RESEARCH-ARTICLE

Multiverse: Transactional Memory with Dynamic Multiversioning

Published: 28 January 2026

Citation in BibTeX format

PPoPP '26: 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming
January 31 - February 4, 2026
NSW, Sydney, Australia

Conference Sponsors:
SIGHPC
SIGPLAN

PPoPP '26: Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (January 2026)
https://doi.org/10.1145/3774934.3786436
ISBN: 9798400723100

Multiverse: Transactional Memory with Dynamic Multiversioning

Gaetano Coccimiglio
University of Waterloo
Waterloo, Canada
gccoccim@uwaterloo.ca

Trevor Brown
University of Waterloo
Waterloo, Canada
trevor.brown@uwaterloo.ca

Srivatsan Ravi
University of Southern California
Los Angeles, USA
srivatsr@usc.edu

Abstract

Software transactional memory (STM) allows programmers to easily implement concurrent data structures. STMs simplify atomicity. Recent STMs can achieve good performance for some workloads but they have some limitations. In particular, STMs typically cannot support long-running reads which access a large number of addresses that are frequently updated. Multiversioning is a common approach used to support this type of workload. However, multiversioning is often expensive and can reduce the performance of transactions where versioning is not necessary.

In this work we present Multiverse, a new STM that combines the best of both unversioned TM and multiversioning. Multiverse features versioned and unversioned transactions which can execute concurrently. A main goal of Multiverse is to ensure that unversioned transactions achieve performance comparable to the state of the art unversioned STM while still supporting fast versioned transactions needed to enable long running reads.

We implement Multiverse and compare it against several STMs. Our experiments demonstrate that Multiverse achieves comparable or better performance for common case workloads where there are no long running reads. For workloads with long running reads and frequent updates Multiverse significantly outperforms existing STMS. In several cases for these workloads the throughput of Multiverse is several orders of magnitude faster than other STMs.

CCS Concepts: • **Computing methodologies → Concurrent algorithms**.

Keywords: Transaction Memory, Software Transaction Memory, Multiversioning, Multiversion Concurrency Control

ACM Reference Format:
Gaetano Coccimiglio, Trevor Brown, and Srivatsan Ravi. 2026. Multiverse: Transactional Memory with Dynamic Multiversioning. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '26), January 31 – February 4, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3774934.3786436

PPoPP '26, Sydney, NSW, Australia

ACM ISBN 979-8-4007-2310-0/2026/01
https://doi.org/10.1145/3774934.3786436

1 Introduction

Software transactional memory (STM) is a synchronization mechanism that allows users to execute sequences of memory accesses as atomic *transactions*. STMs make atomicity easy but not necessarily fast. STMs typically perform read-only transactions using optimistic synchronization. In such STMs [12, 14–17, 19, 21, 25, 28], memory addresses are typically associated with version numbers that are used to determine whether a transaction's reads are consistent. If not, a transaction aborts and retries.

For workloads where transactions are small (accessing few addresses) and contention is low, TMs can typically achieve good performance. On the other hand, STMs often struggle to handle transactions that read a large number of addresses that are frequently updated. This type of transaction is very likely to repeatedly abort.

A classical solution in data structures and databases for supporting large read-only operations is to utilize *multiversioned concurrency control (MVCC)*, which allows one to take atomic *snapshots* of memory [1, 30]. Much of the early work on MVCC focused on maintaining consistent versions and avoiding an unbounded number of versions per address. Many of these MVCC designs which focus only on providing atomic snapshots are typically expensive. Recent work has made significant practical improvements to MVCC [5, 29].

There have been some attempts to combine multiversioning with STM [22, 23, 27]. These existing approaches typically guarantee the weaker correctness conditions of snapshot isolation [2] or serializability [3], as opposed to (the stronger) opacity [18]. Without opacity, aborted transactions are allowed to observe inconsistent state which can lead to various problems [11]. Furthermore, maintaining multiple versions can incur substantial overhead, reducing performance in the case where versioning is not required. In such cases, traditional *unversioned* STMs are preferable.

In this work we present Multiverse, a new opaque STM that leverages recent advancements in MVCC [5] to enable read-only transactions that access a large number of addresses to commit even in the presence of many concurrent updates that would otherwise cause potentially unbounded

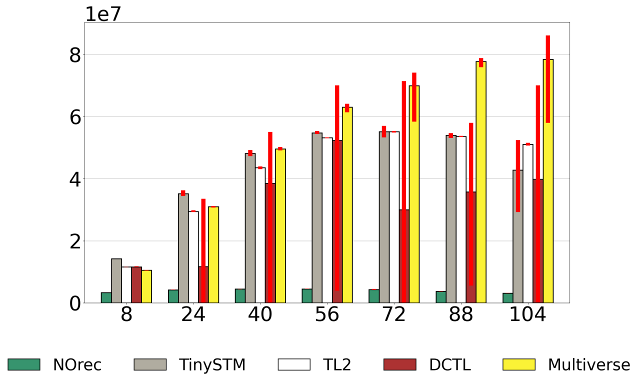


Figure 1. (a,b)-tree benchmark with an 89.99% search, 0.01% RQ, 5% insert, 5% delete workload using a uniform key access pattern. RQ size is 10k (1% of prefill size). Y-axis: ops/sec. X-axis: number of threads.

