Fast Transactions for Multicore In-Memory Databases

by

Stephen Lyle Tu

B.A., University of California, Berkeley (2010),
B.S., University of California, Berkeley (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2013

Certified by .....
Samuel R. Madden
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Fast Transactions for Multicore In-Memory Databases

by

Stephen Lyle Tu

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Though modern multicore machines have sufficient RAM and processors to manage very large in-memory databases, it is not clear what the best strategy for dividing work among cores is. Should each core handle a data partition, avoiding the overhead of concurrency control for most transactions (at the cost of increasing it for cross-partition transactions)? Or should cores access a shared data structure instead? We investigate this question in the context of a fast in-memory database. We describe a new transactionally consistent database storage engine called Maflingo. Its cache-centered data structure design provides excellent base key-value store performance, to which we add a new, cache-friendly serializable protocol and support for running large, read-only transactions on a recent snapshot. On a key-value workload, the resulting system introduces negligible performance overhead as compared to a version of our system with transactional support stripped out, while achieving linear scalability versus the number of cores. It also exhibits linear scalability on TPC-C, a popular transactional benchmark. In addition, we show that a partitioning-based approach ceases to be beneficial if the database cannot be partitioned such that only a small fraction of transactions access multiple partitions, making our shared-everything approach more relevant. Finally, based on a survey of results from the literature, we argue that our implementation substantially outperforms previous main-memory databases on TPC-C benchmarks.

Thesis Supervisor: Samuel R. Madden
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Eddie Kohler and Barbara Liskov's ideas and mentoring were fundamental to the inspiration and early designs which eventually culminated in this thesis. Wenting Zheng's preliminary results on persistence were also critical for shaping the design principles of this thesis. Eugene Wu provided helpful feedback on an early draft of this work. Frans Kaashoek and Nickolai Zeldovich, who were not directly involved in this research, were nevertheless very crucial in helping me become a better researcher. I would also like to thank my family and friends for providing me with support and guidance. Finally, my advisor Sam has been a great mentor and a pleasure to work with.

Contents

List of Figures

Chapter 1

Introduction

There have been two notable trends in computer architecture over the past several years: increasing main memory sizes and dramatic increases in processor core count. A modern high-end server can now have several terabytes of RAM and hundreds of cores. These trends have dramatically altered the landscape of data management technology: data sets and computation that used to be spread across tens of disks and machines can now reside in the memory of a single multicore computer.

Processing this data efficiently requires fundamentally new designs for data management systems. First, many database workloads, especially those involving *online transaction processing* (OLTP) can now fit entirely in main memory. These are the types of databases that run most websites, banks, and other organizations, containing a small set of records for each user or customer. Typical workloads involve many concurrent reads of records and only a few concurrent writes at a time as users buy products, transfer funds, send emails, or perform other operations.

Second, it would be desirable to get as much parallelism as possible from the proliferation of available cores, especially on OLTP workloads where high throughput operations are required. Unfortunately, conventional database designs do not scale well with the number of cores for a variety of reasons [16, 14], including reliance on centralized data structures (shared buffer pool, shared lock manager, etc.) that must

be latched [1] by every operation, and use of memory layouts and index designs that are optimized for disk-resident data rather than main memory.

To address these issues, several new database designs have been proposed for main-memory OLTP workloads. The most common design involves some form of *data partitioning*, where each core is responsible for a different subset of the data [33, 26, 31]. Such approaches work well when data partitions cleanly, because each core essentially operates on a separate logical database, resulting in good cache locality and low levels of contention for memory between cores. However, as core counts grow, more and more data partitions are needed when using this approach. As partitions become finer-grained, it becomes increasingly hard to partition the database to ensure that each transaction accesses only one partition. The resulting multi-partition transactions require latching entire partitions for access to the records of other cores, and have adverse effects on cache locality. Furthermore, many databases schemas are not trivially partitionable (such as social networks), and state-of-the-art partitioning techniques are very workload dependent [11].

In this thesis, we present Maflingo, a new main-memory serializable database system designed to scale to many cores *without* the use of partitioning. Maflingo is focused on OLTP applications that fit into main memory, where the goal is to execute many concurrent, short transactions as fast as possible. In particular, it is not designed to handle long running read-write transactions (which could lead to high abort rates in our system), nor is it designed to handle workloads which are too large to fit in main memory.

Maflingo runs transactions using a variant of *optimistic concurrency control* (OCC) [19]. In OCC, a transaction accesses records without checking for conflicts with other concurrent transactions, tracking read and write sets as it runs. Conflict checks are made after the transaction has finished running. In particular, OCC guarantees serializability by aborting a transaction if its read set conflicts with the write sets of other concurrent transactions.

[1] The database literature reserves the term "lock" for high-level transaction locks, and uses "latch" to refer to low-level data structure locks.

The key advantage of using OCC in a multicore setting, as opposed to more traditional techniques such as two-phase locking (2PL), is that tracking read and write-sets does not require any locking. This is especially relevant for multicore architectures, since the act of obtaining a lock, *even a read lock*, involves transferring cache-lines via the hardware coherency protocol, which is inherently a non-scalable operation [4]. Additionally, OCC has the benefit of minimizing the time when a transaction holds locks to just the commit phase after the transaction has completely executed. MAFLINGO makes several contributions, which we outline below.

Multicore friendly OCC. Our implementation does not involve any global critical sections (which reduce concurrency), and avoids cache contention by not performing any *un-necessary* writes to memory shared between cores (such as database records). Our approach employs a form of multi-versioning on records to support efficient garbage collection of updated and deleted records.

Recent read-only transactions. Multi-versioning allows us to support read-only transactions that see a slightly stale but consistent version of the database. These transactions can be run without tracking read sets and will *never* abort. Read-only transactions can also be used for efficient check-pointing for recovery without interfering with concurrent writes.

Fast performance. We evaluate MAFLINGO on two standard transactional benchmarks, TPC-C and YCSB, showing that (i) it provides absolute transaction throughput that is substantially higher than several recent main memory databases, (ii) its OCC implementation has almost zero overhead on key-value workloads like YCSB, (iii) when compared to a partitioned system that runs one transaction at a time on each partition, MAFLINGO performs better when as few as 15% multi-partition transactions are involved, and finally (iv) our optimizations for read-only transactions allow the system to maintain good performance even with large read-only transactions which read frequently updated records running.

Chapter 2

Related work

A number of recent systems have proposed storage abstractions for main-memory and multicore systems. These can be broadly classified according to whether or not they provide transactional support.

Many of these systems, including Maflingo, draw on a long line of classic and current distributed and database systems work that shows that creating copies of objects on writes (generally known as multi-version concurrency control, or MVCC), can decrease contention for shared objects and allow read-only transactions to proceed without being blocked by writers [19, 10, 3].

In the classic database literature, there are a few systems that combine OCC and multi-versioning, including the multiversion serial validation protocol of Carey et al. [6] and the multiversion parallel validation extensions of Agarwal et al. [1] Both of these protocols include large critical sections (like the original OCC paper) that likely make them impractical for modern multicores.

2.1 Non-transactional Systems

The non-transactional system most related to Maflingo is Masstree [24], which shows that it is possible to build an extremely scalable and high throughput durable main memory B^+-tree using techniques such as *version numbers* instead of read locks, as well as efficient fine-grained locking algorithms that allow many concurrent updates

to different parts of the tree. MAFLINGO's underlying concurrent B^+-tree implementation is inspired by the design of Masstree. MAFLINGO's main contribution is showing how to take an existing tree such as Masstree, and make it fully serializable.

Other recent non-transactional systems include BW-Tree [23], a high throughput MVCC-based tree structure optimized for multicore and flash storage. The authors suggest it could be used in a transactional setting with conventional locking, but do not tightly couple it with a transaction processor. This work is complementary to ours; MAFLINGO could have just as easily used BW-Tree for its concurrent index. PALM [32] is a similar high-throughput tree structure designed for multicore systems. It uses a bulk-synchronous processing (BSP)-based batching technique and intra-core parallelism to provide extremely high throughput, but again does not show how to integrate this into a high throughput transaction processing system.

These recent tree structures build on many previous efforts designed to minimize locking bottlenecks in tree-based structures, including OLFIT [7], Bronson et al. [5], and the classic work of Lehman and Yao on B^{link}-trees [21].

In MAFLINGO, our goal is not to innovate on concurrent tree structures, but to show that by coupling a high performance index structure (similar to the design of Masstree) with an efficient version-based concurrency control scheme, it is possible to process transactions with little additional overhead or significant limitations to scalability.

