread has finished executing batch b_0. This condition holds regardless of which batch v_1 was created in. This intuition forms the basis of the condition Bohm actually uses for garbage collecting old versions:

Condition 3: Whenever a transaction in batch b_i updates the value of a particular record, we can garbage collect the preceding version of the record when every execution thread has completed processing batch b_i.

Garbage collection based on **Condition 3** is amenable to an efficient, scalable implementation based on read-copy-update (RCU) [23]. The heart of the technique is maintaining a global low-watermark corresponding to the minimum batch of transactions processed by *every* execution thread. Each execution thread t_i maintains a globally visible variable $batch_i$, which corresponds to the batch most recently executed by t_i. $batch_i$ is only updated by t_i. We designate one of the execution threads, t_0, with the responsibility of periodically updating a global variable $lowwatermark$ with $min(batch_i)$, for each i.

5. EXPERIMENTAL EVALUATION

Almost every evaluation of new concurrency control protocols published in the last two decades compare the proposed technique with traditional, single-versioned, two-phase locking (2PL). Therefore, we follow this tradition and do the same in this paper. Since so many comparisons with two-phase locking exist in the literature, much can be learned by comparing the relative performance of our multiversioned system vs 2PL with other published comparisons of multiversioned systems vs 2PL. We thus indirectly compare with other published protocols using this method — we believe that this leads to a more "apples-to-apples" comparison, since complete systems like Hekaton and Shore-MT have many novel features beyond concurrency control, and comparing our prototype with these complete systems is not meaningful.

Our experimental evaluation is conducted on a single 40-core machine, consisting of four 10-core Intel E7-8850 processors and 128GB of memory. Our operating system is Linux 3.9.2. All experiments are performed in main-memory, so secondary storage is not utilized for our experiments. We run with garbage collection turned on in Bohm (see Section 4.2.2), so as to not leak memory. In all configurations that we experiment with, we explicitly pin database threads to CPU cores; there is a 1:1 correspondence between threads and cores.

5.1 Microbenchmark

We begin our experimental evaluation by exploring the effect of concurrency control on throughput on a simple microbenchmark. We create a database consisting of a single table of 1,000,000 records, each of size 8 bytes. The workload consists of a single type of transaction that performs a read-modify-write (RMW) of 10 records.

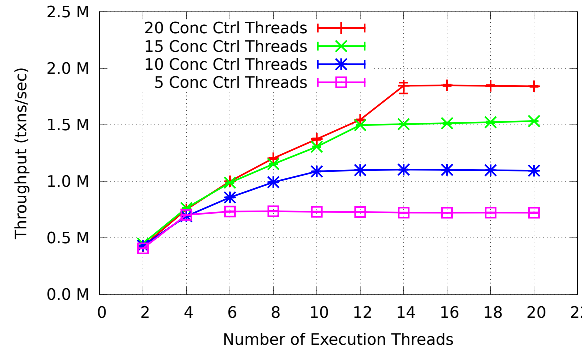

Figure 4: Microbenchmark experiment

Each of the 10 records is unique, and chosen from a uniform distribution.

As described in Section 4, Bohm separates concurrency control from transaction execution. Concurrency control and transaction execution are each handled by two separate modules, each of which is parallelized by a separate group of threads. Both the number of threads devoted to concurrency control and the number of threads devoted to transaction execution are system parameters, and can be varied by the system administrator. We vary both parameters and observe the resulting performance trends.

Figure 4 shows the results of our experiment. The number of threads devoted to transaction execution is varied on the x-axis, while the number of threads devoted to concurrency control is varied via the 4 separate lines on the graph.

Each configuration of the system exhibits the following trend: throughput initially increases as we increase the number of execution threads. After the initial increase, each configuration's throughput eventually plateaus. Before explaining why this is the case, it is important to remember three characteristics of this particular microbenchmark:

- The workload consists of very short transactions; we treat each 8-byte value as a 64-bit integer, and each transaction's read-modify-write operation increments the appropriate value. As a consequence, the execution time of each transaction's logic is very small.
- Each transaction's read/write set is chosen from a set of 1,000,000 records using a uniform distribution; transactions therefore rarely conflict with each other.
- The entire database resides in main memory, so there are no delays to access secondary storage.

As a result of these three characteristics, there are no delays around contending for data, waiting for storage, or executing transaction logic. This stresses the concurrency control layer as much as possible — it is not able to hide behind other bottlenecks and delays in the system, and must keep up with the rest of the database system which consumes very little effort processing each transaction.

