High Performance Transactions in Deuteronomy

Justin Levandoski David Lomet Sudipta Sengupta Ryan Stutsman Rui Wang

Microsoft Research, Redmond, WA
{justin.levandoski, lomet, sudipta, rystutsm, ruiwang}@microsoft.com

ABSTRACT

The Deuteronomy architecture provides a clean separation of transaction functionality (performed in a transaction component, or TC) from data management functionality (performed in a data component, or DC). In prior work we implemented both a TC and DC that achieved modest performance. We recently built a high performance DC (the Bw-tree key value store) that achieves very high performance on modern hardware and is currently shipping as an indexing and storage layer in a number of Microsoft systems. This new DC executes operations more than 100× faster than the TC we previously implemented. This paper describes how we achieved two orders of magnitude speedup in TC performance and shows that a full Deuteronomy stack can achieve very high performance overall. Importantly, the resulting full stack is a system that caches data residing on secondary storage while exhibiting performance on par with main memory systems. Our new prototype TC combined with the previously re-architected DC scales to effectively use 48 hardware threads on our 4 socket NUMA machine and commits more than 1.5 million transactions per second (6 million total operations per second) for a variety of workloads.

1. INTRODUCTION

1.1 Deuteronomy

The Deuteronomy architecture [11, 17] decomposes database kernel functionality into two interacting components such that each one provides useful capability by itself. The idea is to enforce a clean, layered separation of duties where a transaction component (TC) provides concurrency control and recovery that interacts with one or more data components (DC) providing data storage and management duties (access methods, cache, stability). The TC knows nothing about data storage details. Likewise, the DC knows nothing about transactional functionality – it is essentially a key-value store.

An initial implementation [11] demonstrated the feasibility of Deuteronomy via a TC and a number of modest local and cloud-based DCs, though its performance was not competitive with the latest high performance systems. But this low performance was not fundamental to the Deuteronomy architecture. Subsequently, an effort to redesign each Deuteronomy component for high performance on modern hardware led to the Bw-tree latch-free access method [13] and LLAMA [12], a latch-free, log structured cache and storage manager. The result was a key-value store that executes several million operations per second that is now used as the range index method in SQL Server Hekaton [2] and the storage and indexing layer in several other Microsoft products, including

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA.

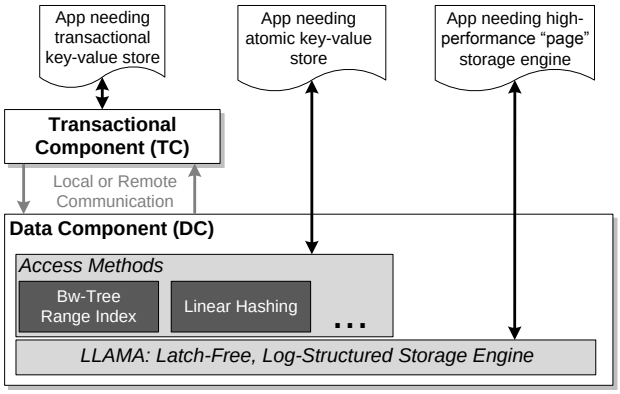

Figure 1: Deuteronomy storage engine architecture.

Azure DocumentDB [29]. Not only was the DC implementation a performance success, but it also showed that the DC could be further decomposed (see Figure 1) to also maintain a hard separation between access methods and the LLAMA latch-free, log structured cache and storage engine.

With a DC capable of millions of operations per second the original TC became the immediate bottleneck. Architected in a "traditional" manner (undo/redo recovery, lock manager, etc.), it was limited to a few tens of thousands of operations per second. Clearly, a new TC design was needed for a high performance transactional key-value store.

This paper confirms the performance story for the full Deuteronomy stack by describing the design and implementation of a high performance transaction component. It describes how the redesigned TC architecture achieves a two order of magnitude speedup to match our DC performance. **Further, the full stack is *not* a main memory-only system; rather, it is a "traditional" transactional system where data is stored on secondary storage and is only cached in main memory.** This shows that such a system can rival main memory system performance while being able to serve substantially more data than can fit in main memory.

The techniques that we use in our new TC are vastly different from the traditional lock management and redo/undo recovery. Nonetheless, the essential nature of the Deuteronomy architecture remains unchanged: the TC can interface with any number and flavor of DC key-value stores, whether local or remote.

1.2 Performance Factors for a New TC

Achieving a two orders of magnitude performance gain requires serious new thinking. We are driven by a number of fundamental design principles.

1. **Exploit modern hardware.** Our TC exploits lessons learned building Hekaton and the Bw-tree. Latch-freedom, log structuring, and copy-on-write delta updates that avoid update-in-place are well-suited to modern multicore machines

with deep cache hierarchies and low-latency flash storage. The TC takes advantage of all of these techniques.

2. **Eliminate high-latency from the critical-paths.** DC access latency can limit performance, especially for remote DCs. This is particularly bad for hotspot data where the maximum update rate of 1/latency (independent of concurrency control approach) can severely limit performance. TC caching is essential to minimize latency.

3. **Minimize transaction conflicts.** Modern multi-version concurrency techniques [8, 16] demonstrate the ability to enormously reduce conflicts. Deployed systems like Hekaton have proven that MVCC performs well in practice. We also exploit MVCC in the TC.

4. **Minimize data traffic between TC and DC.** Data transfers are very costly. Our "distributed" database kernel requires some data to be transferred between TC and DC. We strive to limit this burden as much as possible.

5. **Exploit batching.** Effective batching often can reduce the per "item" cost of an activity. We exploit batching when shipping data updates to the DC.

6. **Minimize data movement and space consumption.** Obviously, one wants only "necessary" data movement. By putting data in its final resting place immediately (within the TC), we can avoid what is very frequently a major performance cost, while reducing memory footprint as well.

1.3 TC Design Overview

A TC is only part of a transactional key value store or database kernel; its function is to provide transactional concurrency control and recovery. Our approach is to weave the factors listed in the prior subsection into all aspects of the TC design. Figure 2 presents a schematic of our TC architecture, illustrating the flow of data within it and between TC and DC. The TC consists of three main components: (a) an *MVCC component* to manage the concurrency control logic; (b) a *version manager* to manage our redo log (also our version store) as well as cached records read from the DC; and (c) a *TC Proxy* that lives beside the DC and whose job is to submit committed operations to the DC. The DC maintains database state.

1.3.1 Caching

One pervasive issue we faced was what it meant to cache data at the TC. Since we use MVCC, we knew we would have versions cached somewhere for concurrency control purposes. Versions resulting from updates are written into the redo recovery log. These recovery log versions are accessed via the MVCC component, which stores version offsets as part of its version entry and requests them through the version manager interface. Our version manager uses the redo log as part of the TC record version cache. In-memory log buffers are written to stable storage and retained in memory to serve as a version cache until they are eventually recycled and reused.

Versions of data not yet updated need to be acquired from the DC. To make them accessible to MVCC, the version manager retains these versions in the *read cache*. Both read cache and recovery log buffers are subject to different forms of log structured cleaning (garbage collection). Thus, an MVCC request to the version manager could hit (1) the read cache; or (2) a log buffer (in-memory or stable). Section 3 provides the details of efficient cache management.

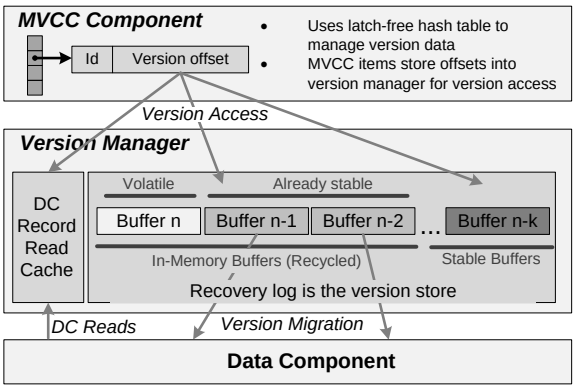

Figure 2: Data flow in the transactional component (TC)

1.3.2 Recovery Implications

To minimize data movement, we immediately post updates in their "final resting place" on the recovery log. Because we are using the recovery log buffers as part of our cache, we use pure redo logging to avoid diluting the cache with undo versions.