2.2 Transactional Systems

Recently, Larson et al. [20] revisited the performance of locking and OCC-based MVCC systems versus a traditional single-copy locking system in the context of multicore main-memory databases. Their OCC implementation exploits MVCC to avoid installing writes until commit time, and avoids many of the centralized critical sections present in classic OCC. However, their design lacks many of the multicore-specific optimizations of MAFLINGO. For example, it (i) has a global critical section when assigning timestamps, (ii) requires transaction reads to perform non-local memory

writes to update dependency lists of other transactions, (iii) possibly forces read-only transactions to abort, and (iv) does not include an efficient garbage collector. These limitations cause it to perform about 50% worse in simple key-value workloads than the single-copy locking system even under low levels of contention, whereas MAFLINGO's OCC-based implementation is within a few percent of a key-value system for small key-value workloads (see §6.3).

Several recent transactional systems for multicores have proposed partitioning as the primary mechanism to achieve scalability. Dora [26] is a locking-based system that partitions data and locks among cores, eliminating long chains of lock waits on a centralized lock manager, and increasing cache affinity. Though this does improve scalability, overall the performance gains are modest (about 20% in most cases) versus a traditional locking system. Additionally, in some cases, this partitioning can cause the system to perform worse than a conventional system when operations touch many partitions.

PLP [27] is follow up work on Dora. In PLP, the database is physically partitioned among many trees such that only a single thread manages a tree. The partitioning scheme is flexible, and thus requires maintaining a centralized routing table. Running a transaction, like in DORA, requires decomposing it into a graph of actions that run against a single partition; such a design necessitates the use of rendezvous points, which are additional sources of contention. Finally, the authors only demonstrate a modest improvement over a conventional 2PL implementation.

H-Store [33] (and its commercial successor VoltDB) employ an extreme form of partitioning, treating each partition as a separate logical database even when partitions are co-located on the same physical node. Transactions local to a single partition run without locking at all, and multi-partition transactions are executed via the use of whole-partition locks. This makes single-partition transactions extremely fast, but, even more so than Dora, creates scalability problems when running multi-partition transactions. We compare MAFLINGO to a partitioned approach in §6.5, confirming the intuition that partitioning schemes can be effective with few multi-partition transactions but do not scale well in the presence of many such transactions.

In MAFLINGO, we eliminate the bottleneck of a centralized lock manager by co-locating locks with each record. VLL [30] also adopts this approach, but is not focused on optimizing for multicore performance. Shore-MT [16] shows how to take a traditional disk-based RDBMS and scale on multicore by removing centralized bottlenecks due to latching. Jung et al. [18] show how to improve the scalability of the lock table in RDBMS systems such as MySQL and Shore-MT by using a latch free data structure and careful parallel programming techniques. All these systems are, however, primarily 2PL systems; as discussed previously, this introduces scalability concerns on multicore architectures.

Porobic et al. [29] perform a detailed performance analysis of shared-nothing versus shared-everything OLTP on a single multicore machine, and conclude that shared-nothing configurations are preferable due to the effects of non-uniform memory accesses (NUMA). Our design does not preclude the use of NUMA-aware techniques; indeed our system takes into consideration the topology of the machine when allocating memory, and carefully sets CPU affinity policies for worker threads.

Chapter 3

Architecture

MAFLINGO is a main-memory transactional database storage manager. It exposes a relational data model, consisting of tables of typed, named records. It provides a simple API that allows clients to issue requests in the form of *stored procedures*. Stored procedures are run as a single transaction, and consist of a sequence of reads and writes to the database, interspersed with arbitrary application logic. The use of stored procedures avoids network round trips and the potential for stalls when users or applications submit individual transaction statements interactively.

Under the covers, each table in MAFLINGO is implemented as a collection of index trees, including a *primary tree* sorted on the table key that contains the records themselves, as well as one or more *secondary trees*, sorted on other table attributes, with references to the primary key values in the leaves of the tree. Reads in MAFLINGO may request a single key value, or a range of values from any index by specifying a *predicate*. Writes are to a single key value. Currently MAFLINGO does not provide a SQL interface; stored procedures are handed-coded as reads and writes to primary and secondary indexes. Supporting a SQL query interface would be straightforward. Our index structure is described in §4.1. The basic design of MAFLINGO is shown in Figure 3-1.

For stored procedures that are read-only, clients may specify that they be run in the past. Such calls are run on a transactionally consistent but slightly (e.g., one second) stale version of the database. Because these read-only transactions never

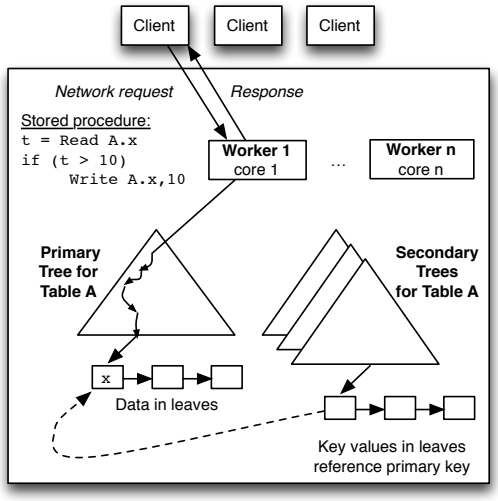

Figure 3-1: The architecture of MAFLINGO.

access data that is being concurrently modified by update transactions, they can be executed more efficiently, and are guaranteed to not be aborted due to conflicts. Read only transactions are described in §4.7, and evaluated in §6.6.

Although the primary copy of data in MAFLINGO is in main memory, transactions are made durable via logging to disk. Transactions do not return results to users until they have been persisted. Persistence is ongoing work and not described in this thesis.

Clients interact with MAFLINGO by issuing requests to execute stored procedures over the network. When a stored procedure is received, it is dispatched to a database worker thread. We generally run one worker thread per physical core of the server machine, and have designed MAFLINGO to scale well on modern multicore machines with tens of cores. Data in MAFLINGO is not partitioned: each worker can access the *entire* database. In non-uniform memory access (NUMA) systems, MAFLINGO is aware of the topology of the machine and provides mechanisms to allow users to configure affinity between worker cores and data. A core runs a single stored procedure at a time, and once a worker accepts a stored procedure, it runs until it either commits or aborts.

As noted previously, MAFLINGO employs a novel commit protocol—a variant of optimistic concurrency control—that provides transactional consistency without impairing performance or scalability. We describe the details of this protocol next.

Chapter 4

Design

This section describes how we execute transactions in Maflingo in a multicore-friendly way. Key aspects of our design are listed below:

Lightweight latching only. We eliminate all transaction locks and rely only on lightweight *latches* for concurrency control. Latches are associated with individual records. Since latches are in line with the data, false conflicts on latches are likely to be rare. Contrast this, for example, with the single critical section used by traditional OCC implementations [19].

No shared-memory writes on reads. We take care to eliminate *all shared-memory writes* for records read (but not written) by a transaction. The version number checking necessary for OCC completes without ever latching the relevant records. Note that Maflingo's index data structure also behaves in a similar manner.

No global critical sections. There are no global critical sections in our protocol. This includes the assignment of transaction identifiers (TIDs), which we perform locally without coordination.

Efficient validation of range scans. The concurrency control primitives (version numbers) of the underlying index tree are used to efficiently check for absent ranges of records (to avoid the *phantom problem*). This is in contract to [20] which re-runs scans at the end of transactions for validation purposes.

4.1 Data Layout

The representation of a database in MAFLINGO is fairly standard for in-memory databases. The database's records are stored in a collection of B^+ trees (our protocol, however, is easily adaptable to other index structures such as hash tables). As noted in §2.1, the design of our B^+ trees was motivated by Masstree [24]; in particular we adopt its trie-like structure and its techniques for safely managing concurrent tree structure modifications. Each table's primary tree is sorted according to its primary key. Each node in the tree contains information about a range of keys; for a key in that range it either points to the record for that key, or to a lower-level node where the search can be continued (this is Masstree's trie structure).

Records. The in-memory representation of a record in MAFLINGO contains the following information:

- **A transaction ID**, whose format is described below.
- **A latch**. The latch prevents concurrent modifications once a transaction starts its commit phase.
- **A *previous version* pointer**. The pointer is null if there is no previous version. The previous version is used to support in the past read-only transactions; we discuss how this is done in §4.7.
- **The record data**. When possible, the actual record data is stored in the same allocation as the record header, avoiding an additional memory read to fetch values.