Recall that in our experimental setup, there is a 1:1 correspondence between threads and CPU cores. So adding more threads to either concurrency control or transaction execution is equivalent to adding more cores dedicated to these functions.

Now, we return to our analysis of Figure 4. Despite the extreme stress on the concurrency control layer in this microbenchmark, when the number of concurrency control threads (cores) significantly outnumber the number of execution threads (cores), the system is bottlenecked by transaction execution, not concurrency control. This is why the throughput of each configuration initially increases as more execution threads are added. However, once the throughput of the execution layer matches that of the concurrency control layer, the total throughput plateaus; the throughput of the system is bounded by the throughput of the concurrency control layer.

As we increase the number of concurrency control threads (represented by the four separate lines in Figure 4), the maximum throughput of the system increases. This is because, due to the scalability of Bohm's concurrency control layer, the throughput of the concurrency control layer increases as we increase the number of concurrency control threads (cores). As explained in Section 4.1.3, increasing the number of concurrency control threads results in greater intra-transaction parallelism in the concurrency control layer and also results in a reduction of the cache footprint of each concurrency control thread.

Overall, this initial microbenchmark provides evidence of the scalability of Bohm's design. As we increase the number of concurrency control and execution threads in unison, the overall throughput scales linearly. At its peak in this experiment, Bohm's concurrency control layer is able to handle nearly 2 million transactions a second (which is nearly 20 million RMW operations per second) — a number that (to the best of our knowledge) surpasses any known real-world transactional workload that exists today.

5.2 YCSB

We now compare the throughput of our multiversioned system against a single versioned locking system on the Yahoo! Cloud Serving Benchmark (YCSB) [11].

For this set of experiments, we use a single table consisting of 1,000,000 records, each of size 1000 bytes (the standard record size in YCSB). Our experiments use two kinds of transactions: the first performs 10 read-modify-writes (RMWs) — just like the microbenchmark experiment above, while the second performs 2 RMWs and 8 reads (which we call 2RMW-8R).

We use the 10RMW workload to compare the overhead of multiversioning in Bohm compared to a single versioned system. If a workload consists of transactions that perform only RMW operations, we do not expect any benefits from multiversioning. Consider two transactions T_1 and T_2 whose read/write sets conflict on a single record x. Since both transactions perform an RMW on x their execution must be serialized. Either T_1 will observe T_2's write or vice-versa. This serialization is equivalent to how a single version would handle such a conflict.

On the other hand, we expect that Bohm will execute 2RMW-8R transactions with greater concurrency than the single-versioned locking system. The reason is that if a transaction T only reads the value of record r, then T does not need to block a transaction T' which writes r (or alternatively, performs an RMW operation on r).

5.2.1 Low Contention

Our first experiment compares Bohm against the locking system using the two transactions mentioned above when the records in each transaction's read/write sets are chosen from a uniform distribution. For the locking system, concurrency control and transaction execution are performed together, and increasing the number of threads results in more transactions that are run in parallel. For Bohm, different threads perform concurrency control and transaction execution. To avoid adding an extra dimension to our analysis and varying the number of both types of threads, we fix the number of concurrency control threads in the multiversion system to 14 threads, and only vary the number of transaction execution threads. However, this puts Bohm at a major disadvantage when there are small numbers of threads. For example, when there are 16 threads

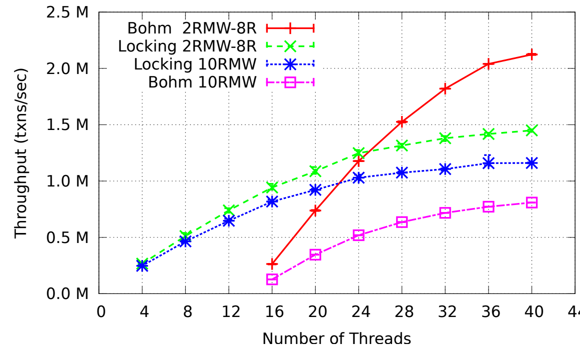

Figure 5: YCSB throughput under low contention

available to the system, the locking system will utilize all 16 to process transactions, while Bohm will only use two of them to process transactions (since the other 14 are reserved for concurrency control — far more than is necessary to keep up with 2 execution threads).