Immediately logging updates means that uncommitted updates are on the log without any means of undoing them if their transaction is aborted. So we cannot post updates to the DC until we know that the containing transaction has committed. Updates from aborted transactions are simply never applied at the DC.

1.3.3 Posting Changes to the DC

The TC includes a *TC Proxy*: a module that resides on the same machine as the DC. The TC Proxy receives log buffers from the version manager after the buffer is made stable and posts updates to the DC as appropriate. Since we use pure redo logging, these updates are "blind", in that they do not require first reading a pre-image of the record for undo. Posting can only be done after the transaction responsible for the update has committed. This posting is part of log buffer garbage collection when the TC Proxy and DC are collocated with the TC. Otherwise cleaning occurs once the buffer has been received by a remote TC Proxy. Posting updates to the DC is not part of the latency path of any operation and is done in the background. However, it is important for it to be somewhat timely, because it constrains the rate at which MVCC entries can be garbage collected.

1.4 Contributions

Deuteronomy's architecture enables a flexible configuration of data management systems. This paper focuses on the transactional component as a piece separate from, but able to exploit, our previous high performance Bw-tree key value store or any key value store used as a DC. While living within the Deuteronomy architecture, we have achieved two orders of magnitude performance gains over our previous TC design by using:

1. Multi-version concurrency control exploiting a variant of timestamp order concurrency control (Section 2).

2. Fast transaction commit that avoids read-only transaction difficulty in a highly efficient way. (Section 2)

3. Commit records in the recovery log as a queue for the deferred delivery of transaction outcome messages, once the recovery log buffer is durable. (Section 2)

4. Version management that exploits the recovery log buffers as a log structured store cache at the TC (Section 3).

5. Batching updates in log buffers when sending them to the DC and posting them from our TC Proxy (Section 4).
6. Applying each update by the TC Proxy at the DC using a blind write operation that does not require reading a prior version (Section 4).
7. New latch-free cache mechanisms in buffer management and epochs that remove performance bottlenecks (Section 5).

We ran a number of experiments (Section 6) that illustrate the performance of the newly designed and implemented TC. Of particular note, while we can run our transactional key-value store solely in-memory and without transaction durability, we report results based on a data component on stable storage (the Bw-tree) and a durable log enforcing durable commit.

2. CONCURRENCY CONTROL

A number of research prototypes [8, 16] and system implementations [2] have confirmed that using multi-version concurrency control (MVCC) is one key to achieving high performance. In particular, it can mostly eliminate read-write conflicts (by reading a version earlier than an uncommitted writer's version). We briefly describe our approach here.

2.1 Timestamp Order MVCC

Timestamp order (TO) concurrency control is a very old method [23], including a variant that uses multiple versions. The idea is to assign a timestamp to a transaction such that all of its writes are associated with its timestamp, and all of its reads only "see" versions that are visible as of its timestamp. A correct timestamp order schedule of operations using the timestamps is then enforced. Transactions are aborted when the timestamp ordering cannot be maintained. Recent work in Hyper [6, 21] showed that, with very short transactions, TO can work well, even in the single version case.

It is possible to use multiple versions to support weaker levels of transaction isolation. For example, Hekaton's design point is snapshot isolation because it avoids validating read/write conflicts (serializability is possible at the cost of validation). However, our focus is on enabling serializability. A real plus for TO is that no validation step is required at the end of a transaction. All validation happens incrementally as versions as accessed.

The TC tracks transactions via a *transaction table*. Each entry in the table denotes a transaction status, its transaction id, and a timestamp issued when the transaction starts. The entry for each version in the MVCC hash table (see §2.2 below) is marked with the transaction id that created the version. This permits an easy check if the transaction information for each version, including its timestamp.

The status in the transaction table entry indicates whether the transaction is active, committed, or aborted. Periodically, we compute the oldest active transaction (the one with the oldest timestamp), or *OAT*, which is used to determine which version information is safe to garbage collect.

2.2 Latch-free MVCC Hash Table

We maintain a latch-free hash table to manage MVCC data. Versions are hashed to a bucket based on their key. Each bucket item represents a record and contains the following entries: (a) a fixed-sized hash of the record key; (b) a pointer to the full key (keys can be variable length); (c) the timestamp of the youngest transaction that read the record; and (d) a version list describing the version(s) for the record. Each record item is fixed length for performance: it is both allocator friendly, and it guarantees items stay within cache line size (important if threads are simply "passing by" looking for other items in the bucket chains). To perform record lookup, the fixed-sized hash is compared to the hash of the record key; if the hashes match the full key pointer is dereferenced and full key comparison is performed to ensure it is the correct key. The version list is then traversed to find the appropriate version to read. The last read timestamp on each item is used in our timestamp order concurrency control approach. It represents the last read time of the most recently written version, and protects a younger transaction's read by ensuring that an older transaction cannot write a new version that the younger transaction should have seen. Only the youngest version needs this read protection. Older versions are protected by the younger versions. Further, a read time for the youngest version only needs to be written when it is later than the read time already present.

Version list items are fixed size and contain: (a) the transaction id of the creating transaction; (b) a version offset, used to reference the payload from the version manager; and (c) an "aborted" bit used to signify that this version is garbage and the transaction that created it has aborted – this is used as a fast track for the garbage collection process (see §2.4)

Both the per-bucket record lists and the per-record version lists are entirely latch-free. New entries are prepended to lists using a compare-and-swap. Removing entries requires multiple steps. When an entry in a list is no longer needed its "next" pointer is atomically marked with a special "removed" bit using a compare-and-swap. Future traversals over the list complete the unlinking of the item. This approach avoids races between the unlinking of an item and its predecessor: without care, this could otherwise result in an unlinked item "coming back to life."

To provide pointer stability for all latch-free data structures, we use an epoch mechanism that ensures that a memory location (for an MVCC record item, version item, etc.) is never reused until it is guaranteed that no active thread can deference a pointer to it. We improved upon our prior epoch mechanism (described in [12]) by reducing need for several atomic operations; we describe epoch management in Section 5.2.

2.3 Committing Transactions

2.3.1 Fast Commit

We use what has been called the "fast commit" optimization [1], where the TC acts as if the transaction has committed once its commit record is written to the recovery log buffer, except for notifying users of the commit. We wait until the recovery log buffer containing the commit record is stable before notifying the user. This works well for read/write transactions for which we write commit records.

Read only transactions typically do not write such a commit record. Instead they have been considered committed immediately once the commit operation is issued. Without care, however, this could lead to a logical bug after crash recovery: a read-only transaction may have read from a transaction that wasn't durable and which would be aborted during recovery. To avoid this problem, by default we write a commit record for read only transactions, delaying commit notification until everything that they read is stable.

However, writing commit records for all read-only transactions became a performance bottleneck. For read-heavy workloads these commit records dominated the recovery log contents and generated extra disk I/O. As a result, we optimized them away in the (common) case where everything read by a read-only transaction came from durably committed transactions by commit time. As a transaction reads versions, it tracks the highest commit log

sequence number (LSN) among all the transactions from which it has read. At commit time, a commit record is written to the log only if that highest commit LSN is not yet durable. If it is durable, then all versions read by the transaction are already stably committed. There is no threat that any version it read will disappear in the event of a crash, so no commit record is needed. In this case, read-only transactions return the commit message immediately.

2.3.2 Durable Commit

We have no separate mechanism to enqueue transaction outcome messages. Rather, we use the commit records in the recovery log as the queue of outcome messages. Each commit record contains the return message that is to be sent when the transaction is durably committed.

Once a recovery log buffer is on the stable log, we scan it for transaction commit records. We link commit records together to enable us to skip over the intervening operation log records, which will usually be the vast majority (e.g. 90%) of the log. During this scan, we read the commit records, which contain the outcome messages, and notify the transaction users that their transaction has committed. The commit scan is performed just before sending the buffer to the TC Proxy.

2.4 Garbage Collection

The MVCC table needs to maintain versions that can be seen by uncommitted transactions. We do this conservatively by identifying versions that are not needed by these transactions or that are available from the DC.