We make the somewhat unusual choice to modify record data in place during the commit phase. This speeds up performance for short writes, mainly by reducing the memory allocation overhead for record objects, but complicates the process of reading a consistent version of a record's data. In this section we assume that versions are read atomically; §5 describes how we achieve this in our implementation.

Transaction IDs. MASSTREE transaction IDs (TIDs) are 8-byte integers used to identify transactions and to detect conflicts, and are stored with every record. Each database record carries the ID of the transaction that most recently modified that record. We assign transaction IDs in a decentralized fashion. Each transaction ID has three parts. From most to least significant, they are:

- An *epoch number*. The epoch number is used to manage efficient record mutation, database snapshots, and read-only transactions, and is described further in §4.6 and §4.7. It is incremented once per snapshot interval, which is a system parameter.
- A *counter*. This is set as described below, to ensure that a record's TID increases monotonically.
- A *core number*. This is the unique ID of the core that executed the transaction. Including the core number ensures that two cores never pick the same transaction ID.

Every record in the database logically starts with TID zero. A core chooses a transaction's ID at its commit point. It picks the smallest ID that (1) is in the current epoch, (2) is larger than all record TIDs observed during its execution, (3) is larger than its most recent TID, and (4) has its core ID in the low bits. Note that the TIDs chosen by a core are monotonically increasing.

Our system selects TIDs in a way that reflects the dependency of a transaction: if T1 reads a modification installed for T2, then T1's TID will be greater than T2's. However, we do *not* ensure that TIDs produced by different cores reflect the serial order; doing so would require global coordination at commit time. We describe later how serializability is maintained nonetheless.

4.2 Running transactions

We now describe how we run transactions. We first discuss transactions that only read and update existing keys in a point-wise manner. Inserts, removals, and range

queries are discussed in §4.3 and §4.5.

As a core runs a transaction, it maintains a *read-set* that identifies all records that were read, along with the TID of each record at the time it was first accessed. For modified records, it maintains a *write-set* that stores the new state of the record (but *not* the previous TID). Records that are both read and modified occur in both the *read-set* and the *write-set*. A record that was only modified—for example, an inserted record, or a record that was completely overwritten without reference to any previous value—occurs only in the *write-set*, which helps us avoid unnecessary conflicts.

```
   Data: read-set R, write-set W
 1 for tuple, new-contents in sorted(W) do                    /* Phase 1 */
 2 |   status ← trylock(tuple)
 3 |   if status = busy then abort()
 4 |
   /* serialization point for transactions which commit     */
 5 fence()                  /* compiler-only on x86 */
 6 for tuple, read-tid in R do                                /* Phase 2 */
 7 |   if tuple.tid ≠ read-tid or is_locked_by_other(tuple) then
 8 |   |   abort()
 9 |   else
   |   |   /* validation of tuple passed                 */
10 commit-tid ← generate_tid(R, W)            /* see §4.1 */
11 for tuple, new-contents in W do                           /* Phase 3 */
12 |   write(tuple, new-contents, commit-tid)
13 |   unlock(tuple)
```

Figure 4-1: Commit protocol

When the transaction has finished running, the core attempts to commit. The commit protocol works as follows, and is summarized in Figure 4-1:

Phase 1. The core examines all records in the transaction's *write-set*. Each record is first locked by acquiring the record's latch (see §4.1). To avoid deadlocks during this phase, cores latch records in a global order. Any deterministic global order is fine; MAFLINGO uses the pointer addresses of records. If a transaction cannot obtain a latch after some number of spins, then it gives up and aborts.

If a transaction only does reads, a memory fence is necessary after Phase 1 to

ensure serializability (we expand on this below). This fence is not necessarily expensive; for example, on x86 and other total store order (TSO) machines, it can be a compiler fence that does not actually affect compiled instructions (it just prevents the compiler from moving code aggressively). If a transaction contains any writes, the interlocked instructions executed to obtain latches provide the necessary memory ordering properties. However it is obtained, this memory fence is the serialization point for transactions that commit.

Phase 2. The core examines all the records in the transaction's *read-set* (which may contain some records that were both read and written). If some record has a different TID than that observed during execution, or if some record is latched by a different transaction, the transaction releases its latches and aborts.

If the TIDs of all read records are unchanged, then at this point the transaction is allowed to commit, because we know that all its reads are consistent with each other. The core assigns the transaction an ID as described above.

Phase 3. During this phase, the core writes the modified records and updates their TIDs to the transaction ID computed in the previous phase. Latches can be released as soon as the new record has been written. Maflingo must ensure that the new TID becomes visible before the latch is released, which on the x86 requires a compiler-only memory fence between assigning the TID and releasing the latch.

4.2.1 Serializability

This protocol is serializable because (1) it latches all written records before validating the TIDs of read records, (2) it treats latched records as dirty and aborts on encountering them, and (3) the memory fence between Phase 1 and Phase 2 ensures that readers will see writes in progress.

Consider two committed transactions t_1 and t_2 whose *read-* and *write-set*s overlap. For these transactions to be serial-equivalent, we must show that either t_1 observed all of t_2's writes and t_2 observed none of t_1's, or vice versa. Non-serializable write interleavings cannot occur because all latches are acquired before any data is written. It suffices to limit the analysis to the case where t_1 reads at least one record that t_2

Core 1	Core 2
$z_1 :=$ Read y	$z_2 :=$ Read x
$z_1 := z_1 + 1$	$z_2 := z_2 + 1$
Write z_1 into x	Write z_2 into y

Figure 4-2: A read-write conflict

writes.

First, suppose that t_1 observed one of t_2's writes, say a write to record x. This means that t_1's *read-set* contained TID t_2 for x. Values are placed in a *read-set* before the commit process begins, so we know that t_2 was executing its Phase 3 (which is where TIDs get written) before t_1 began its commit process. This shows that t_2 observed none of t_1's writes. Now we show that unless t_1 observed *all* of t_2's writes, it will abort. Assume that t_1 observed an old TID of some other record, y, written by t_2. But we know that t_2 latched y (in Phase 1) before writing x's new TID (in Phase 3), and the memory fence between Phase 1 and Phase 2 ensures that the latch of y is visible to other cores before x's new TID becomes visible to other cores. As a result, t_1's check of y's TID in its Phase 2 will either observe the latch or y's new TID. In either case t_1 will abort.

Now, suppose that t_1 observed none of t_2's writes. We show that if t_2 committed it must have observed all of t_1's writes. If t_1 did not write anything this is trivial. Otherwise, as above, note that t_1 latches all of its written values before checking its *read-set*. Since t_1 observed none of t_2's latches or new TIDs, we know that t_2 hadn't started its commit process when t_1 entered Phase 2. As a result, the memory fence between t_2's Phase 1 and Phase 2 will ensure that t_2 observes the latches set by t_1. □

A short example. Suppose we have a table with two records x and y that both start out with value zero (and TID zero). Consider the race shown in Figure 4-2. It is clear that a final state of $\langle x = 1, y = 1 \rangle$ is not a valid serializable outcome. Note that this is the classic *snapshot isolation anomaly*. Let us illustrate why such a final state is impossible in MAFLINGO. Suppose that this final state occurred, meaning core 1 read $y_{v=0} = 0$ and core 2 read $x_{v=0} = 0$. For this to happen, both cores must have passed Phase 1, so x and y are latched by cores 1 and 2, respectively. Consider the

execution of Phase 2 by core 1 (an identical argument would apply for core 2). Core 1 checks x and sees that x is latched, so it aborts. But then the final state would not contain $x = 1$.

4.3 Range queries

Range queries are complicated by the *phantom problem* [12]. If we scanned a particular range but only kept track of the records that were present during our scan, then new records could be added to the range without being detected by the protocol, violating serializability.

Typical solutions to this problem in databases involve predicate locking [12]. Because predicate locking is often expensive to implement in practice, another commonly deployed technique is next-key locking [25]. All these methods involve some form of locking for reads, and go against MAFLINGO's design philosophy of lock-free reads.

In order to deal with this issue, we take advantage of the underlying B$^+$-tree's version number on each leaf node. The underlying B$^+$-tree guarantees us that structural modifications (remove/insert) to a tree node result in a version number change. A scan on the interval $[a, b)$ therefore works as follows: in addition to recording all records within the interval in the *read-set*, we also maintain an additional set, called the *node-set*, which tracks the versions of B$^+$-tree nodes that we examined. We add the tree nodes that cover the entire key-space from $[a, b)$ to the *node-set* along with the version number examined during the scan. Phase 2 then also checks that the version number of all tree nodes in the *node-set* have not changed. This ensures there were no structural modifications (i.e., inserts or deletes) to any of the ranges examined.