aborts. Unlike prior multiversioned STMs, our primary design goal is to ensure that *our unversioned transactions* are competitive with the *fastest unversioned STM*, Deferred Clock Transactional Locking (DCTL) at present [25], while also ensuring that our *versioned* transactions are fast as possible subject to the constraint that they do not add substantial overhead to unversioned transactions. As Figure 1 shows, Multiverse can outperform the fastest unversioned STM even for workloads with very few range queries (RQs), where versioning is not always necessary.

Our algorithm features unversioned and versioned transactional code paths. Transactions begin as unversioned, and transition to the versioned code path based on a heuristic function that considers the number of times the transaction has aborted, the thread's recent transactions' behaviour, and the recent behaviour of other transactions in the TM system. Our design associates versions with individual addresses at the word level of granularity. We dynamically switch addresses between versioned and unversioned states. Switching addresses between unversioned and versioned states is subtle and deeply connected to both correctness and performance.

A key insight in our work is that versioning of addresses should be done quite differently in different workloads. More specifically, if a versioned transaction only needs to access a small number of addresses under low contention, it is most efficient to leave most addresses unversioned, and allow concurrent non-versioned update transactions to proceed with as little overhead as possible. On the other hand, if a versioned transaction needs to access a very large number of addresses under high contention, it is more efficient to preemptively (and globally) force all concurrent updating transactions to preserve old versions of all addresses.

Our TM thus features two global modes that we switch between based on a heuristic. In *Mode Q*, transactions on the *versioned code path* are responsible for *marking addresses as versioned*, and transactions on the unversioned code path can be largely oblivious to versioned transactions. A versioned transaction in Mode Q that encounters an unversioned address will abort if the address has changed since the transaction began. So, this mode is suitable when transactions that require multiversioning only access relatively few memory addresses, and/or are infrequent.

If aborts due to encountering unversioned addresses are frequent, then Mode U can help substantially. In *Mode U*, transactions on the *unversioned* code path *that write to memory* are responsible for marking addresses as versioned, and transactions on the versioned code path can simply behave as if all relevant addresses are already versioned. Mode U is crucial for enabling high performance when versioned transactions perform a large number of accesses that are likely to be aborted by concurrent updates.

The complexities of versioning are completely hidden from the user, and Multiverse does not require any modifications to a program's memory layout—only replacement of variable types with analogous transactional types. This is the gold standard in TM. By avoiding changes to the memory layout, we preserve the cache behaviour of the underlying program as much as possible, limit intrusive changes to the program, and allow standard object serialization techniques for disk storage or network transfer of transactional objects. We utilize separate parallel lock- and version-tables so that when the versioning mechanism is not actively engaged, the cache behaviour of unversioned transactions is as similar as possible to the state of the art in unversioned STM.

Contributions:

- We introduce Multiverse, a novel opaque STM that combines unversioned STM and MVCC. It features a distinct usage of dynamic multiversioning and multiple TM modes that adapt the behaviour of the TM to fit the needs of the workload §3.
- To our knowledge, Multiverse is the first full featured opaque multiversioned STM implemented in C++ that has proper memory management which avoids crashes that would occur in other STMs like TL2 or DCTL §4.
- We implement Multiverse and experimentally evaluate it on a real system comparing it against several existing opaque STMs §5.

2 Background

2.1 Transactional Memory

A transaction is a sequence of transactional accesses, (reads and writes), performed on a set of *transactional addresses.* A transaction either *commits* and appears as a single indivisible step, or *aborts* and has no visible effect.

A TM implementation provides operations to start a transaction, read and write transactional addresses, commit a transaction, and voluntarily abort a transaction. If two transactions are concurrent and one or both of the transactions writes to an address that the other has already accessed then we say that these transactions conflict. Conflicts cause transactions to abort. A transaction that aborts due to a conflict will typically be retried until it either succeeds and commits, or until it is voluntarily aborted. We call the set of all addresses read by a transaction its *read set* and the set of all addresses written by a transaction its *write set*.

2.2 Correctness

Snapshot isolation (SI) [2], serializability, strict serializability [3] and the stronger opacity [18] are common correctness conditions used in TM. A history of *committed* transactions is serializable if it is equivalent to some serial history. Strict serializability further requires that the history maintains the real-time order of transactions. Opacity requires that the history of all transactions (including aborted ones) be equivalent to some sequential history. In other words, opacity requires that *all* transactions must observe consistent state. As discussed in [11], one should care about opacity since it prevents various problems. In particular, without opacity, one loses even single-threaded invariants since transactions could observe inconsistent state and continue running. The weaker SI, intuitively, allows transactions to perform (consistent) reads in the past, but write in the present, which can be difficult to use correctly. Opacity is most common in TM.