1. Any updated version older than the version visible to the OAT cannot be seen by active transactions and can be discarded from the hash table. These versions are never needed again.
2. Any version visible to the OAT but with no later updates can also be discarded once it is known to have been applied at the DC. We are guaranteed to be able to retrieve such a record version from the DC. If we could not delete these entries, the MVCC table would eventually contain the entire database.

Versions needed but not present in the MVCC table are read from the DC.

Section 4 describing the TC Proxy explains how the TC concisely reports to the TC progress of installing versions at the DC, and Section 6.5 evaluates the impact of MVCC garbage collection on overall TC performance.

3. MANAGING VERSIONS

3.1 Version Sources

Providing fast access to versions is critical to high performance in Deuteronomy. The TC serves requests for its cached versions from two locations. The first is directly from in-memory recovery log buffers. The second is from a "read cache" used to hold hot versions that may not have been written recently enough to remain in the recovery log buffers.

3.2 Recovery Log Caching

The TC's MVCC approves and mediates all updates, which allows it to cache and index updated versions. To make this efficient, the TC makes dual-use of both its recovery log buffers and its MVCC hash table. When a transaction attempts an update, it is first approved by MVCC. This permission, if given, results in the new version being stored in a recovery log buffer within the version manager. Afterward, an entry for the version is created in the MVCC hash table that contains an offset to the version in the recovery log and associates it with the updating transaction. Later reads for that version that are approved by the MVCC can directly find the data in memory using the version offset. Thus, in addition to concurrency control, the MVCC hash table serves as a version index, and the in-memory recovery buffers play the role of a cache.

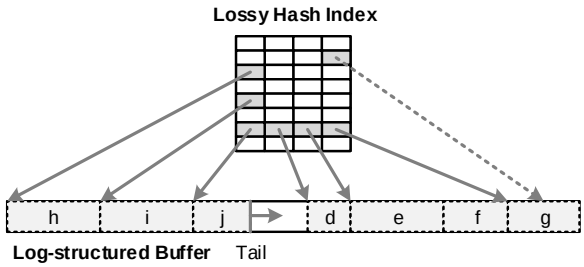

Figure 3: The read cache is structured as two lock-free structures. A lossy hash index maps opaque 64-bit identifiers to offsets within a large log-structured buffer.

Each updated version stored at the TC serves both as an MVCC version and as a redo log record for the transaction. The TC uses pure redo logging and does not include before images in these log records. Versions are written immediately to the recovery log buffer to avoid later data movement. This means that an update (redo log record) cannot be applied at the DC until its transaction is known to be committed, since there is no way to undo the update. The TC Proxy (§4) ensures this.

Recovery log buffers are written to stable storage to ensure transaction durability. However, our use of the buffers as a main memory version cache means recovery buffers are retained in memory even after they have been flushed to disk. Buffers are lazily recycled via a log structured cleaning process [24, 25] that results in the relevant updates being sent by the TC Proxy to the DC. The version manager initiates this process by lending or sending stable buffers to the TC Proxy depending on whether the DC is local or remote.

3.3 Read Cache

The recovery log acts as a cache for recently written versions, but some read-heavy, slow-changing versions are eventually evicted from the recovery log buffers when they are recycled. Similarly, hot read-only versions may preexist in the DC and are never cached in the recovery log. If reads for these hot versions were always served from the DC, TC performance would be limited by the round-trip latency to the DC.

To prevent this, the TC's version manager keeps an in-memory *read cache* to house versions fetched from the DC. Each version that is fetched from the DC is placed in the read cache, and an entry is added to the MVCC table for it. As a further optimization, the version manager relocates uncommitted and hot versions into the read cache from recovery log buffers that are about to be recycled.

The read cache is latch-free and log-structured, similar to recovery log buffers (§5.1). One key difference is that the read cache includes a lossy, latch-free index structure; this index provides a level of indirection that allows versions to be addressed by an opaque 64-bit identifier. This simplifies relocating versions into the cache from recovery log buffers. The MVCC table refers to versions by their log-assigned offsets, and the index allows a version to be relocated into the cache without having to update references it.

Figure 3 illustrates the read cache. Versions are added to the buffer in a ring-like fashion, overwriting objects that were stored in the previous "lap" over the buffer. New versions are added to the read

cache in two steps. First, the "tail" offset of the log-structured buffer is atomically advanced. The tail offset is monotonically increasing and never wraps; it is mapped to a virtual address inside the buffer using the tail modulo buffer size. Once the tail has been advanced, space has been reserved for the new version in the buffer. The version's 64-bit identifier, the size of the version, and the version data itself is copied into the reserved space. Then an entry is added to the hash index mapping the 64-bit identifier to the tail offset where it was copied.

In the process of reserving space for a new version, the tail offset "passes over" older versions that were placed into the buffer earlier and new data is copied on top of the old data. For example, in Figure 3, if a new object *k* is allocated at the tail, then its reservation may extend into object *d* (or further). For older versions (like *d* after *k* is appended or *g*), offsets stored in the hash index may "dangle," pointing to places in the log buffer that have since been overwritten.

The read cache makes no attempt to fix this up: lookups must treat the offsets returned by the hash index as hints about where entries might be in the buffer. Lookups must check the current tail offset against the offset given by the index both before and after they copy data out of the buffer for safety (some additional care is needed when copying the version out of the buffer to ensure that an overwritten size field doesn't cause memory corruption).

Not only does the index sometimes point to locations in the buffer that have been overwritten, but it also "forgets" mappings over time. When an entry is installed it is added to a row in the index based on the hash of its 64-bit identifier. Within the row, the word with the lowest buffer offset is overwritten. Effectively, the index has a row-by-row FIFO eviction policy of mappings. As a result, the hash index may even forget about versions that are still accessible in the read cache buffer (version *f*). The size of the index and the buffer are co-calculated to minimize these mismatches; however, no other attempt is made to synchronize evictions between the buffer and the index. Cache semantics make this safe: a missing entry in the index manifests itself as a cache miss.

As a log-structured ring buffer, the read cache is naturally populated and reused in FIFO order. So far, this policy has been sufficient for us; it symbiotically works with the TC's MVCC to naturally preserve hot items. This is because whenever a record is updated it is "promoted" to the tail of the record log buffers, naturally extending the in-memory lifetime of the record (though, via a new version). The read-cache could be augmented with a second-chance cleaning policy that would copy read-only hot versions from the head to the tail instead of overwriting them. So far, our metrics indicate there is little incentive to make this optimization.

Our read cache is similar to the concurrently developed MICA key-value store [14], though we were unaware of its design until its recent publication.

3.4 Latch-free Buffer Management

Similar to the LLAMA cache/storage subsystem [12], posting versions to either recovery log buffer or read cache buffer is done in a fully latch-free fashion. It is the buffer space reservation that requires coordination, while copying of data can proceed in a thread safe manner without coordination. We have improved the scalability of buffer space reservation by using atomic-add instead of a compare-and-swap (see §5).

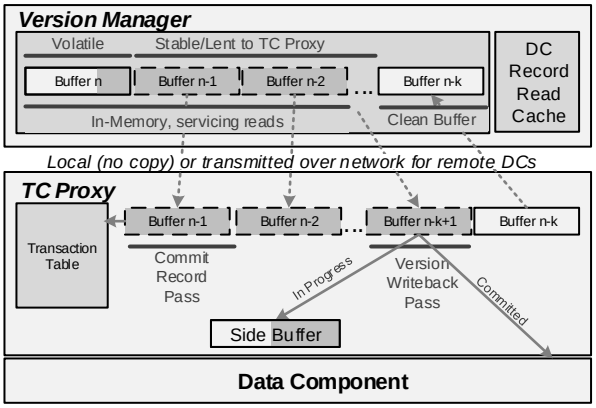

Figure 4: The TC Proxy receives full, stable log buffers from the TC. An eager pass updates transaction statuses; a lazy pass applies committed operations to the DC.

4. TC Proxy

The TC Proxy's main job is to receive stable recovery log buffers from the TC and efficiently apply the versions within them to the DC. The TC Proxy runs co-located with the DC. It enforces a well-defined contract that separates the TC and the DC sufficiently to allow the DC to run locally with the TC or on a remote machine.