Figure 5 shows the results of the experiment. In both the 10RMW and 2RMW-8R workload, the throughput of the locking system increases as we increase the number of locking threads. We notice, however, that at high thread counts, the slope of both locking lines begins to gradually decline. This is due to the fundamental lack of scalability of having a centralized locking data structure. This problem is exacerbated by the the Non-Uniform Memory Access (NUMA) architecture of the machine we use to perform our evaluation. In particular, as we increase the number of cores used by the system, threads may have to read and write memory that is connected to a remote NUMA node. There are two consequences of these accesses; first, the latency of performing a read or write operation increases, second, the interconnect over which cores access memory located on a remote NUMA node gets congested.

Figure 5 also shows that the throughput of the locking system is higher when it executes the 2RMW-8R workload. This is because transactions need to hold fewer exclusive locks than in the 10RMW workload, and two reads of the same record do not conflict. As a consequence, the 2RMW-8R workload has more concurrency than the 10RMW workload.

Bohm's throughput in the 10RMW workload is lower than that of the locking system. As mentioned earlier, this is because there is no benefit to multiversioning when the workload consists of transactions that only perform RMWs. On the contrary, Bohm has to pay a higher cost per transaction than the locking system. When the locking system performs an RMW operation on record x, it brings the record into cache in order to do the read, and the same cached value is then updated. In contrast, when Bohm performs an RMW operation on x, the corresponding execution thread needs to bring the memory words corresponding x into cache, and write a *different* set of words corresponding to the value of the new version.

Bohm's throughput is *significantly* higher than that of the locking system in the 2RMW-8R workload when there are enough execution threads to balance the 14 concurrency control threads. The reason for this is that Bohm is able to extract much more concurrency from this workload as compared to the locking system. Bohm ensures that transactions that read a particular record do not block other transactions that write the value of the record. In contrast, the locking system requires a transaction that reads the value of a record to take a shared mode lock on the record, which has the effect of blocking transactions trying to write the record.

5.2.2 High Contention

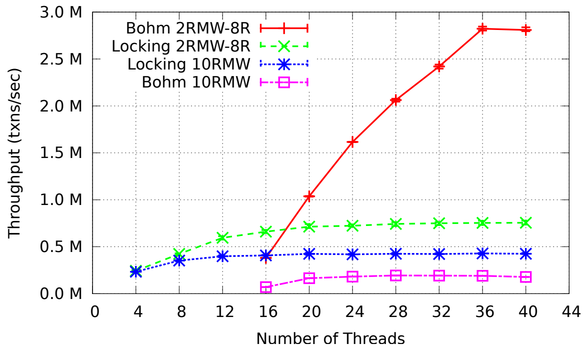

Figure 6: YCSB throughput under high contention (theta = 0.9)

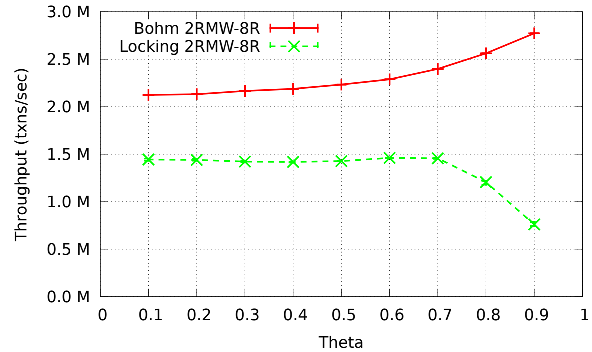

Figure 7: YCSB 2RMW-8R throughput varying contention

This section compares the throughput of the multiversioned system with that of single-versioned locking system under high contention. We use the same set of transactions as before, but instead of using a uniform distribution to select the records in each transaction's read/write sets, we use a zipfian distribution. The zipfian distribution is parameterized by a variable, *theta*, whose value is correlated with the degree of contention in the workload [16]. Theta can take values in the range $[0, 1)$; smaller values correspond to lower contention, as the value of theta increases, the contention in the workload increases.

The read/write set of both transactions consists of 10 unique records, each of which is selected according to the zipfian distribution. For 2RMW-8R transactions, we first select 2 records for RMWs, and then select 8 records to be read.

Figure 6 shows the throughput of Bohm and the locking system while varying the number of database threads when theta is set to 0.9. Unlike the low contention experiment, we see that throughput of the locking system does not scale beyond a certain threshold. This is expected because of the high contention in the workload — there are simply not enough transactions that don't conflict that can be run in parallel. The throughput of the locking system running the 2RMW-8R workload is higher than when it runs the 10RMW workload because transactions in the 2RMW-8R workload have fewer conflicts than transactions in the 10RMW workload.