3 Algorithm

Multiverse is an opaque word based STM in which **both addresses and transactions** can either be *unversioned* or *versioned*. Transactions always begin as unversioned. Transactions that write remain unversioned. Unversioned read-only transactions will switch to versioned after some number of attempts or under certain conditions discussed in Section 4.

A core goal of Multiverse is that our unversioned transactions should match the performance of the fastest unversioned STM (DCTL), and our versioned transactions should be as fast as possible subject to the former. This goal motivated many of our design choices. By default, Multiverse prioritizes the performance of unversioned transactions. Similar to DCTL, the leading STM, we use a global clock and transactional addresses are protected by *versioned locks*. The data structures used in Multiverse are illustrated in Figure 2 and described below.

3.1 Word Based Dynamic Versioning

Addresses in Multiverse are initially unversioned, meaning they do not have associated version lists. An address can *become versioned* if we determine that maintaining additional versions is likely to reduce aborts. Likewise, if we later determine that we do not need additional versions of a particular address then we can *unversion* the address by removing and freeing (via epoch-based reclamation) its version list.

3.1.1 Versioning Addresses. Versioning an address requires creating and associating a version list with the address. Versioning and unversioning of an address is done while holding the associated address lock. This ensures no concurrent updates can modify the address.

To avoid changing a program's memory layout, we store all versioned locks and version lists in hash tables which we refer to as the *lock table* and *Version List Table (VLT)*. Each bucket in the VLT is a linked list. Within a bucket, each node contains (1) a pointer to the head of a version list, (2) the address for which the version list is tracking changes, and (3) a pointer to the next bucket node. The VLT and lock table are identical in size, which allows us to use the same mapping function from addresses to entries for both, and enables a convention that an address' lock also protects its version list.

When we create a new version list, we insert an initial version. This requires a timestamp and the data. For the data, we take the last consistent value of the address. Since we must hold the lock before versioning an address, the last consistent value is simply the current value of the address. Choosing the timestamp is more nuanced. A timestamp will correspond to some value of the global clock. We attempt to take the earliest possible timestamp (details in Section 4).

3.1.2 Checking if an Address is Versioned. Determining whether or not a particular address is versioned requires traversing the associated bucket in the VLT. We know that the address is versioned if we find a node in the bucket that has the address. We use bloom filters to make this efficient. Each address is associated with a bloom filter. When an address becomes versioned we add it to the bloom filter. To determine if an address is versioned we first check the bloom filter. If we do not find the address in the bloom filter we know the address is unversioned. Similar to the VLT, we store these bloom filters in a separate table of identical size.

3.1.3 Unversioning Addresses. We unversion entire VLT buckets rather than individual addresses. There are several reasons for this approach. First, one cannot remove items from a bloom filter—one can only reset it. Resetting the filter means all addresses that map to that VLT bucket are now unversioned. Doing this periodically is worthwhile since otherwise bloom filters will slowly fill up, and produce many false positives. Second, there is value in keeping the length of a VLT bucket small to reduce the overhead of traversals. Any "collateral damage" in unversioning can affect performance but not correctness.

Unversioning is performed by a background thread. Unversioning a VLT bucket requires removing and freeing the linked list in the bucket along with all of the version lists in the bucket. Before unversioning, the background thread must claim the associated lock. We determine when to unversion

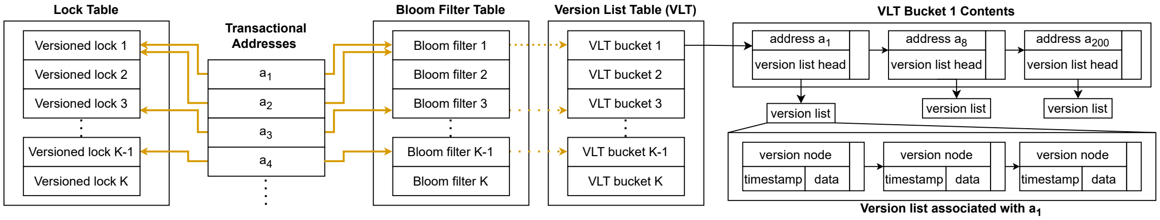


Figure 2. Data structures used in Multiverse. In this example addresses a_1 and a_2 map to the first VLT bucket but only a_1 is versioned. The orange arrows indicate a mapping while the black arrows indicate a memory pointer. The dotted arrow from the bloom filter to the VLT is indicating that we access the bloom filter first before the VLT.