4.1 Buffer Communication

Once a recovery log buffer is stable, it is sent to the TC Proxy; the recovery log buffer itself acts as an update batch. It is only at the TC Proxy that the operations are unbundled and submitted to the DC as appropriate.

When the DC is local, it avoids copying the recovery log buffer by borrowing a reference to it from the version manager. In the case that the DC is remote, the networking subsystem similarly holds a reference to the buffer, but only until the remote TC Proxy has acknowledged its receipt.

4.2 Interaction Contract

In the earlier Deuteronomy design [11], updates were sent to the DC as they were generated. If a transaction aborted, then undo operations were sent to the DC to compensate for the earlier updates. TC operations were latency-bound since they were issued directly to the DC. Further, the updates were sent to the DC before becoming stable in the recovery log (in fact updates were sent prior to being on the recovery log at all). An end-of-stable-log control operation (EOSL) informed the DC when operations up to a given LSN could be made stable to enforce the write-ahead log (WAL) protocol.

With the new design, all updates are in the stable log at the time that they are sent to the TC Proxy. It is safe for the TC Proxy to apply any committed operation to the DC without delay. The EOSL conveys to the DC that it is allowed to write updates to secondary storage because it knows that the WAL protocol has been enforced.

The DC can also use the EOSL to determine when it has seen all operations up to some LSN. This permits it, for instance, to re-organize and optimize its storage layout to simplify idempotence checking.

4.3 Applying Operations at the DC

4.3.1 Delayed Application

Good performance depends upon processing log records as close to a single time as possible. Our pure redo design requires that only

durably committed operations be submitted to the DC. Ideally, the TC Proxy would only encounter records of committed transactions when it processes a recovery log buffer. Then, all operations could be applied as they were encountered. When the scan of the buffer completed, the buffer could be marked as clean and reused.

To make that ideal scenario "almost true", we delay update processing of buffers. Figure 4 depicts this process. When the TC Proxy is remote, it queues multiple buffers, but it immediately scans arriving buffers for commit records and updates its version of the transaction table to indicate what transactions have committed (indicated by encountering their commit records). When the TC is collocated with the DC, this scan is not necessary since the TC Proxy can reference the TC's transaction table. During processing the TC stores the LSNs of commit records for transactions in the transaction table as they commit volatilely. In either case (remote or local DC), a "global" high water mark is maintained denoting the largest stable LSN on the recovery log. The combination of commit record LSN in the transaction table and high water mark tells us when a transaction is durably committed.

In the delayed operation scan, all operations of transactions known to be committed are applied. Operations of transactions known to be aborted are discarded. Operations of transactions whose outcomes are not known are relocated into a side buffer. The result is that at the end of the operation scan, the entire recovery log buffer is cleaned and can be reused.

This strategy works well when very few operations have undecided outcomes. Delaying operation installation to the DC minimizes the number of operations that must be relocated into a side buffer. The side buffer space requirement for storing the undecided transaction operations is very modest. Side buffer operations are applied to the DC once they are determined to be committed, and discarded when they are determined to be aborted.

4.3.2 Blind Writes

The TC Proxy is actually executing a form of redo recovery. In that situation, it only needs to apply the update. There is no need to inspect the prior state of a record to generate a pre-image. Because of this, the TC Proxy uses "upsert" operations in applying operations at the DC. An upsert has the same effect regardless of the prior state of the record, so no read of the prior state is needed.

We changed our Bw-tree implementation to permit it to service upserts. This is an important optimization. Not needing a prior read means that we can simply prepend a delta update to a Bw-tree page without accessing the rest of the page. One implication of this is that we do not even need to have a page fully in cache to perform the upsert. We can prepend the delta update to what we call a page stub containing just enough information to allow us to identify that we are updating the correct page (e.g. the boundary keys contain the key being updated).

Should multiple upserts add multiple versions of a single record to a page, an idempotence test will permit us to identify the correct version (only the latest). We delay the execution of this idempotence test until we need to read the page or the record or until we want to consolidate the page. Until then, we can permit multiple versions of a record to exist as delta updates.

4.4 Tracking DC Progress

Over time, MVCC version entries must be garbage collected from the MVCC hash table. The MVCC uses the transaction table to track the oldest active transaction's timestamp (OAT) and only retain version entries that remain visible to all ongoing and future transactions. This prevents the MVCC from removing entries needed to correctly perform concurrency control for the active transactions.

However, the MVCC must be careful; dropping version entries that have not been applied at the DC may result in the TC reading incorrect versions. For example, if a read is performed on a key for which the MVCC has no entry, it forwards the read to the DC. If the TC wrote a newer version for that key and the MVCC has "forgotten" it, then when the read is issued to the DC the wrong version could be returned.

To prevent this, MVCC entry garbage collection must be aware of the progress of the TC Proxy in applying updates to the DC. The TC Proxy provides a concise two-integer summary to the TC for this purpose (described below). Thus, the TC retains a read and write timestamp for a record version until it knows that the update has been applied at the DC.

The TC Proxy could use a single LSN to track the earliest unapplied version in the log in reporting progress to the TC. Unfortunately, this would allow long running transactions to stall MVCC garbage collection. Instead, the TC Proxy uses a pair <T-LSN, O-LSN> that is maintained in the two separate TC Proxy passes. The T-LSN tracks the progress of the transaction scan in identifying transactions whose commit records have been encountered. The O-LSN tracks the prefix of the log that has been applied at the DC, excluding operations belonging to transactions with a commit record LSN greater than the T-LSN. No transaction can hold back the O-LSN or T-LSN even if it is long running. An MVCC version can be safely discarded (case 2 of Section 2.4) if its LSN ≤ O-LSN and its transaction's commit record LSN ≤ T-LSN.

5. SUPPORTING MECHANISMS

A high performance system needs a global design that provides a framework that enables the performance. It also needs careful design and implementation at every level. In this section, we describe some of the technological innovations that we have used to achieve great performance.

5.1 Latch-free Buffer Management

We described latch-free buffer management in prior papers [12, 13]. There are common elements to what we do in our TC with the recovery log. Unlike the prior work, which was based on compare-and-swap, buffer reservation now leverages atomic-add instructions.

As with our prior technique, we maintain a one word OFFSET to where the next storage within a buffer is to be allocated. OFFSET is initialized to zero when the buffer is empty. When allocating space, we execute an atomic-add instruction to add the SIZE of the allocation to the OFFSET. The atomic-add returns the value of OFFSET as it was modified; the requestor uses this as the end of his allocated space and subtracts SIZE to determine the start location for the allocation.

There are no losers with atomic-add: it always succeeds and returns a different result (and hence a different region of the buffer) to each space requestor. If two atomic-adds are concurrent, one of them returns the sum of the two sizes. Subtracting each's SIZE correctly identifies the different starts of the reserved spaces for each; and further, it leaves the OFFSET pointing to the remaining unallocated space.

In our experiments, using atomic-add improves performance and scalability. Atomic-add avoids extra cache coherence traffic that compare-and-swap suffers under contention by reducing the conflict window. Further, atomic-add immediately acquires a cache line as modified, while the load for the pre-image preceding a

compare-and-swap may also first fetch the cache line in shared mode.

We also need to track the number of active users of the buffer. We do this by including a USERS field as the high order 32 bits of the OFFSET field. Both fields are modified atomically with a single add by using an addend of 2^{32} + SIZE so that an allocation request both reserves space in the buffer and increases the number of users as recorded in the USERS field. When finished populating the reserved space, the USERS field is decremented using atomic-add of (-2^{32}).

We need to cleanly terminate this when the buffer fills and "seal" it so that others will see that it should no longer be used and, thus, shift their attention to the next buffer. Further, when the buffer is filled, it needs to be written to secondary storage.

We determine whether a buffer is sealed by whether the offset returned from an atomic add is larger than the buffer extent. At that point, the OFFSET is also larger than the buffer extent and serves to seal the buffer. When that happens and the USERS field has dropped to zero, the buffer is both full and ready to be written.