When Bohm runs 10RMW transactions, its performance trend is similar to that of the locking system since there is no difference between locking and multi-versioning in terms of how transactions conflict when every operation is a write. Just like the low contention experiment, the peak throughput of the multiversioned system is lower than that of the locking system; as mentioned previously (Section 5.2.1), this is because the multiversioned system needs to perform more work per transaction than the single-versioned locking system.

In contrast, in the 2RMW-8R workload, Bohm's throughput performs much better than the alternatives. This is because reads do not block writes for multi-versioning, so there is far more possible concurrency. Surprisingly, Bohm's throughput *exceeds* its throughput in the low contention case (Figure 5). The reason is that, in the 2RMW-8R workload, as soon as there are enough transaction execution threads, Bohm is bottlenecked by the throughput of the concurrency control layer. Increasing the contention in the workload increases the cache locality in the concurrency control layer. Section 4.1 explained that concurrency control threads do not contend on any shared data; the database's records are partitioned among the concurrency control threads, and new versions of a single record are *always* written out by the same concurrency control thread. As the popularity of a few records dramatically increases, the corresponding concurrency control thread(s) will very frequently create new versions for these records. A reference to the latest version of a hot record (and often the version itself) will therefore frequently be cache resident. This improved cache locality results in improved performance.

On a similar workload under high contention, Larson et al. found that Hekaton's throughput was more or less equivalent to that of a single-version locking system [21].

Figure 7 shows this phenomenon in more detail for the 2RMW-8R workload. For this graph, the amount of contention is varied by varying theta on the x-axis, and the number of threads is fixed at 40 (14 concurrency and 26 execution threads for Bohm). Not only is Bohm's baseline throughput higher than locking due to increased concurrency, but their trends are the *opposite* of each other. While Bohm yields improved throughput with higher contention due to improved cache locality, locking yields reduced throughput due to a smaller number of transactions that are capable of being executed in parallel.

5.2.3 Comparison to Hekaton

One of the reasons we chose to use the 2RMW-8R workload is that Larson et al. use a similar workload to compare Hekaton's multi-version system against a single version locking system [21]. Their workload consisted of transactions which performed 10 reads and 2 writes. Our 2RMW-8R workload performs the same number of reads and writes. While a raw throughput comparison would be unfair, we compare the *trend* of Hekaton and Bohm's throughput against the "common baseline" single-version locking system.

In a low contention setting, Hekaton was *outperformed* by single-version locking by about a factor of two. The disparity in performance between Hekaton and single-version locking was attributed to the overhead of maintaining and garbage collecting versions. The fact that Bohm outperforms single-version locking by about 50%, therefore, highlights the effectiveness of our scalable concurrency control and garbage collection techniques.

In a high contention setting, Hekaton's throughput matched that of the single-version locking system. In comparison, in the high-contention 2RMW-8R workload, Bohm outperforms single-version locking by almost 4x. We believe that the difference between Hekaton and Bohm is largely due to Bohm's scalable concurrency control and avoiding writes to shared memory on reads. Hekaton requires readers to atomically increment a counter associated the version being read. At high levels of contention, this can lead to severe scalability issues due to cache-line bouncing.

5.3 SmallBank Benchmark

Our final set of experiments evaluate Bohm's performance on the SmallBank benchmark [9], since this benchmark was used by Cahill et. al. for their research on serializable multi-versioned con-