	Mode Q	Mode QtoU (Transient)	Mode U	Mode UtoQ (Transient)
Unversioned txn	Writes add versions iff address is already versioned	Writes forced to version addresses	Writes forced to version addresses	Writes forced to version addresses
Versioned txn (read-only)	Reads version addresses	Reads version addresses	Reads assume all addresses are versioned	New or retried txns enter Mode Q instead
Background Thread	Unversioning enabled	Unversioning disabled	Unversioning disabled	Unversioning disabled

Table 1. Differences in the behaviour of transactions running in different modes. Note that a transaction's mode does not always match the TM's mode. For example, a versioned transaction in Mode UtoQ enters Mode Q on abort, and the TM can enter mode Q only once all such transactions are in Mode Q.

a bucket using a heuristic which considers the timestamps of version lists in the bucket as well as the current global clock version. We discuss this further in Section 4.4.

3.2 Transaction Paths Basic Overview

3.2.1 Unversioned Transactions. The basic execution of an unversioned transaction follows an approach similar to DCTL. At the start of each attempt, the transaction will read and record a local copy of the global clock to acquire its *read clock*. For each TM access of an address, the version of the lock protecting the address is validated against the transaction's read clock. TM writes use encounter time locking and writing. If the address being written to is versioned, the write *also* updates its version list (keeping *both* the version list, and the location where an unversioned transaction would read, up to date). Any modifications to version lists are marked to-be-determined (TBD) until the transaction commits, which prevents versioned transactions from seeing inconsistent states. Unversioned transactions keep track of a read set and two write sets. The *standard write set* tracks *all* addresses written by the transaction and the *versioned write set* tracks only the *versioned* ones (so *versioned* ⊆ *standard*). Depending on the TM mode, even unversioned transactions may cause addresses to become versioned (Mode U).

At commit time, the read set is revalidated. If this succeeds, the transaction rereads the global clock to obtain a *commit clock*. Then, any versioned addresses that were written have their TBD marks removed, and all locks are released (and their versions updated to the commit clock). Any validation failures cause the transaction to abort and retry, and after sufficiently many aborts, it may become versioned. See Section 4 for optimizations to unversioned transactions.

3.2.2 Versioned Transactions. Only read-only transactions can be versioned. (We discuss the possibility of versioned *writing* transactions under a weaker correctness condition in the extended version of this paper [9]). A versioned transaction begins in the same way as an unversioned transaction, except that its read clock also serves as a *versioned timestamp*. For each address read, the transaction will determine if the address is versioned. Depending on the TM mode, versioned transactions will either cause these addresses to become versioned (Mode Q), or rely on relevant addresses being versioned already by updaters (Mode U).

Accesses to versioned addresses involve version list traversals to find a suitable version. Such traversals are blocked by TBD markers forcing the executing thread to wait. The traversal continues once TBD markers are removed. If a suitable version is found, the data at that version is returned. Otherwise the transaction aborts and retries.

Versioned transactions additionally save some information in shared memory for use by the background thread in determining when to unversion addresses. Specifically, on the first attempt of a versioned transaction the thread will save its *initial versioned timestamp*, and when a versioned transaction commits, it computes the difference between the current global clock, and its initial versioned timestamp, and saves the resulting *commit timestamp delta* (see Section 4.4).

3.3 TM Modes

Multiverse dynamically switches between four modes: two **stable** (Mode Q & U) and two **transient** (Mode QtoU & UtoQ). Each mode changes the behavior of both unversioned and versioned transactions. A summary of the modes appears in table 1. In *Mode Q*, versioned transactions are responsible

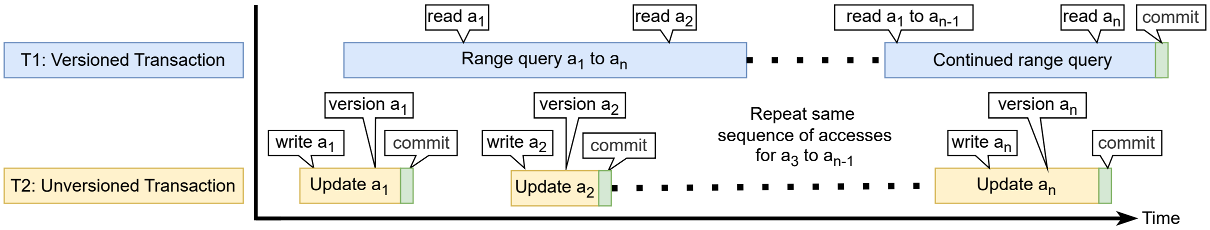


Figure 3. Example execution with the same transactions from Figure 4 but now the TM is in Mode U forcing the unversioned transaction to version each address it updates. The versioned transaction commits without any aborts.