To ensure that only one thread schedules the buffer write, we give that responsibility to the thread whose requested reservation straddled the end of the buffer. That is, the responsibility belongs to the sole thread whose atomic-add operation returned an offset beyond buffer end, but whose begin offset was within the buffer extent. To preserve lock-free discipline, any thread concurrently attempting a reservation on a buffer whose offset is beyond its extent attempts to atomically set a fresh buffer as the active log buffer.

5.2 Epoch Mechanism and Memory Reuse

Lock-free data structures pose challenges for memory reclamation. When an item is unlinked from a lock-free data structure, some threads may still be accessing the item. To make this safe, the TC uses an epoch-based mechanism to provide *pointer stability;* it posts unlinked items to a *garbage list* where they remain until no thread can reference them again, at which point they can be reused.

The basic idea of epoch-based memory management is that before each operation (for example, a read or update) a thread joins the current epoch E; this is usually done by incrementing E's membership count. If a thread frees a memory block M during its operation (for example, it unlinks an MVCC item), it places a pointer to M on E's *garbage list*. Memory on E's garbage list is not reclaimed until (a) E is no longer the current epoch and (b) E's membership count is zero; this is sufficient to ensure that no other thread can possibly dereference memory on E's garbage list.

In previous work, we described how to implement an epoch mechanism using two epochs [12, 13]. Its major performance issue was that the epoch membership counter was a hotspot, especially on multi-socket NUMA architectures. It required an atomic fetch-and-increment (and decrement) to the counter before and after every operation. Our new epoch design avoids such a hot spot.

The new epoch protection consists of a monotonically increasing global epoch (an unsigned 64-bit integer), a set of thread-local epochs (aligned on separate cache lines), and a garbage list where each item is held until it is safe to reuse its memory.

Whenever a thread starts a TC operation like read, write, or commit, it copies the global epoch into its slot in the thread-local epoch set. After completing an operation this thread-local epoch is set to ∞. Each thread-local epoch indicates to reclamation operations "when" the thread entered the TC and what it might have observed.

When an item is unlinked from one of the TC's internal data structures, the global epoch is copied into a garbage list entry along with a pointer to the unlinked item. Each time an entry is added to the garbage list, an old entry's item is removed for reuse. The garbage list is a fixed-size ring: when a new item is inserted at the head of the ring, the old item is removed and deallocated (if it is safe to do so).

It is safe to deallocate a garbage list item when the epoch stored with its entry is less than the minimum epoch found in the thread-local epoch set. If the item's epoch is smaller, then no thread has a reference to the item (the item must have been unlinked before any of the threads in the thread-local set entered the TC). Recomputing this minimum epoch for each item to be deallocated would be prohibitively slow. Instead, the minimum thread-local epoch is recomputed whenever the global epoch is incremented or if items in the list cannot be reclaimed because of low thread-local epochs.

The global epoch is incremented with an atomic increment only *periodically* whenever the garbage list has accumulated a significant number of new items that need to be returned to the allocator. This is the only atomic operation in the epoch manager, and the only operation that modifies a hot shared location. We currently increment the global epoch when the fixed-size garbage list has overwritten a quarter of its capacity; the minimum thread-local epoch is recomputed and cached at the same time.

Epoch protection is a tiny fraction of the total cost of each TC operation (less than 0.1% of wall time). Three things make it extremely fast. First, there are no locked, atomic, contended, or uncached operations on the fast path. Threads entering and exiting the TC only consult the (slowly changing) global epoch and update a single word in an exclusively owned cache line. Second, TC operations are short and non-blocking, enabling protection of all TC data structures with a single epoch enter/exit call pair per TC operation. This makes programming within the TC easier: all TC operations can safely hold and use references to any item in any data structure for the entire duration of the operation. Finally, the overhead to determine the minimum thread-local epoch is amortized over hundreds-of-thousands of unlink operations.

5.3 Latch-free Memory Allocation

Allocations for version storage are always handled using the latch-free recovery log buffers and read cache. However, managing MVCC hash table record and version entries and record keys is also allocation intensive and can limit TC performance.

These allocations are harder to deal with in a "log structured" way. Instead we use a general-purpose latch-free allocator for these data structures and other less common allocations. We use Hekaton's lock-free memory allocator, which in turn is based on Michael's allocator [20]. It uses per-core heaps and allocates from a small set of size classes up to 8 KB. Larger allocations are handled with Windows' low-fragmentation heap.

5.4 Thread Management

Threads must be carefully managed to achieve high performance on modern multi-core processors. Indeed, this requires the program to have some understanding of the "topology" of the cores as well. Consequently, peak performance cannot be obtained by blindly running across more cores despite Deuteronomy's latch-freedom. Work must be carefully partitioned as the system scales beyond a single CPU socket. At the same time, our goal of building a general-purpose and easy-to-use system deters us from using data

partitioning. We want the best performance possible even with workloads that do not partition cleanly by data.

We limit the number of threads and assign work to them as if they were cores. This keeps us from over scheduling work and allows us to control thread placement relative to sockets. Deuteronomy scales by splitting TC and TC Proxy/DC operations onto separate sockets when load exceeds what can be handled within a single socket. This allows it to leverage more cores and reduces cache pressure on both modules at the cost of increasing cache coherence traffic between them. We also make the best of the processor interconnect topology by minimizing communication between processor pairs with no direct connection.

At full load, the DC consists of a thread pool of up to 16 additional threads pinned to a single socket; the precise number of threads adapts to try to ensure that the DC applies operations as quickly as they are logged by the TC. To the extent possible, TC transaction processing occurs on hardware threads from adjacent sockets first, only spilling onto a socket that is two "hops" from the DC when necessary.

5.5 Asynchronous Programming

Deuteronomy uses a specialized asynchronous programming pattern to give it tight control over scheduling, to avoid thread context switch overheads, and to avoid holding indeterminately large stack space for each blocked thread. This programming pattern is used extensively within Microsoft to limit memory footprint in multi-job system settings.

This asynchronous pattern requires the use of a "continuation" whenever an asynchronous event (an I/O, for example) may make immediate and synchronous completion of a request impossible. In the naïve case, a continuation consists of the entire "paused" stack and CPU register set. With the asynchronous pattern, an explicit continuation must be provided. The stack is then "unwound" by programs returning up the call chain indicating that the call thread has "gone asynchronous".

Deuteronomy continuations usually require only the re-execution of a request. But each system level that requires re-execution needs to add its task state to the set that constitutes the full continuation. And continuations need to be on the heap when needed, as the return path will have folded up the stack allocations. The conventional way this is handled is to provide a heap-based continuation upon entering a code module. We optimize this asynchronous pattern to avoid this expensive heap allocation of context state in the common case.

Deuteronomy, unlike a naïve use of asynchronous continuation posting, stores its continuations on the stack, not the heap. When the execution of an operation can be done synchronously all heap allocation is avoided; this is the common case, since often all needed data is cached. Only when the asynchronous path is required (for example, if the operation needs to perform an I/O operation to bring needed data into the cache) are the continuations copied from the stack to the heap. This simple technique is fundamental to Deuteronomy's high performance: it means that for the vast majority of operations, the overhead of heap allocation for task state is avoided completely. This is especially essential for read operations that hit in memory, for which no heap allocation operations occur at all.

6. EVALUATION

Our goal is to show Deuteronomy's performance is competitive with modern, monolithically architected databases on realistic workloads. Indeed, even further, we hope to show that

OS	Windows® Server 2012
CPUs	4× Intel® Xeon® E5-4650L 32 total cores 64 total hardware threads
Memory	192 GB DDR3-1600 SDRAM
Storage	320 GB Flash SSD Effective read/write: 430/440 MB/s Effective read/write IOPS: 91,000/41,000

Table 1: Details of the system used for experiments.

Deuteronomy is competitive with main memory databases when the working set of an application fits in Deuteronomy main memory.

We used our previously built DC (Bw-tree and LLAMA, evaluated elsewhere [12, 13]) in the evaluation of our TC. The Bw-tree-based DC is deployed in existing Microsoft products including Hekaton [2] and DocumentDB [29].