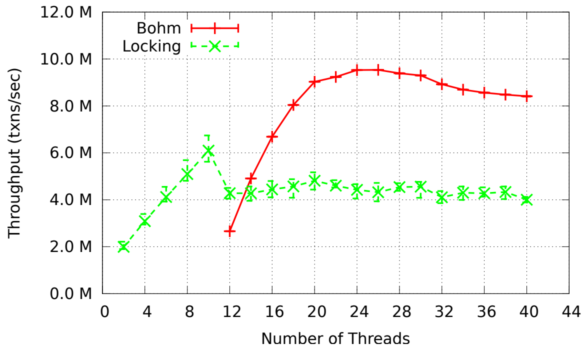

Figure 8: Small Bank throughput under high contention

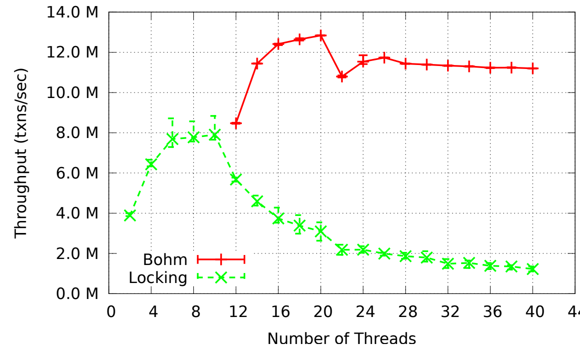

Figure 9: Small Bank read-only throughput

currency control. SmallBank is designed to simulate a banking application. The application consists of three tables, (1) Customer, a table which maps a customer's name to a customer identifier, (2) Savings, a table whose rows contain tuples of the form <Customer Identifier, Balance>, (3) Checking, a table whose rows contain tuples of the form <Customer Identifier, Balance>. The application consists of five transactions: (1) $Balance$, a read-only transaction which reads a single customer's checking and savings balances, (2) $Deposit$, makes a deposit into a customer's checking account, (3) $TransactSaving$, makes a deposit or withdrawal on a customer's savings account, (4) $Amalgamate$, moves all funds from one customer to another, (5) $WriteCheck$, which writes a check against an account. None of the transactions update the customer table — only the Savings and Checking tables are updated.

The number of rows in the $Savings$ and $Checking$ tables is equal to the number of customers in the SmallBank database. We can therefore vary the degree of contention in our experiments by changing the number of customers; decreases the number of customers increases the degree of contention in the SmallBank workload.

The transactions in the SmallBank workload are much smaller than the transactions in the YCSB workload from the previous section. Every transaction performs reads and writes on between 1 and 3 rows. In comparison, our configuration of the YCSB workload performs exactly 10 operations on each transaction. As a consequence, SmallBank requires fewer concurrency control threads than our configuration of YCSB. We dedicate 10 threads to the concurrency control layer, the remaining threads are used by the transaction execution layer. As before, both Bohm and the locking system pin each thread to a specific CPU core, hence, the number of threads in the system is precisely equal to the number of cores utilized.

5.3.1 Full transaction mix

We first compare Bohm's throughput against that of the locking system when running the full set of SmallBank transactions. Each transaction is chosen randomly with equal probability (20%).

Figure 8 shows the throughput of both systems while varying the number of threads used by the database. We configure the database to run under high contention by setting the number of customers to 100.

The throughput of the locking system increases as we increase the number of threads from 2 to 10, and then throughput experiences a sudden drop from 10 to 12 threads. After this, it remains constant until 40 threads. The locking system's throughput drops between 10 and 12 threads because of NUMA effects. Each NUMA socket consists of 10 CPU cores. When the number of threads increases beyond 10, the system uses more than one NUMA socket. The throughput of the locking system thereafter remains flat because of high contention.

Bohm's throughput has a similar trend to that of the locking system; throughput increases until it reaches the maximum achievable throughput, after which it stops increasing because of high contention. However, Bohm's absolute throughput is significantly higher than that of the locking system. At 40 threads, Bohm's throughput is nearly *double* that of the locking system. Bohm is able to better exploit the fact that SmallBank contains a significant fraction of read-only $Balance$ transactions (20%). This effect is amplified due to the high contention in this experiment. Bohm ensures the read-only $Balance$ transactions never block other concurrent transactions with conflicting read/write sets.

5.3.2 Read-only mix

We now compare the throughput of Bohm and the locking system on a transaction "mix" consisting of just the $Balance$ transaction. $Balance$ is a read-only transaction; its logic reads a single customer's savings and checking balance.

Figure 9 plots throughput as a function of the number of database threads. The database is configured with 100 customers.

The throughput of the locking system increases from 2 to 10 cores. The throughput then decreases as we add more threads. This decrease is quite surprising — for a read-only workload, transactions should not block each other (even in the locking system), and there is no limit to the amount of theoretically possible parallelism. However, the throughput decreases for the locking system because of contention in the lock manager. $Balance$ transactions are very short — each transaction effectively reads two integers and computes their sum. As a consequence, the rate of lock acquisitions is very high, even in comparison to the full SmallBank transaction mix. Even though the lock acquisitions are compatible with each other, this workload exhibits contention for the *latches* protecting a single hash-bucket in the lock manager. As the number of threads contending for the same latch increases, throughput decreases due to cache line "bouncing" [4, 24].