4.4 Secondary indexes

MAFLINGO supports secondary indexes with no changes over what has been described above; we are effectively oblivious to these indexes. To us they appear to be simply

additional tables. These tables map secondary keys to records that store the primary keys of records in the table.

When modifications occur that affect the secondary index, the secondary index must be changed. This happens via extra accesses in the body of the stored procedure that did the modification. These modifications will cause aborts in the usual way: if a transaction uses the secondary index, it will abort if the record it accesses has changed.

4.5 Inserts and deletes

Inserts. Phase 2 as described in §4.2 handles write-write conflicts by requiring transactions to first lock records. However, a non-existing record means there is no shared record for transactions to lock.

We deal with this problem by inserting a new record *before* starting the commit protocol. We add an atomic *insert-if-absent* primitive to the underlying B^+ tree. (Such an operation is straightforward to implement in a Masstree-like tree.) *Insert-if-absent*(k, v) succeeds and inserts a $k \to v$ mapping if k does not already exist in the tree. Otherwise, it fails and returns k's current record without modifying the tree.

An insert operation on key k then works as follows. First, a new empty (absent) record v is constructed, and *insert-if-absent*(k, v) is called. If the call succeeds, v is added to the *write-set* as if a regular put occurred on k. If the call fails, v is deleted, and the existing value v' is added to the *write-set*. The atomic operation is necessary to ensure that at any given time, there cannot be more than one record for a key.

The careful reader will have noticed a slight issue with insert operations (which themselves can trigger structural modifications) and the *node-set* tracking discussed in §4.3. Indeed, we need to distinguish between structural modifications caused by the current transaction (which must not cause aborts) and modifications caused by other conflicting transactions. This is straightforward to fix: we simply have *insert-if-absent* return $(node, v_{\text{old}}, v_{\text{new}})$ if it makes a modification. We then check the *node-set*

for *node*. If *node* is not present in *node-set*, then we do not have to do anything extra. Otherwise, we check its version. If the version was v_{old}, we change it to v_{new}; otherwise we abort.

An important insert optimization is to latch the new record before inserting it into the tree. Though this can violate the normal latch order, it is safe to do so because there is no possible deadlock: the core processing the transaction holds the only reference to the new record. Furthermore, early latching is desirable: it is inexpensive (can be done without using interlocked instructions) and reduces the time to acquire latches during commit.

Deletes. We implement deletes as a two-step process. First, transactions only logically remove records from the tree, by writing an empty size value for a record (a tombstone). Thus, from the point of view of transactions, removal is just a regular write. To actually clean up logically deleted records from the tree, however, we take advantage of our infrastructure to support read-only transactions by deferring the task to a background garbage collection. We discuss these details in §4.8.

4.6 Avoiding reference counting

A transaction that deletes, or otherwise replaces, a record cannot immediately free the record's memory: that record might be in some transaction's read set. Although reference counting could solve this problem, it would require shared memory writes for records that were only read, a large expense we aim to avoid. Maflingo therefore uses a form of read-copy-update [13] to garbage collect dead records. RCU is much more scalable than local reference counting.

In RCU terms, each transaction forms a "read-side critical section." The system cannot free data until all concurrent read-side critical sections have ended, since those read-side critical sections might be examining the data.

The epoch numbers embedded in transaction IDs track read-side critical sections. They allow us to use an epoch-based reclamation scheme [15], an efficient form of

- T_{cur}. The current epoch used by read/write transactions.
- $T_{readonly}$. The epoch where read-only transactions are running. Is either $T_{cur}-1$ or $T_{cur}-2$.
- $CurTable[num\_cores]$ contains an entry for each core with the current read/write epoch being used by that core.
- $ReadonlyTable[num\_cores]$ contains an entry for each core with the current read-only epoch being used by that core.
- T_{gc}. The latest epoch that is no longer needed.

Figure 4-3: State used to support read-only transactions.

RCU reclamation. We explain how we do epoch management in §4.7 and garbage collection in §4.8.

4.7 Read-only transactions

We support running read-only transactions in the past by retaining additional versions for records. These versions are used to form a *read-only snapshot*. This snapshot provides a consistent state: it records all modifications of transactions up to some point in the serial order, and contains no modifications from transactions after that point.

We provide consistent snapshots using the *epoch numbers* described in §4.1. Transactions either run in the current epoch or the previous epoch. In-the-past transactions run in an even earlier epoch. A new epoch starts every d seconds, where d is a system parameter. Our current implementation sets d to 1 second.

Now we describe how we manage our read-only snapshots. The tricky part is ensuring that the move to the next read-only snapshot is done in a way that ensures the snapshot provides a consistent state.

Our scheme for changing epochs uses the variables shown in Figure 4-3. T_{cur} identifies the epoch used to run read/write (current) transactions, while $T_{readonly}$ is the epoch number used by read-only (past) transactions. The $CurTable$ has an entry for each core; it reflects the epoch that core is using to run read/write transactions.

The *ReadonlyTable* also has an entry for each core; it records the epoch that core is using to run read-only transactions. Finally, T_{gc} indicates when it is safe to remove old versions.

One of the cores is responsible for moving the system to the next epoch. It does this by advancing T_{cur} to start the next read/write epoch. Later it will advance $T_{readonly}$, but it does this only after it knows that all cores are running read-write transactions in the latest epoch; at this point it is safe to run read-only transactions in the previous epoch because we know that there will be no more modifications in that epoch.

Before the start of each transaction, a core c does the following updates to advance its local epoch state:

$$CurTable[c] \leftarrow \max(T_{cur}, CurTable[c])$$
$$ReadonlyTable[c] \leftarrow \max(T_{readonly}, ReadonlyTable[c])$$

A core runs a read/write transaction using the most recent version for each record; if it observes a larger epoch number than it knows about, then we currently abort the transaction. Note that it is also possible to switch to the higher epoch and run the transaction in that epoch (in our experiments, aborts due to this issue are very infrequent). In Phase 2, if a transaction does a modification to a record whose version has a smaller epoch number, it creates a new version to hold the new update which points to the previous version.

A core runs a read-only transaction by accessing versions of records based on its current read-only epoch number: for each record it reads the version with the largest epoch number that is less than or equal to this number. When a read-only transaction reaches its end it is committed without any checking. This is correct by construction, since the latest versions $\leq T_{readonly}$ of records which are still reachable (i) reflect a prefix of the serial execution history, (ii) will not be modified by any transactions, (iii) and will not be garbage collected (see §4.8).

The core in charge of advancing epochs periodically updates $T_{readonly}$ as follows:

$$T_{readonly} \leftarrow \min_{c}(CurTable[c]) - 1$$

4.8 Garbage collection

This scheme allows read-only epochs to advance, but we also need to garbage collect old versions. An old version can be collected only after we are sure that no read-only transactions need it; this is what the *ReadonlyTable* is used for. The core in charge of advancing epochs also reads this table periodically, and updates T_{gc} in a similar fashion to the way it updates $T_{readonly}$:

$$T_{gc} \leftarrow \min_{c}(ReadonlyTable[c]) - 1$$

We run a periodic background tree walker that traverses each tree and removes unneeded versions by breaking the links to them and collecting their storage. An old version can be removed as soon as its epoch number is $\leq T_{gc}$ *and* there is a later version that can be used for read-only transactions running in $T_{readonly}$.

Physically removing logically deleted records (i.e., records that transactions delete) that are indeed the latest version, as discussed in §4.5, is also accomplished by this background tree walker; recall that these versions are tombstones. We proceed by reading a record's epoch number (without locking), and checking whether its version is $\leq T_{readonly}$. If so, we lock the node and check again. If the condition still holds we physically unlink the record from the tree and mark the record as deleted; this way any concurrent readers and writers that raced with the unlink will know to abort. Note that the check does not need to be $\leq T_{gc}$ (which is overly conservative) because it does not matter if a read-only transaction either reads a logically removed record or notices the absence of a record—both are equivalent from its point of view.

Our scheme usually requires at most two versions for each record; records that are rarely modified usually have just one version. There is a period at the beginning of an epoch when some records might need three versions, but this period is short.

However, because we run garbage collection asynchronously, our system does not enforce this as a hard limit; some chains can grow longer before the walker sweeps around again. §6.7 shows that this is not a big issue in our experiments.