Our experimental machine set up is described in Table 1. Each of its four CPUs reside in separate NUMA nodes, which are organized as a ring.

6.1 Experimental Workload

For these experiments we use workloads similar to the YCSB benchmarks. The DC database is preloaded with 50 million 100-byte values. Client threads read and update values with keys chosen randomly using a Zipfian distribution with skew chosen such that 20% of the keys receive 80% of the accesses. Transactions are created by grouping operations into sets of four. Recovery logging and DC updates share a single commodity SSD.

Overall, this workload stresses the TC with a large, fine-grained, and high-churn dataset. Most realistic cloud deployments would exhibit stronger skew and temporal locality due to historical data. For example, assuming throughput of 1.5 M transactions/s (which the TC can sustain for this workload) Figure 5 shows that in less than 20 seconds the majority of the records in the database have been read or written (nearly 2.5 GB of the initial set of records, and this does not count the additional versions generated due to updates). Likewise, the 100-byte record size represents a worst-case for the TC: for every version newer than the oldest active transaction the MVCC uses more space for a hash table entry than it does for the space to store the version contents. Thus, these experiments show the TC performance under an exceptionally difficult workload.

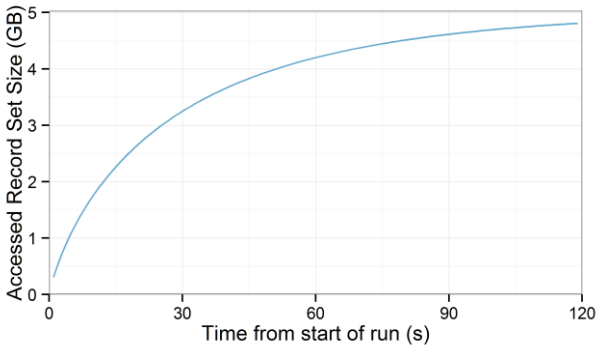

Figure 5: For our experimental workload of about 5 GB the TC accesses more than 2.5 GB of the records every 20 seconds. It accesses nearly all of the records every two minutes.

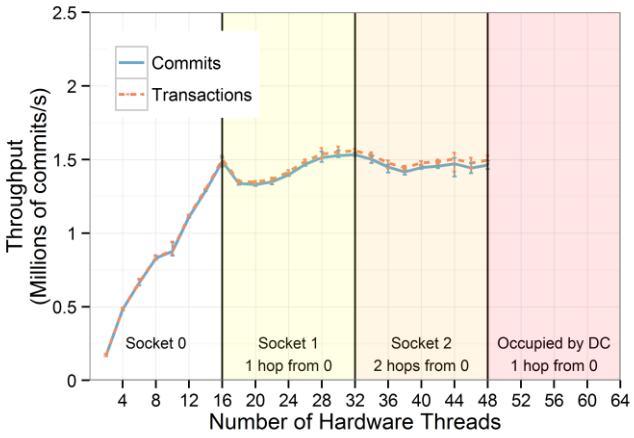

Figure 6: Stable-state performance as the TC expands to execute transactions across more hardware threads.

6.2 Scalability

Figure 6 shows Deuteronomy's performance as the TC is scaled across multiple hardware threads and sockets (each core hosts two hardware threads). The combined TC Proxy and DC use up to 16 additional hardware threads on the last socket (not included in the count on the axis). Points are averaged over 5 runs; error bars show the min/max of the 5 runs (variance is insignificant for most points).

Overall, 84% of the operations issued are independently and randomly chosen to be read operations; this results in about 50% of transactions being read-only with the other half issuing one or more writes. With 32 threads (two sockets) issuing transactions, Deuteronomy sustains 1.5 million transactions per second in steady state (6 million total operations per second) with an insignificant abort rate. Performance scales well within a single socket, and it shows small improvement running across two sockets. It levels off once all of the machine's NUMA groups are involved.

Overall, performance is limited by DRAM latency, primarily MVCC table accesses, read cache and log buffer accesses, and Bw-tree traversal and data page accesses. The large working set hampers the effectiveness of CPU caches and the DTLBs. Large or relatively fixed-size structures like the recovery log buffers, the read cache, and the transaction table are all backed by 1 GB super-pages to reduce TLB pressure.

6.3 Read-write Ratio

Figure 7 shows how the read-write mix of the workload impacts throughput. Read-intensive workloads stress MVCC table performance and the read cache, while write-intensive workloads put pressure on recovery log buffer allocation, I/O, and TC Proxy/DC updates. For common read-intensive data center workloads Deuteronomy provides between 1 and 2 million transactions per second (4 to 8 million operations per second). For more heavily skewed and more nearly read-only workloads than those shown above, the TC can sustain around 14 million operations per second.

For workloads with less than about 70% reads all of the cores of the TC Proxy/DC are busy updating records. Ideally, I/O bandwidth would be the limiting factor for write-intensive workloads rather than software overheads. In our write-only workload, our SSD sustains 390 MB/s: 89% of its peak write bandwidth. DC updates occasionally stall waiting for I/O, so adding additional write bandwidth capacity may still yield increased performance.

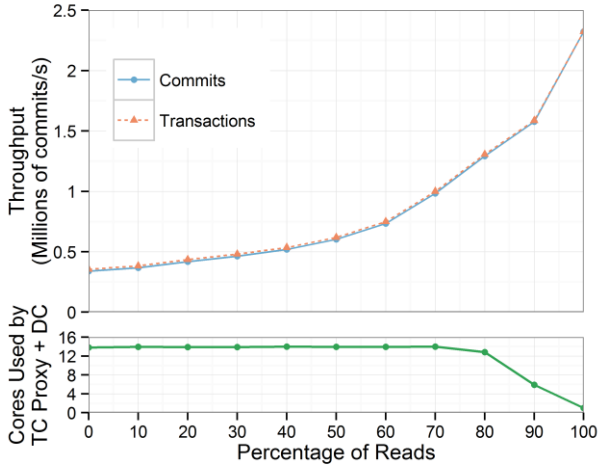

Figure 7: The impact of workload read/write mix on performance and DC core utilization.

6.4 Caching Effectiveness & TC Latency

Figure 8 shows a cumulative distribution of TC and DC operation latencies which illustrates three key points. First, TC caching is effective. Versions for TC read operations are serviced from the combined log buffers and read cache 92% of the time (this can be seen as the "knee" in the TC Read line on the left graph). Reads are on the critical path, so the resulting 4× improvement in mean read latency (compare the mean TC Read time of 2.4 µs to the mean DC Get time of 11 µs) translates almost directly into 4× improved transaction throughput since most of the increased latency is added instruction path.

Second, TC Reads are limited by DC Get performance on cache misses. Thus, the line for DC Get operations also appears (compressed and shifted) as part of the tail of the TC Read and TC Update lines, since some TC Read and TC Update operations resort to DC Gets. However, DC Upsert latency never appears as part of TC operation lines, since upserts are performed as a background activity by the TC Proxy, outside of any operation's execution (latency) path.

Third, the right graph of Figure 8 shows that MVCC garbage collection only impacts a small fraction of TC response latencies. The long tails of the "TC Read" and "TC Update" lines show that less than 0.1% of operations are delayed by garbage collection. This can be seen as the nearly 100 µs-long "shelf" in the TC Read and TC Update lines. The graph also shows that updates are delayed by garbage collection more frequently than reads; this is because each update issues multiple MVCC table operations, giving it twice as many opportunities to be recruited to do garbage collection (§6.5 explores garbage collection in more detail).