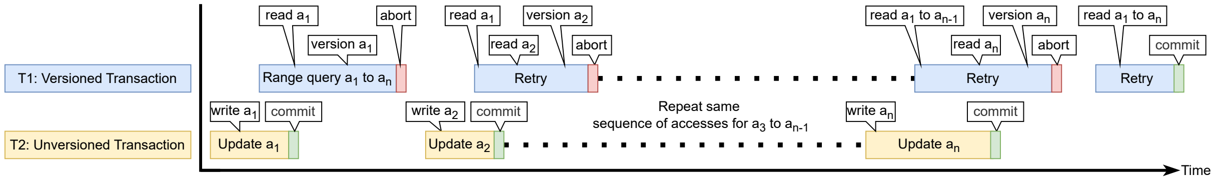


Figure 4. Example execution where Mode Q is not suitable. All addresses are initially unversioned. The versioned transaction T1 needs to read addresses a_1 to a_n but it must perform $O(n^2)$ accesses to commit as a result of aborts caused by conflicts with the concurrent unversioned transaction. T1 would perform only n accesses if the addresses were already versioned.

for versioning addresses, while unversioned transactions are largely oblivious to ongoing versioned transactions. Mode Q optimizes for unversioned transactions by ensuring new versions are added on demand only by versioned transactions that rely on them. This mode is suitable when there is low contention, or transactions are small enough that the relevant addresses can be versioned before a conflict occurs.

However, Mode Q is not suitable under high contention, or when there are long running transactions. Figure 4 shows an example execution where the TM is in Mode Q and a versioned transaction requires n^2 accesses to commit a transaction over n addresses. In this case, it would be better to use *Mode U*, which optimizes for versioned transactions. Figure 3 shows an example execution with the same transactions and accesses from Figure 4, but in this case the TM is in Mode U and the versioned transaction can commit without aborting.

In Mode U, unversioned transactions that write are forced to version addresses while versioned transactions operate as if every address is already versioned. Mode U essentially enables *global versioning for writes*. As a result, if an address is not versioned, then it has not been written since the TM entered Mode U. So, versioned transactions can safely read unversioned addresses without versioning them. Section 4.2 discusses the subtleties that arise if an address becomes versioned *while* it is being read.

The current (global) TM mode is visible to all transactions, and each transaction has a *local mode* it operates in. Before each attempt, a transaction records the current TM mode and uses it as its local mode. It is possible for the local mode to differ from the TM mode, since the TM mode can change after the transaction decides its local mode. For this reason, we cannot immediately change (globally) from TM Mode Q to Mode U since there might still be ongoing transactions operating *locally* in Mode Q. To ensure correct transitions between (global) modes, we use two transient modes. To transition the TM from Mode Q to Mode U, we first transition to *Mode QtoU*. The purpose of this mode is to allow ongoing local Mode Q writers to commit or abort before the TM enters Mode U. Similarly, when we transition from Mode U back to Mode Q, we first transition into *Mode UtoQ*, allowing ongoing versioned transactions in local Mode U to commit or abort before the TM enters Mode Q. Together, these transient modes ensure that Mode U transactions can always rely on writing transactions to version all written addresses—a property that concurrent Mode Q writing transactions would violate.

3.3.1 Transitioning between TM Modes.

The TM mode can only change in a fixed order: Mode Q, Mode QtoU, Mode U, Mode UtoQ, Mode Q, and so on. The TM begins in Mode Q, and while in Mode Q, any transaction can attempt transitioning the TM to Mode QtoU. All other mode transitions are performed by the same background thread that handles unversioning. The specifics of how we decide when to transition between modes is discussed in Section 4.3.

3.4 Correctness

Multiverse guarantees opacity. In this section we provide a brief intuitive correctness argument. Unversioned transactions in Multiverse ensure consistency via the versioned locks. On each TM access, the lock versions are validated against the transaction's read clock. Versioned transactions in Multiverse ensure in a similar manner. During a traversal of a version list, the timestamp of each individual version is validated against the transaction's versioned timestamp. In either case, a validation failure immediately causes an abort.

This ensures that a transaction only observes writes from other committed transactions; The use of timestamp based validation makes it easy to see that an equivalent sequential order would order transactions based on their read clocks / versioned timestamps.

It is easy to see that a history in which the global TM mode is fixed in (the default) Mode Q, is opaque since the synchronization would be the same as in DCTL. A history in which the TM mode changes is complicated by the fact that the local mode of a transaction can differ from the global mode. The classic case based TL2 opacity proof (as seen in Appendix A of [7]) can be applied to Multiverse. This approach would completely removes the notion of the TM modes. Alternatively, by considering histories that are partitioned based on the global TM mode, the opacity of a given history can be proved via an induction on the number of mode changes. We discuss this further in the extended version of this paper [9].

4 Implementation Details

In this section we describe the full details of the versioned and unversioned code paths for each of the TM modes. We also provide some pseudocode. Listing 1 shows the pseudocode for the TM interface functions `beginTxn`, `tryCommit` and `abort`. These functions are the same regardless of the local mode of the executing thread. **Listing 2** shows the pseudocode for TM reads and writes for the **simplified case where we assume all threads are in Mode Q.** Full pseudocode appears in an extended version of this paper [9].

4.1 Mode Q Code Paths

Update Transactions. Transactions that write (update), are always unversioned. A TM write (`modeQ_Write`) to an address begins by computing the mapping of the address to an index in the lock table. This same index is used for accessing the bloom filter table and the VLT. Next the associated lock is read. The lock may be marked to indicate that it is held by a concurrent transaction solely for the purpose of versioning, in which case we wait for the lock to be released. The lock version is then validated against the transaction's read clock (`validateLock`).