Chapter 5

Implementation

This section describes our prototype implementation of MAFLINGO in more detail. The MAFLINGO storage layer is implemented in $\sim 20,000$ lines of C++. In the remainder of the section, we describe some low level details necessary to ensure correctness and implementation tricks that we use to achieve fast performance.

TID rollover. The system must ensure that the epoch number is incremented before the counter portion of a TID rolls over. It can do this by blocking if necessary (for at most one epoch). The system must also ensure that epoch numbers do not roll over. If truly necessary (using 32 bits for the epoch with one-second epochs means rollover happens once every 136 years), the checkpointing/garbage collection phase could ensure this by artificially advancing the epochs of very old records.

Record modification. For efficiency reasons, MAFLINGO's commit phase modifies record data in place when possible. This can expose garbage to concurrent readers instead of meaningful record data. A *version validation* protocol is used to ensure readers can detect garbage and either abort their transaction or retry. The version validation protocol uses an additional word per record called the *data version*, and a bit in that word, called the *dirty bit*. This word also stores additional important information, such as the latch bit and a bit indicating whether the record is deleted.

To *modify* record data during Phase 3 of the commit protocol, a writer (while the

latch is held) (1) sets the dirty bit; (2) performs a memory fence; (3) updates the record; (4) updates the transaction ID; (5) performs a memory fence; and (6) clears the dirty bit, releases the latch, and increments the data version in one atomic step. This write procedure is shown in Figure 5-1.

```
   Data: tuple to write to, new-contents, and tid
   /* Assume tuple is already latched                */
 1 tuple.version.dirty ← true
 2 fence()
 3 tuple.contents ← new-contents
 4 tuple.tid ← tid
 5 fence()
 6 atomic
 7 |   tuple.version.dirty ← false
 8 |   tuple.version.locked ← false
 9 |   tuple.version.counter++
```

Figure 5-1: Tuple write procedure

To *access* record data, a reader (1) checks the dirty bit and spins until it is clear; (2) takes a snapshot of the data version word; (3) performs a memory fence; (4) reads the transaction ID and data; (5) performs a memory fence; and (6) compares the record's data version to its snapshot. If the data version word changed, the reader must retry. This read validation is shown in Figure 5-2.

An alternate protocol might treat the record's latch as its dirty bit and its TID as its data version. However, the dirty bit and data version have several advantages. Large read/write transactions hold record latches for much longer than they modify record data. (A latch is held throughout the commit protocol, but modification happens record-by-record in Phase 3.) In addition, separating the TID, which is used during commit, from the data version, which is used simply to validate read consistency, can reduce memory contention.

As in the commit protocol, the memory fences in the read validation protocol are compiler-only on x86 and other TSO machines.

If it is not possible to do the update in place (e.g., because there is insufficient space in the memory block to hold the update), we allocate a new record with sufficient

```
   Data: tuple to read from
   Result: contents at tid
 1 retry:
 2 version ← tuple.version
 3 while version.dirty = true do
 4 |   version ← tuple.version
 5 fence()
 6 contents ← tuple.contents
 7 tid ← tuple.tid
 8 fence()
   /* =stable is equality ignoring the lock bit          */
 9 if version =stable tuple.version then
   |   /* valid read from tuple of contents at tid       */
10 else
11 |   goto retry
```

Figure 5-2: Tuple read validation procedure

space, set it to point to the old record, and atomically install the new record to map to the existing key. Performing this installation requires some care to avoid potential race conditions (the details of which we omit.)

Avoiding unnecessary memory allocations. We use the standard technique of maintaining thread-local memory arenas, and reusing memory from the arenas for per-transaction data structures, such as read/write sets. This reduction in pressure on the memory allocator had a non-trivial performance benefit in our experiments.

Inconsistent reads in aborting transactions. The protocol as presented does not force a transaction to abort until after it enters the commit phase, even if it reads different versions of a value or sees some but not all of another concurrent transaction's updates. This means stored procedures *that will ultimately abort* may see inconsistent values of data before they do so. Other OCC protocols [19] provide the same semantics, but if needed, we can prevent these situations through additional checks of versions on reads and by disabling the previously described dirty bit optimization. We did not employ these additional checks in our experiments.

Read-your-own-writes. In general, reading a record involves checking the *write-set* so a transaction sees its own writes. However, this check is unnecessary if a transaction never reads a record that it previously wrote. Simple static checks can detect this easily by checking if, for example, a stored procedure ever reads a table after updating it. In our experiments in §6, no transactions depended on reading their own writes, so we ran them with this check disabled.

Chapter 6

Evaluation

In this section, we evaluate the effectiveness of the techniques in Maflingo, answering the following questions:

- How does the performance of Maflingo compare with other state-of-the-art OLTP database systems (§6.2)?
- What is the overhead of Maflingo's read/write set tracking for simple key-value workloads (§6.3)?
- Do the techniques of Maflingo allow the system to scale as more cores become available (§6.3, §6.4)?
- How do the techniques of Maflingo compare to a partitioned data-store, especially as we vary the level of cross-partition contention (§6.5)?
- How do large read-only transactions affect Maflingo's performance, when compared to running the in present versus the past (§6.6)?
- How effective is Maflingo's garbage collector at reclaiming record versions that are no longer needed in the presence of many concurrent modifications (§6.7)?

We have done an initial evaluation of persistence to confirm that the overheads of our logging thread and disk writes do not substantially affect performance, but have

not fully implemented it and do not discuss it further in this evaluation section.

6.1 Experimental Setup

We used two machines to run our benchmarks, described below. We disabled Hyper-Threading on all CPUs on both machines for predictability of results.

Machine A. This machine has eight 10-core Intel Xeon E7-8870 processors clocked at 2.4GHz, yielding a total of 80 physical cores. Each core has a private 32KB L1 cache and a private 256KB L2 cache. Ten cores on a single processor share a 30MB L3 cache. The machine has 256GB of DRAM with 32GB of DRAM attached to each socket, and runs 64-bit Linux 3.2.

Machine B. This machine has four 8-core Intel Xeon E7-4830 processors clocked at 2.1GHz, yielding a total of 32 physical cores. Each core has a private 32KB L1 cache and a private 256KB L2 cache. Eight cores on a single processor share a 24MB L3 cache. The machine has 256GB of DRAM with 64GB of DRAM attached to each socket, and runs 64-bit Linux 3.2.

Experiments described in §6.3 and §6.4 were run on **Machine A**, whereas experiments described in §6.5, §6.6, and §6.7 were run on **Machine B**. In experiments where we vary thread counts, we increase thread counts one socket at a time for **Machine A** (we progress in increments of 10 threads), and half a socket at a time for **Machine B** (we progress in increments of 4 threads).

We pay careful attention to memory allocation and thread affinity in our experiments. When possible, we allocate memory on a specific NUMA node and pin worker threads to that node. For example, in our TPC-C experiments (§6.4, §6.5, and §6.6), we allocate the records associated with a particular warehouse on a NUMA node, and pin the worker thread(s) for the warehouse to that NUMA node. Before we run an experiment, we make sure to pre-fault our memory pools so that the scalability bottlenecks of Linux's virtual memory system [8] are not an issue in our benchmarks. Finally, we use 2MB "super-pages" (a feature offered by the x86-64 architecture and supported in recent Linux kernels) in our memory allocator to reduce TLB pressure.

In all graphs, each point reported is the median of three consecutive runs, with the minimum and maximum values shown as error bars. We ran each experiment for 60 seconds. In our experiments, we follow the direction of [24] and size both internal nodes and leaf nodes of our B$^+$-tree to be roughly four cache-lines (a cache-line is 64-bytes on our machine), and use software prefetching when reading B$^+$-tree nodes.

In the interest of measuring *only* the overhead involved in Maflingo's transactions versus other designs, our experiments are not run with any networked clients. We believe this is still meaningful, as [24] shows that commodity network hardware is not a bottleneck for high-throughput workloads. In our experiments, each thread combines a database worker with a workload generator. These threads run within the same process, and share Maflingo trees in the same address space. This is a different setup than other systems such as H-Store, which do *not* share trees within the same address space, even on a single machine. We use `jemalloc` as our memory allocator for all allocations minus records and tree nodes, which use the custom memory allocator described previously. Additionally, for our experiments, we did not enable logging/persistence and return results immediately. Once again, [24] shows that logging can be designed to not be a bottleneck in a high-throughput system. Finally, we set the epoch time of the system to one second.