Overall, Figure 8 shows that overhead on the fast (and common) path is extremely low. The MVCC table's dual purpose minimizes redundant work on the fast-path: the TC can track MVCC hash table entries and find the corresponding versions with a single hashed lookup on one data structure. This puts TC fast-path operations on par with the internal lookup performance of other highly latency-focused non-transactional in-memory storage

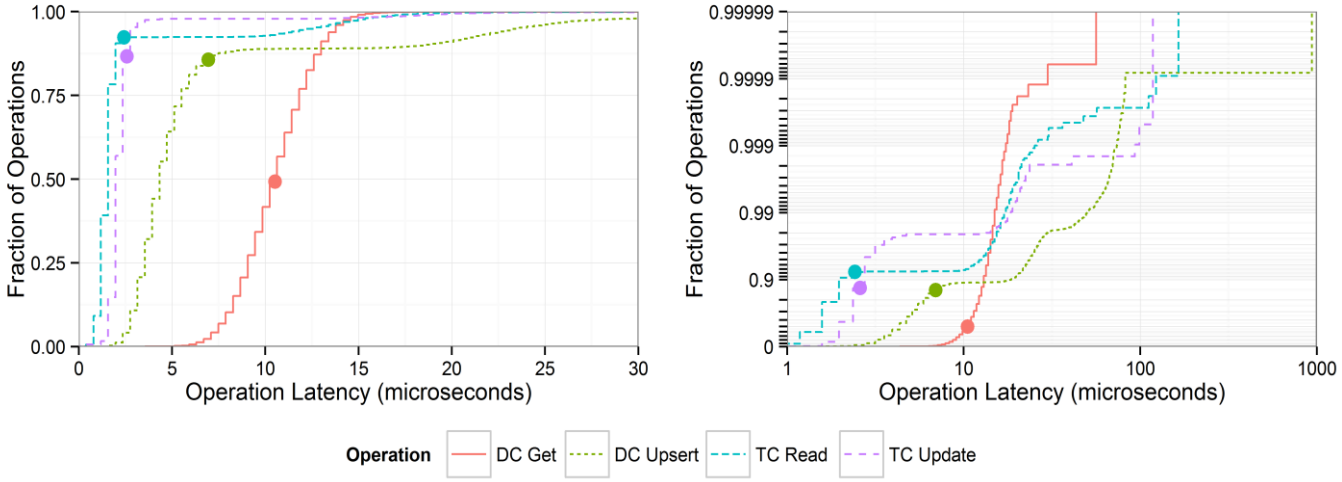

Figure 8: Cumulative distribution of TC and DC operation latencies. The mean of each set of samples is shown as a point. The right graph represents the same data against a log scale to highlight tail latency. The stair step effect in the left graph and on the left side of the right graph are due to low clock resolution (about 400 ns) rather than a low number of samples.

systems like RAMCloud [25] even while Deuteronomy does additional work for concurrency control.

6.5 MVCC Version Garbage Collection

Over time the MVCC table accumulates information about versions and grows large. In steady-state operation, the MVCC table must reclaim space by removing versions that can never be referenced again and records that are unlikely to be referenced again soon.

To do this, the MVCC table is configured with two "watermarks;" if the MVCC table size reaches these watermarks, threads issuing TC operations are co-opted into performing garbage collection operations on the table. At the soft limit, threads begin evicting older version entries from records they access. At the hard limit, threads begin scanning portions of the MVCC table to evict version and record table entries for colder records. Also, no new operations are permitted until MVCC table size is brought back under the hard limit. If table size cannot be controlled, then the TC begins to shed load by aborting transactions. Ideally, the TC should be configured so that this isn't necessary.

Figure 9 explores the impact of MVCC version garbage collection on transaction throughput and MVCC table size. Each of the three graphs tracks a different TC metric over the first 5 minutes of its life after it starts up. The top graph shows transaction throughput over time from the start of the TC until it reaches steady state while it runs the same experimental workload as in Section 6.2. The middle graph tracks the MVCC table size relative to its configured soft and hard watermarks, and the bottom graph shows what fraction of the hardware threads executing transactions goes into performing garbage collection work.

After 100 seconds the MVCC table size grows beyond the soft limit, and threads begin discarding old versions associated with the records they are accessing. This curbs MVCC table growth somewhat, but once the version lists for the hot records in the table have been trimmed the table begins to grow again. At about 160 seconds, the table hits the hard limit and threads begin scanning and evicting not only old versions, but also whole record information for cold records.

Overall, garbage collection operations grows to consume about 4% in total of the 16 hardware threads dedicated to running transactions while holding MVCC table size steady. Steady state transaction throughput drops by about 20% from 1.6 down to 1.3 million transactions per second.

The impact of MVCC table garbage collection on CPU utilization and transaction throughput is highly (and non-linearly) dependent on the choice of thresholds and the working set size of the workload. Higher limits defer garbage collection longer, making it more likely garbage collection will be able to reclaim space quickly without needing to pace normal operations. On this workload, for example, a 1 GB hard limit only decreases performance to 1.0 million transactions per second, but garbage collection operations continuously consume 50% of the TC CPU's socket.

In the future, automatically setting the soft and hard limits will be important for real-world deployment. MVCC table space competes for DRAM with other applications as well as in-memory log buffer and read cache space, which can be used to reduce version access latency. One idea is to create a feedback loop that increases the limits when transaction throughput is suffering and cycles spent in garbage collection is high and tightens limits when cache misses appear to be limiting performance.

6.6 Performance Impact of Checkpointing

Periodically, the TC induces a DC checkpoint operation to ensure that the DC has durably written all applied operations up to some LSN. The TC uses this so that it can truncate the recovery log and bound recovery time. To enable this, the TC Proxy tracks the lowest LSN such that all prior operations are applied at the DC. It sends this to the DC occasionally as a Deuteronomy "end of stable log" (EOSL) control operation that permits the DC to make stable operations with lower LSNs.

From time to time, the TC Proxy executes a Deuteronomy "redo scan start point" (RSSP) operation to request a checkpoint that forces the DC to make stable all operations up to and including the RSSP LSN stable. After the checkpoint completes, the TC is free to truncate the recovery log up to the RSSP LSN; the log earlier than RSSP only contains operations whose transaction outcome is known and, if committed, the operations have been applied stably at the DC. In our experiments, the DC completes a checkpoint about every 45 seconds; this also serves as a rough bound on recovery time.

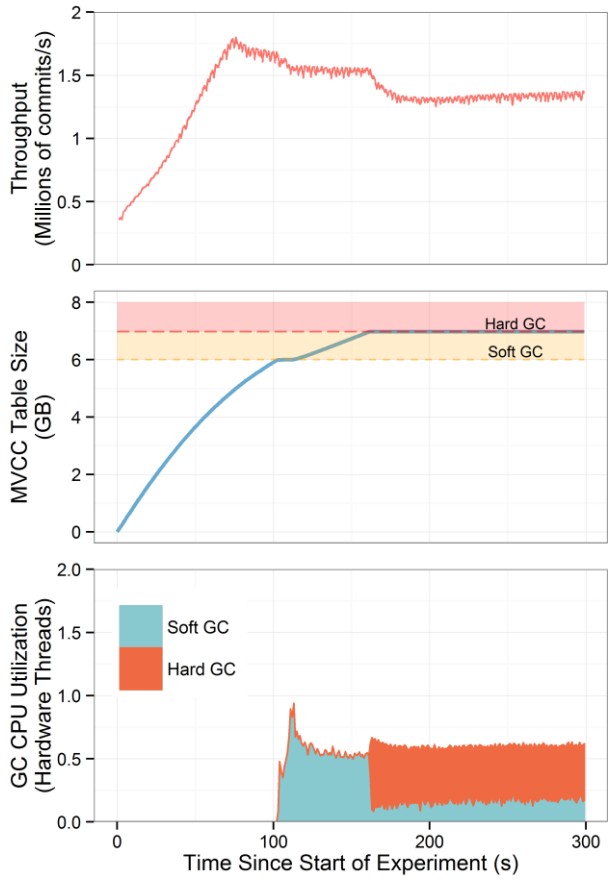

Figure 9: MVCC version garbage collection slows steady-state transaction throughput by about 20%. The top graph shows TC transaction throughput. The middle shows MVCC table size relative to its soft and hard garbage collection watermarks. The bottom shows CPU utilization due to MVCC table garbage collection.

Checkpointing runs on a single thread bound to the same socket as the TC Proxy and DC. Overall, it doesn't have a statistically significant impact on performance except under heavy write loads (it contends for storage bandwidth). However, even for a 100% write workload throughput only decreases by 8%.

All experiments in this paper include checkpointing overhead except for Figure 9, where the attempt is to isolate MVCC garbage collection overhead (though, even for that figure checkpointing made no measurable difference).