The transaction then attempts to claim the lock. After claiming the lock, the transaction checks if the address is versioned. If the address is versioned then the transaction will perform a *versioned write*. If the transaction has already written to the address then the head of the list will be marked TBD and the transaction will just update the data of the TBD version. If this is the first write to the address by this transaction then it adds a new version to the version list (`tryWriteToVersionList`). For the new version's timestamp, we use the transaction's versioned timestamp and mark it TBD. The address is then added to the transaction's versioned write set. Finally, the transaction then performs the *in-place write* to update the location read by unversioned transactions before adding the address to its standard write set.

```
thread locals: localMode, rClock, stickyModeU,
  readOnly, versioned, commitTSDelta, readCnt,
  consecSmallTxns, attempts, tid

beginTxn():
  setjmp()
  localMode = globalMode
  rClock = globalClock
  reset per-attempt data // logs and stats
  announce stickyModeU and localMode

tryCommit():
  if readOnly return
  if versioned
    announce commitTSDelta
    update consecSmallTxns
    stickyModeU = heuristic(readCnt)
  validateReadSet(rClock)
  commitClock = globalClock
  versionedWriteSet.unsetTBDs(commitClock)
  writeSet.releaseLocks(commitClock)
  consecSmallTxns++

abort():
  writeSet.rollback()
  snapshotWriteSet.rollback()
  clear eventual frees
  nextClock = gClock.increment()
  writeSet.unlock(nextClock);
  if readOnly
    if heuristic(localMode, readCnt, attempts)
      globalMode.transitionToMode(QtoU)
    versioned = heuristic(readCnt, attempts)
  attempts++
  longjmp() // retry at start of beginTxn
```

Algorithm 1. Pseudocode for TM interface functions

When an update transaction commits (`tryCommit`), the read set is revalidated. If the validation succeeds, the transaction will read the global clock to get its commit timestamp. The transaction then removes all of the TBD markers from addresses in its versioned write set before releasing its write set locks.

Any validation failures cause the transaction to abort (`abort`). If the transactions aborts, all of its writes will be rolled back. This includes rolling back any versioned writes by replacing the TBD marked timestamps with a *deleted timestamp* and retiring the added version. The deleted timestamp ensures that concurrent versioned transactions are not permanently blocked waiting for a TBD marked timestamp to be resolved.

Read-only Transactions. We track the number of TM reads performed during a read-only transaction. On abort, this read count is used to determine if the transaction is more likely to commit in Mode U. When a read-only transaction

```
thread locals: rClock, tid, readCnt
type VersionedLock:[locked, version, tid, flag]
type VListNode:[oldNode, timestamp, data, tbd]

modeQ_read(addr):
  readCnt++
  if versioned return modeQ_versionedRead(addr)
  data = *addr
  lockState = reread lock until flag is false
  if not validateLock(lockState,rClock) abort()
  readSet.add(addr)
  return data

modeQ_versionedRead(addr):
  if not bloomFltr.tryAdd(addr) // exists already
    vlist = tryGetVList(addr)
    if vlist return traverse(vlist)
  return versionThenRead(addr)

modeQ_write(addr, value):
  lockState = reread lock until flag is false
  if not validateLock(lockState, rClock) abort()
  if not tryLock(lockState,rClock) abort()
  standardWriteSet.add(addr, *addr) // undo log
  *addr = value
  tryWriteToVersionList(addr, value)

tryWriteToVersionList(addr, value)
  if not bloomFltr.contains(addr) return
  if vlist = tryGetVList(addr) is null return
  if vlist.head.tbd then vlist.head.data = value
  else
    vNode = [oldNode = head, timestamp =
      rClock, data = value, tbd = true]
    vlist.head = vNode // protected by addr lock
    versionedWriteSet.add(addr)
    eventualFree(vNode.next) // freed on commit

validateLock(lockState, rClock):
  if lockState.tid == tid return true
  if lockState.locked return false
  return lockState.version < rClock

versionThenRead(addr):
  lockState = lockAndFlag(addr)
  data = *addr
  add new version(lockState.version, data)
  unlock(addr)
  if not validateLock(lockState,rClock) abort()
  return data
```

Algorithm 2. Pseudocode assuming **all threads** in Mode Q

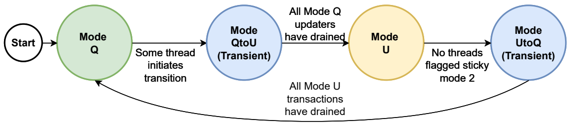


Figure 5. State transition diagram of the TM mode.

first begins it will always start as an unversioned transaction. Unversioned reads (`modeQ_read()`) follow an approach similar to writes, which are also unversioned. We find the associated lock, wait while the lock state indicates that the address is currently being versioned then validating the lock state against our read clock. An unversioned transaction which fails to commit after $\mathcal{K}_1$ attempts will switch to the versioned path. $\mathcal{K}_1$ is a tunable parameter.