Even though our machines have 80 and 32 physical cores, respectively, we do not run experiments using more than 70 and 28 database worker threads, respectively. This is so we can dedicate a few cores to performing background tasks in parallel, such as tree traversal and garbage collection (§4.8), which is a standard practice for many database systems. These background tasks are CPU intensive tasks that start to compete with worker threads for CPU time as we approach one worker per core. We do not believe this issue would affect scalability for larger core counts. It would be possible to set aside fewer background threads at the expense of less aggressive garbage collection; §6.7 shows that our current configuration is aggressive about collecting unneeded records to keep space overheads low.

6.2 Comparison to other OLTP systems

In this section we report the per-core throughput of several state-of-the-art OLTP systems from both industrial and research communities. We use the TPC-C [34] benchmark as the basis of comparison; see §6.4 for more details on the TPC-C benchmark. Here, we do not measure the other systems, but rather summarize the reported numbers in the literature (all of these numbers were generated on modern multicore machines).

System	Throughput (txns/sec/core)	Notes
H-Store [28]	$\sim 2,000$	Standard mix
Jones et al. [17]	$\sim 12,500$	Standard mix
Calvin [36]	$\sim 1,250$	100% new-order
Calvin$^+$ [30]	$\sim 2,500$	Standard mix
Shore-MT$^+$ [29]	$\sim 6,500$	Standard mix
Dora [26]	$\sim 2,500$	100% order-status
Maflingo	$\sim 32,000$	Standard mix (Figure 6-3)

Figure 6-1: Summary of various TPC-C benchmark reports

Figure 6-1 shows a summary of our findings. Note that Jones et al. [17], Calvin [36], and Calvin$^+$ [30] are all primarily focused on distributed transaction performance, but we include their reported numbers because they are reasonably competitive.

For each system, we show its best reported number for TPC-C. If a single core number was available, we reported it (the single core number always performed the best). Otherwise, to compute per-core throughput, we normalized the transaction throughput rate by the number of CPU cores (parallel hardware contexts, in the case of Dora) used in the corresponding benchmark. Since different systems were evaluated on different hardware and (sometimes) in configurations with overheads from disk and network I/O, the exact performance differences are not interesting. What is clear, however, is that Maflingo provides competitive, if not substantially higher per-core throughput than previous main-memory multicore databases.

6.3 Overhead versus Key-Value

Since MAFLINGO is layered on top of a fast, concurrent key-value store, we wanted to demonstrate the overhead of MAFLINGO's read/write set tracking versus no tracking in a key-value store.

For this experiment, we evaluate two systems. The first system, **Key-Value**, is simply the concurrent B$^+$-tree underneath MAFLINGO. That is, **Key-Value** provides only atomic gets and puts for a single key, and no transactions. The second system is MAFLINGO. We ran a variant of YCSB [9] workload mix A. YCSB is Yahoo's popular key-value benchmark. Our variant differs in the following ways: (a) we fix the read/write ratio to 80/20 (instead of 50/50), (b) we change the write transaction to a read-modify-write transaction, and (c) we shrink the size of records to 100 bytes (instead of 1000 bytes) . (a) is to prevent the memory allocator from becoming the primary bottleneck. (b) is to use an operation that actually generates read-write conflicts (note that for **Key-Value**, we implement the read-modify-write as two separate non-atomic operations). (c) is to prevent `memcpy` of the record values from becoming the primary bottleneck (this affects both systems equally). Both transactions sample keys uniformly from the key-space. We fix the tree size to contain 320M keys, and vary the number of database workers performing transactions against the tree.

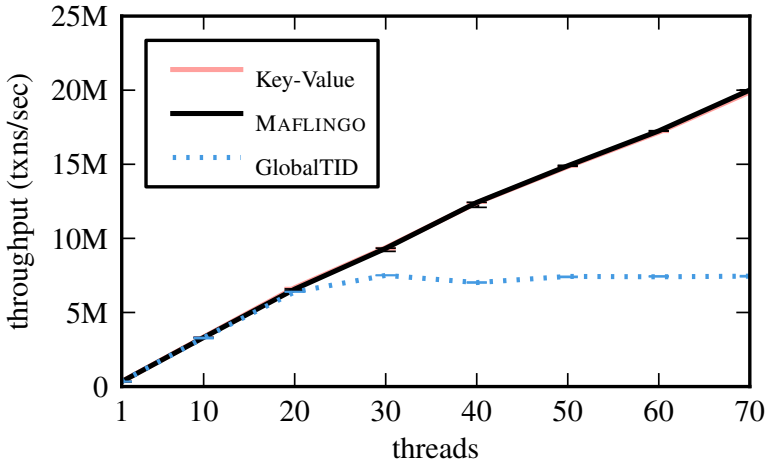

Figure 6-2: YCSB Benchmark

Figure 6-2 shows the results of running the YCSB benchmark on both systems (ignore **GlobalTID** for now). Here, the overhead of MAFLINGO compared to **Key-Value** is negligible; **Key-Value** out-performs MAFLINGO by a maximum of 1.02× at 10 threads.

6.4 Multi-core scalability of Maflingo

Our next experiment evaluates the scalability of MAFLINGO as we increase the number of transaction processing cores. We already saw linear scalability on YCSB in Figure 6-2 from the previous section. We now answer two more questions: (i) what did we gain from designing a commit protocol with no global coordination, and (ii) how does our protocol work on a more complex benchmark?

To answer the first question, we take MAFLINGO and add one atomic *compare-and-swap* (CAS) instruction on a single shared cache-line in the commit protocol, to simulate global TID generation. Note that this is exactly the critical section that the commit protocol from Larson et al. [20] has. We call this protocol **GlobalTID**. From Figure 6-2, we see that for low core counts, **GlobalTID** performs identically to MAFLINGO, which is expected. However, as soon as we hit 20 cores, we see the performance of **GlobalTID** start to plateau, as the atomic CAS becomes a contention point in the system.

To answer the second question, we use the popular TPC-C benchmark [34], which models a retail operation. All transactions in TPC-C are structured around a local warehouse. To run TPC-C on MAFLINGO, we assign each transaction processing thread a local warehouse to model the client affinity of TPC-C. For this experiment, we size the database such that the number of warehouses equals the number of processing threads, so the database grows as more threads are introduced (we fix the contention ratio of the workload). We do not partition any of the tables. For TPC-C, we do not model client "think" time, and we run the standard workload mix involving all five transactions. Note that the standard mix involves cross-warehouse transactions.

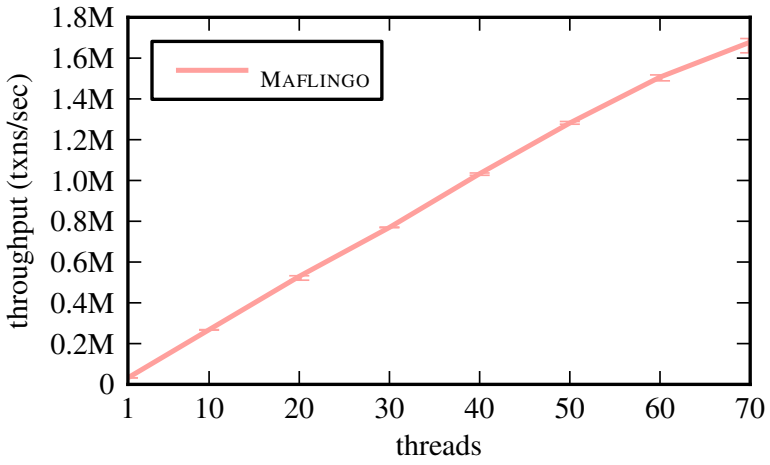

Figure 6-3: TPC-C Benchmark

Figure 6-3 shows the throughput of running TPC-C as we increase the number of threads (and thus warehouses) in MAFLINGO. Here, we see linear scale up to 60 threads, with a slight dip between 60 to 70 threads. The per-thread throughput at one thread is 1.29× the per-thread throughput at 60 threads, and 1.40× the per-thread throughput at 70 threads.

Note that even with zero-overhead concurrency control, we expect the single warehouse (one thread) workload to have slightly higher per-thread throughput than the multi-warehouse workloads. This is because the one thread running the single warehouse workload gets all 30MB of a shared L3 cache to itself, whereas each thread running the multi-warehouse workloads must share the 30MB L3 cache between nine other threads on the same socket. Indeed, the per-thread throughput at 10 threads drops to 1.07× the per-thread throughput at 60 threads, and 1.15× the per-thread throughput at 70 threads.