Bohm's throughput also initially increases, drops between 20 and 22 threads, and then remains constant. The drop in throughput is due to NUMA effects (recall the each NUMA socket consists of 10 cores). Bohm's throughput is then bottlenecked by the concurrency control phase. Once the throughput of the execution phase equals that of the concurrency control phase, throughput remains constant (had we not fixed the number of concurrency control threads, throughput would have increased further). Unlike the locking system, Bohm's throughput does not collapse. This experiment validates our design decision of ensuring that reads do not result in writes to shared memory.

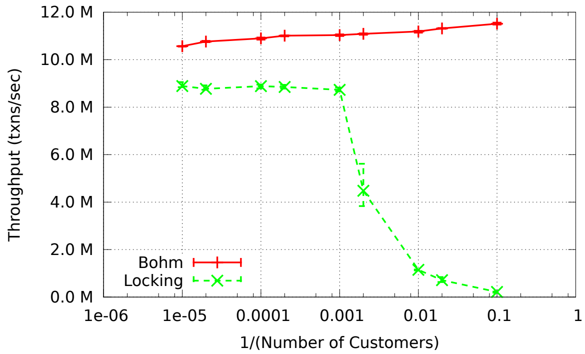

Figure 10: Small Bank read-only throughput varying contention

Figure 10 shows how read-only throughput varies with contention. At the highest level of contention, the throughput of the locking system is *lower* than the throughput of the full transaction mix from Section 5.3.1. Meanwhile, Bohm's throughput increases due to the same cache effects described in Section 5.2.2.

6. CONCLUSIONS

Most multi-versioned database systems either do not guarantee serializability or only do so at the expense of significant reductions in read-write concurrency. In contrast Bohm is able to achieve serializable concurrency control while still leveraging the multiple versions to ensure that reads do not block writes. Our experiments have shown that this enables Bohm to significantly outperform single-versioned locking systems on most workloads (without giving up serializability). Bohm is the thus the first multi-versioned database system to accomplish this in main-memory multi-core environments.

7. REFERENCES

[1] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 67–78. IEEE, 2000.

[2] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1):45–59, 1987.

[3] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[4] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6–16, 1990.

[5] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *ACM SIGPLAN Notices*, volume 46, pages 487–498. ACM, 2011.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proc. of SIGMOD*, SIGMOD '95, pages 1–10, 1995.

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

[8] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, Jr. Concurrency control in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):18–51, 1980.

[9] M. J. Cahill. *Serializable Isolation for Snapshot Databases*. PhD thesis, University of Sydney, 2009.

[10] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proc. of SIGMOD*, SIGMOD '08, pages 729–738, 2008.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. SoCC '10, pages 143–154.

[12] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. SIGMOD '14, pages 15–26, 2014.

[13] A. Fekete. Allocating isolation levels to transactions. In *Proc. of PODS*, PODS '05, pages 206–215, 2005.

[14] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[15] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record*, 33(3):12–14, 2004.

[16] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. SIGMOD '94, pages 243–252, 1994.

[17] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *Proc. of SIGMOD*, SIGMOD '08, pages 981–992, 2008.

[18] K. Jacobs, R. Bamford, G. Doherty, K. Haas, M. Holt, F. Putzolu, and B. Quigley. Concurrency control, transaction isolation, and serializability in SQL 92 and Oracle 7. Oracle Whitepaper, Part No. A33745, 1995.

[19] R. Johnson, I. Pandis, and A. Ailamaki. Improving oltp scalability using speculative lock inheritance. *Proceedings of the VLDB Endowment*, 2(1):479–489, 2009.

[20] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *Proc. of SIGMOD*, SIGMOD '13, pages 73–84, 2013.

[21] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[22] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *Proc. of ICDE*, pages 604–615. IEEE, 2014.

[23] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *In Ottawa Linux Symposium*, pages 338–367, 2001.

[24] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[25] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of SOSP*, pages 358–372. ACM, 2013.

[26] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley db. In *USENIX, FREENIX Track*, pages 183–191, 1999.

[27] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proceedings of the VLDB Endowment*, 3(1-2):928–939, 2010.

[28] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23.

[29] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. PVLDB, 2013.

[30] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *PVLDB*, 7(10):821–832, 2014.

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

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

[33] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Fast distributed transactions and strongly consistent replication for oltp database systems. *ACM Trans. Database Syst.*, 39(2):11:1–11:39, May 2014.

[34] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. SOSP, 2013.