Versioned readers in local Mode Q begin by checking if the address is versioned (`modeQ_versionedRead`). If the address is versioned then the transaction will traverse the version list beginning from the newest version to find a suitable version with a timestamp less than or equal to its versioned timestamp. This traversal is blocked if the newest version is suitable but marked TBD. If a suitable version is found the data is returned otherwise the transaction is aborted.

If the address is unversioned then the transaction will version it (`versionThenRead`). This requires updating the associated lock version to both claim the lock and mark indicating versioning is in progress. The transaction will repeatedly try-lock until successful, and then validate the lock version by comparing against its read clock. If this validation fails then, after versioning the address, the transaction must abort. The transaction must then re-check if the address is versioned since it is possible that a concurrent transaction versioned it while this transaction was waiting for the lock. If the address is still unversioned, the transaction will version it. This requires allocating a new version list, a new versioned node to serve as the initial version and a new VLT bucket node. For the initial version, we take the current value of the address for the data and, when the TM is in Mode Q, we use the lock version as the timestamp. The new VLT bucket containing the address and new version list is inserted at the front of the VLT bucket. The versioning process completes by inserting the address into the associated bloom filter after which the lock is released.

4.2 Mode U Code Paths

Update Transactions. In Mode U, and any mode other than Mode Q, updaters perform the same steps as in Mode Q up to and including checking if the address is versioned (pseudocode in full paper only). If the address is not versioned, updaters in local Mode U must version it by following the same steps as versioned readers in Mode Q (except the updater will already hold the lock).

In Mode U, when versioning an address, we can apply an optimization when choosing the timestamp of the initial version. Specifically, rather than using the lock version, we can use the timestamp that existed immediately after the TM entered into Mode U. This *first observed Mode U timestamp* is recorded by the background thread and stored in shared memory. Using an earlier timestamp will reduce the likelihood of aborting versioned readers. In Mode U, if a transaction needs to version an address, then it must be the first transaction to write to the address (otherwise it would already have been versioned). Since no writes occurred since the TM transitioned to Mode U, it is safe to use the earlier timestamp. (This optimization is also applied by read-only transitions in local Mode Q if the TM concurrently transitioned into Mode U after the reader obtained its local mode). The first observed Mode U timestamp is invalidated by the background thread before it transitions back to Mode Q ensuring that we only apply this optimization in Mode U. After versioning, the updater must also perform the versioned write and in-place write following the same steps as in Mode Q. Likewise, committing or aborting also follows the same steps as in Mode Q.

Read-only Transactions. When a read-only transaction is in local Mode U, it still begins as unversioned and the execution of unversioned reads is the same as in Mode Q. When reading an address, versioned read-only transactions in local Mode U still need to check if the address is versioned. If the address is already versioned then we perform the same traversal of the version list as in Mode Q.

If a versioned transaction in Mode U encounters an unversioned address then we know that there has not been any writes to this address since the transition to Mode U, otherwise, some other writer would have already versioned the address. However, we check if an address is versioned before reading the data. Since these steps are not done atomically together, it is possible for a concurrent transaction to update the address after we observe that it is unversioned, which could result in the versioned transaction observing inconsistent data. To prevent this, if a versioned transaction encounters an unversioned address it will read and make a local copy of the associated lock state. If it is unlocked, the versioned transaction must read the data at the address then reread the lock. If the lock version has not changed then we know that no concurrent updates to this address have occurred so it is safe to return the data that we read. If lock version has changed then the transaction must abort.

In the case that the (unversioned) address is locked then the versioned transaction must read and makes a local copy of the data then re-checks if the address is versioned. If the address is still unversioned, the transaction must redo the reads of the lock and data. An address in Mode U can be locked iff a writer is concurrently updating the address or due to a lock table collision. If the address is still unversioned but we observe a change in the lock version, then the address must have been locked as a result of a lock table collision, since otherwise, the writer holding the lock would have versioned the address. Alternatively, if the address is still unversioned and we do not observe a change in the lock version or the data, then the lock could be held by a transaction seeking to update this address. However, in this case, our first read of the data must have occurred before any such update, since again, otherwise, the address would have been versioned.

We record in shared memory, a global *minimum Mode U read count*, which is the minimum number of reads performed by versioned transactions that commit in Mode U. At commit, versioned transactions in local Mode U will update the minimum Mode U read count if they performed fewer reads. When a transaction in local Mode Q aborts, this value is used to predict if it is more likely to commit in Mode U.