Furthermore, we believe the slight dip does not indicate any inherent bottleneck in our protocol, but rather is the result of competition over finite CPU resources. The high throughput rates between 60 to 70 threads result in many old items which must be processed and reaped by the garbage collector, and also larger indexes which must be scanned by the background tree walker. Because we parallelize both the garbage collector and the tree walker, these background threads start to compete for CPU

cycles with the worker threads. As noted in §6.1, we could dedicate less threads to these tasks at the expense of increased space overheads.

6.5 Overhead versus Partitioned-Store

In this section, we describe our evaluation of MAFLINGO versus a statically partitioned data store, which is a common configuration for running OLTP on a single shared-memory node [33, 26, 31]. TPC-C is typically partitioned around warehouses, such that each core is responsible for the districts, customers, and stock levels for a particular warehouse (the read-only items table is usually replicated amongst all partitions). This is a natural partitioning, because as mentioned in §6.4, each transaction in TPC-C is centered around a single local warehouse.

Here we fix the number of workers and vary the contention. This is in contrast to §6.4, where we fixed contention and varied the number of workers. Our goal is to show the scalability of our system in two dimensions: the level of parallelism available (§6.4), and the level of workload contention (this experiment).

The design of **Partitioned-Store** is motivated by H-Store[33] and is as follows. We physically partition the data by warehouse, so that each partition has a separate set of B^+-trees for each table. We do this within the same process to avoid the overhead of message passing/IPC. We then associate a global *partition lock* with each partition. Each transaction, before executing, first obtains all partition locks (in sorted order). Once all locks are obtained, the transaction can proceed as in a single-threaded data-store, without the need for doing record-level locking (2PL) or maintaining read/write sets (OCC). We assume that we have perfect knowledge of the partition locks needed by each transaction, so once the locks are obtained the transaction is guaranteed to commit and never has to acquire new locks. We implement these partition locks using fast spinlocks, and take extra precaution to allocate the memory for the locks on separate cache-lines in order to prevent false sharing.

Partitioned-Store uses the same B^+-trees that **Key-Value** and MAFLINGO use;

we left the concurrency control mechanisms in place in the B$^+$-tree. Mao et al. [24] show that the overhead of uncontended concurrency control is low (partition locks ensure zero contention within the tree), so we do not believe this adversely affected the experiments. We *do* remove the concurrency control for record values (which is something **Key-Value** had to do to ensure atomic reads/writes to a single key).

As in our previous experiments on TPC-C, MAFLINGO executes transactions by associating each thread with a local warehouse, and issuing queries against the shared B$^+$-trees; we do not physically partition the B$^+$-trees. No special action is needed if a transaction touches remote warehouses. Note that even if we had physically partitioned the B$^+$-trees, our system would naturally support cross-partition transactions without any modifications or special per-partition locks as in **Partitioned-Store**.

We focus on the most frequent transaction in TPC-C, which is the new-order transaction. Even though the new-order transaction is bound to a local warehouse, each transaction has some probability (which we vary) of touching non-local warehouses via the stock table; our benchmark explores the tradeoffs between MAFLINGO and **Partitioned-Store** as we increase the probability of a cross-partition transaction from zero to over 60 percent.

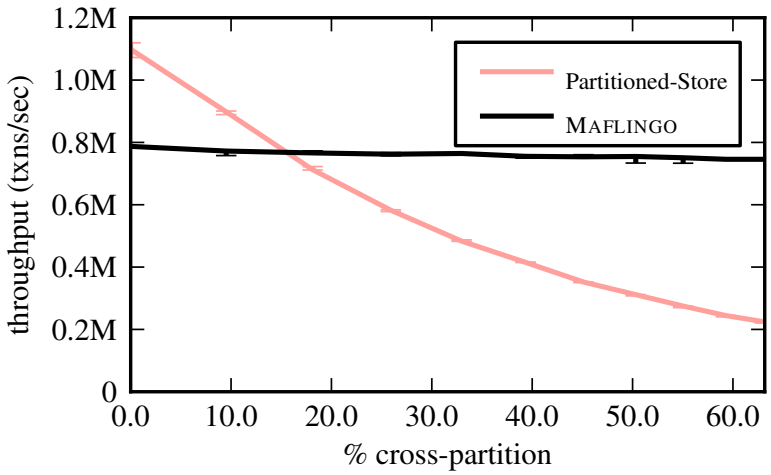

Figure 6-4: TPC-C New Order Benchmark

Figure 6-4 shows the throughput of running all new-order transactions on both MAFLINGO and **Partitioned-Store**. In this setup, we fix the number of warehouses

and the number of threads at 28. The parameter varied is the probability p that a particular single item is drawn from a remote warehouse. On the x-axis we plot the probability, as a function of p, that any given transaction will touch at least one remote warehouse (each new-order transaction includes between 5 to 15 items, inclusive) and thus require grabbing more than one partition lock in **Partitioned-Store**.

The curves in Figure 6-4 confirm our intuition about data partitioning; **Partitioned-Store** is clearly the optimal solution for perfectly partitionable workloads. The advantages are two-fold: no concurrency control mechanisms are required, and there is better cache locality due to the partitioned trees being smaller. **Partitioned-Store** out-performs MAFLINGO by 1.43× at $p = 0$. However, performance suffers as soon as even a small fraction of cross-partition transactions are introduced. At roughly 15%, the throughput of **Partitioned-Store** drops below MAFLINGO and gets worse with increasing contention, whereas MAFLINGO maintains a steady throughput throughout the entire experiment. At $p = 10$, MAFLINGO out-performs **Partitioned-Store** by 3.31×. The results here can be understood as the trade-off between coarse and fine-grained locking. While MAFLINGO's OCC protocol initially pays a non-trivial overhead for tracking record-level changes in low contention regimes, this work pays off as the contention increases. Note that the OCC overhead in Figure 6-4 is greater than the OCC overhead in Figure 6-2 because the read/write set of the TPC-C new-order transaction is much larger than the YCSB transactions.

Discussion. Although these results show that partitioning is not *required* for performance, our main conclusion is not that partitioning is a bad idea. Clearly, partitioning is required to get good performance in a distributed setting, and our results show that in a local setting with few cross-partition transactions, partitioning can beneficial. Hence, one of our future work goals is to try to obtain the benefits of partitioning without requiring it explicitly, by exploring soft-partitioning approaches and/or hybrid protocols (see §7.1 for more details).

6.6 Effectiveness of read-only queries

The next question we answer is how effective MAFLINGO's read-only query mechanism is in the face of concurrent updates. To evaluate this part of our system, we again use the TPC-C benchmark, changing the setup as follows. We fix the number of warehouses to 8, and the number of threads to 16 (each warehouse is assigned to two threads). We run a transaction mix of 50% new-order, and 50% stock-level, which is the largest of two read-only queries from TPC-C. On average, stock-level touches several hundred records in frequently updated tables (by the new-order transaction), performing a nested-loop join between the order-line table and the stock table.

We measure the throughput of MAFLINGO under this load in two scenarios: one where we use MAFLINGO's read-only versions to execute the stock-level transaction (**Snapshots**, labelled **SS**) at roughly one second in the past, and one where we execute stock-level as a regular transaction (**No-Snapshots**, labelled **No-SS**) in the present. In both scenarios, the new-order transaction executes in the same way. Figure 6-5 shows the results of this experiment. The x-axis is the same probability p that we varied in Figure 6-4, except here we actually plot p on the x-axis (so at $p = 100$, every item is serviced by a remote warehouse).

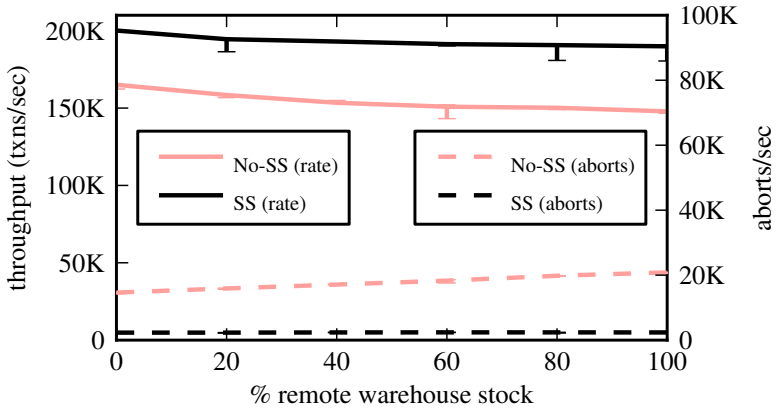

Figure 6-5: TPC-C Full Benchmark

Figure 6-5 shows that using slightly stale read-only snapshots outperforms executing read-only queries in the present by a minimum of 1.18× at $p = 20$, and a maximum of 1.34× at $p = 60$. This is due to the larger number of aborts that oc-

cur when a large read-only query executes with concurrent modifications (recall that read-only snapshot queries never abort). Here, the abort rate is shown in the dashed lines in the figure. As the contention rate increases, we see that the abort rate for **No-Snapshots** grows, but stays constant for **Snapshots**.