7. RELATED WORK

Database kernel architecture. The database kernel has traditionally been treated as a monolithic module, intertwining transactional concurrency control with data caching and storage [4]. Our work shows that great performance is possible when decomposing a database kernel into a transactional component (for concurrency control and recovery) and data component (for data storage).

High performance transactional and storage engines. There has been a flurry of research and development of novel main-memory database systems. Examples include Hyper [6], H-Store/VoltDB [5, 24], Microsoft Hekaton [2], Oracle TimesTen [7], IBM SolidDB [15], and SAP HANA [9]. MICA [14], Masstree [18], and RAMCloud [25] are fast main-memory key value storage engines. As key-value stores, these engines could be made to work as a DC in the Deuteronomy architecture akin to the Bw-tree. Silo [27] is a main-memory database built atop Masstree. The Silo design follows a traditional monolithic architecture that couples concurrency control with access methods. In contrast, Deuteronomy decouples transactional concurrency control from data storage while providing comparable performance for working sets that fit in memory. Nonetheless, in contrast to the aforementioned systems, Deuteronomy is not a main-memory transactional engine where all records are assumed to "live" in memory. Rather, data lives on secondary storage and is only cached in main memory.

Concurrency control. Classic database architectures use a pessimistic concurrency control scheme based on locking [3]. Recently a number of optimistic concurrency control schemes have been revisited in the context of main-memory optimized systems. Hekaton uses an optimistic multi-versioned concurrency control technique [8]. H-Store/VoltDB uses a partitioned model where all threads execute serially on a partition. HyPer has evaluated the use of timestamp ordering (TO) [10, 21].

Our TC design uses TO as its concurrency control scheme. Our architecture stores record TO entries in memory. However, these TO entries may reference record versions on secondary storage (either the redo log or in the DC).

Recovery. Most classically architected databases model recovery on the ARIES protocol [22]. However, most high-performance transactions engines diverge from ARIES. For example, Hekaton logs updates and rebuilds in memory tables from scratch after a crash. VoltDB recovers by using coarse-grained command logging and replays transactions from a consistent checkpoint [19]. Our recovery scheme also diverges from traditional ARIES recovery. We use logical redo-only logging and only apply committed record updates at the DC.

Log structuring. We make heavy use of log structured storage techniques throughput the TC similar to those first proposed for use in file systems [24]. Our TC performs log-structured cleaning of in-memory redo log buffers. The TC read cache is also log structured even though it resides entirely in memory; this allows fast allocation and efficient space utilization. Similar benefits were observed with RAMCloud's in memory logging [25]. As mentioned earlier, MICA caches in a log-structured ring with many similarities [14]. In previous work we described our log-structured storage techniques used in our DC implementation [12, 13] that reduce write amplification by writing deltas instead of whole pages.

8. DISCUSSION

8.1 Status of Prototype

We planned a staged implementation of for our new transactional component. Our first focus, and what we report on in this paper, is to provide transactional support for the usual key-value store CRUD operations. Our goal was to provide 100× the performance of our original TC. We have succeeded in this, even while providing full transactional durability.

Part of this performance is due to the data component we are using, which is based on the Bw-tree and LLAMA that together provide an atomic record store. This atomic record store is a product quality implementation that is being used currently in a number of Microsoft products.

8.2 Limitations and Plans

We have focused on serializable transactions for our TC implementation. And, for the set of operations supported, that is what we implement. However, we do not yet support transactional key range operations. This support is needed to fully realize our serializability goal, and we are in the midst of implementing this capability.

A number of experimental results over different workload mixes reinforces our belief that we need to augment timestamp order concurrency control. We expect that this will (at least some of the time) require commit time validation. But we worry about transactions with higher latency, especially in a high contention setting. Our expectation is that this enhancement will enable many transactions that currently might abort using pure timestamp order concurrency control to successfully commit.

Both of these work items will add overhead to TC execution paths, and, as always, our goal will be to minimize this so as to achieve very high performance. We have high confidence that overhead for both these capabilities can be offset by further tuning of our existing code paths, and/or more astute thread management.

8.3 Conclusions

We believe the importance of the work reported here is that it points to new ways of realizing stateful transactional resource managers. The Deuteronomy architectural framework enables such resource managers to be composed with reusable components, each component providing useful functionality in and of itself. Our latch-free and log structured implementation approach enables the construction of these components with very high performance. This combination of architectural flexibility and great performance has already found a warm reception within Microsoft products. Our Deuteronomy effort brings main memory database-level performance to a system where data lives on secondary storage and is only cached in main memory. This is a unique Deuteronomy capability.

9. REFERENCES

[1] D. Dewitt et al. Implementation Techniques for Main Memory Database Systems. In SIGMOD, 1984, pp. 1-8.

[2] C. Diaconu et al. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In SIGMOD, 2013, pp. 1243-1254.

[3] J. Gray, R. A. Lorie, G. R. Putzolu, I. L. Traiger. Granularity of Locks in a Shared Data Base. In VLDB, 1975, pp. 428-451.

[4] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a Database System. Foundations and Trends in Databases. 1(2) pp. 141-259, 2007.

[5] R. Kallman et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. PVLDB 1(2): 1496-1499, 2008.

[6] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System based on Virtual Memory Snapshots. In ICDE, 2011, pp. 195-206.

[7] T. Lahiri, M-A. Neimat, S. Folkman. Oracle TimesTen: An In-Memory Database for Enterprise Applications. IEEE Data Eng. Bulletin 36(2): 6-13, 2013.

[8] P-A Larson et al: High-Performance Concurrency Control Mechanisms for Main-Memory Databases. PVLDB 5(4): 298-309, 2011.

[9] J. Lee et al. High-Performance Transaction Processing in SAP HANA. Data Eng. Bulletin 36(2): 28-33, 2013.

[10] V. Leis, A. Kemper, and T. Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In ICDE, 2014, pp. 580-591.

[11] J. Levandoski, D. Lomet, M. Mokbel, and K. Zhao. Deuteronomy: Transaction Support for Cloud Data. In CIDR, 2011, pp. 123–133.

[12] J. Levandoski, D. Lomet, S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. PVLDB 6(10): 877-888, 2013.

[13] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In ICDE 2013: 302-313.

[14] H. Lim, D. Han, D. G. Andersen, M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Stroage. In NSDI, 2014, pp. 429-444.

[15] J. Lindström et al. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. IEEE Data Eng. Bulletin 36(2): 14-20, 2013.

[16] D. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-Version Concurrency via Timestamp Range Conflict Management. In ICDE, 2012, pp. 714-725.

[17] D. Lomet, A. Fekete, G. Weikum, M. Zwilling. Unbundling Transaction Services in the Cloud. In CIDR, 2009: 123–133.

[18] Y. Mao, E. Kohler, R. T. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In EuroSys, 2012, pp. 183-196.

[19] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking Main Memory OLTP Recovery. In ICDE, 2014, pp. 604-615.

[20] M. Michael. Scalable Lock-Free Dynamic Memory Allocation. In PLDI, 2004, pp. 35-46.

[21] H. Mühe, S. Wolf, A. Kemper, and T. Neumann: An Evaluation of Strict Timestamp Ordering Concurrency Control for Main-Memory Database Systems. In IMDM Workshop, 2013, 74-85.

[22] C. Mohan, et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM TODS 17(1): 94-162, 1992.

[23] D. P. Reed. Naming and Synchronization in a Decentralized Computer System. Ph.D. dissertations, Dept. or Electrical Engineering, M.I.T., Cambridge, MA, Sept. 1978.

[24] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. ACM TOCS 10(1): 26–52, 1992.

[25] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for DRAM-based storage. In FAST, 2014, pp. 1-16.

[26] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. IEEE Data Eng. Bulletin 36(2): 21-27, 2013.

[27] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-Memory Databases. In SOSP, 2013, pp. 18-32.

[28] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast Databases with Fast Durability and Recovery through Multicore Parallelism. In OSDI, 2014, pp. 465-478.

[29] Microsoft Azure DocumentDB: **`http://www.documentdb.com`**