6.7 Effectiveness of garbage collector

Finally, in this section we demonstrate that our previous benchmark numbers are not artificially inflated due to a lack of garbage collection. In all our experiments we allowed the garbage collector to run. Recall from §4.8 that multiple versions of records are kept in order to support read-only transactions. For infrequently updated records, we expect exactly one version; for frequently updated records, we expect mostly two versions. If we had perfect garbage collection, there would never be more than three versions for a record.

Extra versions	YCSB (§6.3)	TPC-C (§6.4)
0	0.641	0.274
1	0.927	0.990
2	0.990	0.991
3	0.999	0.991
4	0.999	0.992

Figure 6-6: Cumulative distribution of the number of versions of records in the benchmarks from §6.3 and §6.4.

Figure 6-6 shows the cumulative breakdown of the number of *extra* versions for all records in the database *immediately* after running the experiments described in §6.3 and §6.4 on **Machine B** with 28 threads. For example, ∼93% of records in YCSB had either zero or one extra versions after the experiment from §6.3. This figure shows that for both experiments, our garbage collector kept the number of extra versions ≤ 2 (the maximum number of versions ever needed) for over 99% of all records.

Chapter 7

Conclusions

In this thesis, we presented Maflingo, a new OCC-based serializable database storage engine designed to scale to large multicore machines without requiring the use of partitioning. Maflingo's concurrency control protocol is optimized for multicores, avoiding global critical sections and non-local memory writes for read operations. It employs a multi-versioning scheme to support efficient epoch-based memory management. This versioning scheme allows us to support in-the-past read-only transactions without tracking read sets and in a way that guarantees they will never abort. Overall our results show that Maflingo performs well, providing (i) linear scalability on YCSB and TPC-C, (ii) raw transaction throughput on TPC-C that exceeds that of other recent prototype databases designed for multicore environments, and (iii) low overheads relative to a non-transactional system . Together, these results show that transactional consistency and scalability *are* possible on modern machines without the use of partitioning.

7.1 Future Work

Given the encouraging performance of Maflingo, we are investigating a number of areas for future work, designed to further improve its performance and augment its usability. In this section we expand upon these ideas in more detail.

Hybrid partitioning. We saw in Figure 6-4 that the performance of partitioned

data stores was better than MAFLINGO for workloads with low cross-partition transactions. Can we design a hybrid commit protocol which executes effectively using partition locks when the system has low cross-partition contention, and falls back to MAFLINGO after some threshold, getting the benefits of both protocols? We have done some preliminary work exploring several algorithms with this flavor.

Hybrid data structures. MAFLINGO's main index data structure is a variant of a B^+-tree, in the interest of supporting range scans. However, typical OLTP workloads only scan small ranges of the key space at a time, so storing all the keys of a table in sorted order is most likely un-necessary. Given this, can we design efficient data structures which trade off large range scans for faster point-wise lookups? A simple idea would be to logically partition the key space and maintain separate B^+-trees per partition, similar to how **Partitioned-Store** operates. We are investigating more sophisticated techniques.

Hybrid locking versus OCC. Another observation to be made for typical OLTP workloads is that writes are often non-uniformly distributed over the key space. For example, in TPC-C, it is much more likely for a transaction to update an entry corresponding to a particular district than an entry corresponding to a particular customer. For very high contention workloads, it is often the case that the conflicts are centered around a single record or a few records. These kind of workloads perform very poorly in OCC, and often result in livelock for transactions as they continually abort. On the other hand, OCC has superior performance in lower contention regimes. We are interested in exploring protocols which decide whether to use OCC or locking on a *per-record* basis. This way, we could fallback to locking a few very contentious records and speculate on the remaining records, allowing our system performance to avoid degenerating into livelock.

Persistence/Recovery. Our current system does not have a fully implemented persistence layer. While techniques to persist transactions are well understood, the tricky part is to persist transactions while avoiding global critical sections on the critical path. Even though a lot of these issues can be solved with a background persistence worker, there is still the issue of the overhead of communicating data

(transaction updates) between worker and persistence threads. Our persistence story is further complicated by the lack of a globally unique transaction identifier. There are many possible design points in this space; command logging versus data logging and centralized log versus multiple logs are just some of the designs that need to be evaluated. We currently have a design which addresses many of these issues, which we aim to fully evaluate in the future as a separate piece of work.

Distributed Maflingo. Regardless of main memory sizes, at some point it will become infeasible to run entire workloads on a single database server. Making Maflingo run in a distributed setting while retaining its performance without sacrificing consistency is another interesting direction of future work. Recently, there has been a lot of work on distributed commit protocols including systems built around determinism [35, 36] and systems which use distributed consensus protocols to agree on log record positions [2, 10]. Whether or not distributed Maflingo can use any of these techniques successfully is an open question; for instance, the most obvious way to make Maflingo deterministic is through the use of deterministic thread scheduling which can introduce significant overhead.

Out of core execution. So far in Maflingo we have assumed that the entire database will fit in main-memory. As the data grows, this is no longer the case, especially with high transaction throughput rates. While it seems straightforward to write out parts of the database to secondary storage when running out of main-memory, getting the details right can be tricky. Recent work [22] has focused on algorithms for identifying which tuples are considered "cold" based on the workload, and are thus good candidates for eviction from main-memory. However, to the best of our knowledge there has not been any extensive study of the various trade-offs when implementing such a system.

Performance Modeling. While we have done an empirical study of the performance of partitioning versus shared everything designs, we do not have a good analytical understanding of the trade-offs, especially for a large multicore database. We are interested in developing analytical models to predict performance of various main-memory database designs (partitioning versus sharing, OCC versus 2PL, etc.).

While a lot of fundamental work in this area has been done in the classic database literature, the classic models understandably do not take into consideration details such as contention in the cache coherency layer, non-uniform memory access speeds, and other details which are critical to database performance on modern multicores. Recent work [4] has shown promising results in modeling these low level hardware details via queuing theory and Markov models; we hope to extend this work to MAFLINGO.

Bibliography

[1] Divyakant Agrawal, Arthur J. Bernstein, Pankaj Gupta, and Soumitra Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1), 1987.

[2] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[3] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control theory and algorithms. *ACM TODS*, 8(4), 1983.

[4] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Linux Symposium*, 2012.

[5] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *PPoPP*, 2010.

[6] Michael J. Carey and Waleed A. Muhanna. The performance of multiversion concurrency algorithms. *ACM TODS*, 4(4), 1985.

[7] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, 2001.

[8] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Eurosys*, 2013.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[11] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2), 2010.

[12] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11), 1976.

[13] Keir Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003.

[14] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *SIGMOD*, 2008.

[15] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12), 2007.

[16] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-mt: a scalable storage manager for the multicore era. In *EDBT*, 2009.

[17] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, 2010.

[18] Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, 2013.

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

[20] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4), 2011.

[21] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM TODS*, 6(4), 1981.

[22] Justin Levandoski, Per-Åke Larson, and Radu Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, 2013.

[23] Justin Levandoski, David Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware. In *ICDE*, 2013.

[24] Yandong Mao, Eddie Kohler, and Robert Morris. Cache craftiness for fast multicore key-value storage. In *Eurosys*, 2012.

[25] C. Mohan. Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *VDLB*, 1990.

[26] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2), 2010.

[27] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. Plp: page latch-free shared-everything oltp. *Proc. VLDB Endow.*, 4(10), 2011.

[28] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *SIGMOD*, 2012.

[29] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. Oltp on hardware islands. *Proc. VLDB Endow.*, 5(11), 2012.

[30] Kun Ren, Alexander Thomson, and Daniel J. Abadi. Lightweight locking for main memory database systems. *Proc. VLDB Endow.*, 6(2), 2012.

[31] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database engines on multicores, why parallelize when you can distribute? In *Eurosys*, 2011.

[32] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *Proc. VLDB Endow.*, 4(11), 2011.

[33] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007.

[34] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). `http://www.tpc.org/tpcc/`, June 2007.

[35] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3(1-2), 2010.

[36] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.