4.3 How to Switch Between Modes

We utilize a monotonically increasing integer for the TM mode. Transitions require incrementing the TM mode. Figure 5 shows a state transition diagram which summarizes when Multiverse transitions between TM modes. A transaction in local Mode Q, can attempt a *compare-and-swap (CAS)* operation to transition the TM mode from Mode Q to Mode QtoU. After $\mathcal{K}_2$ attempts, an unversioned or versioned read-only transaction will attempt the CAS iff its read count is greater than or equal to the minimum Mode U read count. A versioned transaction will always attempt the CAS after $\mathcal{K}_3$ attempts. Both $\mathcal{K}_2$ and $\mathcal{K}_3$ are tunable parameters.

Any thread that attempts this CAS sets a thread-local *sticky bit* to indicate that it wants to operate in Mode U. The background thread will inspect this bit to decide whether to remain in Mode U. This flag bit is removed after the thread completes $\mathcal{S}$ consecutive *small* transactions, where $\mathcal{S}$ is a tunable parameter. The size of a transaction refers to the number of TM reads performed by the transaction. Any unversioned transaction is considered small. Each thread dynamically computes its own *small transaction read count* to be $\frac{1}{\mathcal{S}}$ times the size of the transaction that the thread first committed after its last attempt of the CAS.

The background thread handles all TM mode transitions whenever the TM is not in Mode Q. Mode transitions by the background thread are performed via atomic writes. The background thread determines when to transition by examining the local mode and per-thread sticky bit of every active thread. The background thread will repeatedly iterate over this data for all active threads until an iteration is completed satisfying certain mode-specific conditions.

Transitioning from Mode QtoU to Mode U requires completing an iteration of the relevant thread data without observing any update transactions with a local mode that is less than the value of current TM mode. Immediately after this transition, the background thread will read and save in shared memory, the first observed Mode U timestamp (to be used in the optimized path when versioning). Transitioning from Mode U to Mode UtoQ requires completing an iteration of the relevant thread data without observing any threads with the sticky Mode U flag. Finally, transitioning from Mode UtoQ back to Mode Q requires completing an iteration of the relevant thread data without observing any versioned transactions with a local mode whose value is less than the value of the current TM mode. Immediately prior to performing this transition, the background thread invalidates the first observed Mode U timestamp.

4.4 How to Unversion

Unversioning is only enabled in Mode Q. We unversion any VLT bucket in which there is a sufficiently large difference between the most recent timestamp of any version in the bucket and the current global clock version. We utilize a heuristic approach to determine a suitable difference which is computed as follows: First the background thread will compute the average of all transactions' commit timestamp deltas. It will then add the average to a list. The background thread will repeat this process until the list reaches a size of $\mathcal{L}$ where $\mathcal{L}$ is given as a tunable parameter. The background thread then sorts the list into descending order, and then computes the average of a prefix of the list. The length of the prefix, $\mathcal{P}$, is also a tunable parameter. Next the background thread iterates over each VLT bucket and unversions the bucket if the difference between the current global clock version and most recent timestamp in the bucket is larger than the average of the prefix. To unversion, the background thread will acquire the associated lock. It then removes and retires every node in the bucket along with every version list in those nodes after which the lock is released.

4.5 Memory Management

During a transaction, all allocations are buffered such that they can be rolled back if the transaction aborts. We utilize a simple epoch based reclamation (EBR) mechanism to enable safe memory reclamation within transactions. EBR pairs naturally with TM since we can tie the epoch management into transaction commits and aborts. Immediately after an update transaction adds a new version to a version list, the previous version is retired. However, if the transaction aborts then the previous version should not be reclaimed. Thus, when we rollback the effects of an update transaction we also revoke any of its retires. Any of the new versions added by an aborted update transaction will also be retired (these retires will not be revoked).

Algorithms like TL2 and DCTL benefit from the fact that they do not require read-only transactions to revalidate read sets. While this tends to lead to better performance, it permits a crucial memory reclamation race condition. Consider a concurrent singly linked list where the synchronization is handled by DCTL. Suppose our list contains four nodes: A, B, C and D (linked in that order). Furthermore, consider two threads t_1 and t_2. t_1 will execute a transaction to read the entire list from A to D and t_2 will concurrently execute a transaction to remove the latter half of the list by removing C and D via a single write to change B's next pointer to `null`. Let t_1 begin and progress until it reaches C. During its traversal, t_1 will pass all validation. Now we let t_2 begin. t_2 will traverse the list until it reaches B, successfully validating each read and adding the addresses to its read-set. t_2 will then successfully claim the lock protecting B's next pointer before performing the write. At commit time, t_2 will successfully revalidate its read-set and commit. Now, C and D have been unlinked from the list but since t_1 will not revalidate any of its reads, it will not be aborted. If t_2 then, deallocates C and D, t_1 could experience a segmentation fault when it continues its traversal. TL2 also permits this scenario.

In section 2.3 of the TL2 paper, the authors mention the problem of memory reclamation but they only discuss an issue where we want to free objects that exist within the write-set of a transaction. Their proposed reclamation scheme only works if the issue can be caught during read-set validation, which will never occur in our example. This problem is also briefly discussed in [4] and the authors similarly avoid it via EBR. Alternatively, we could require read-only transactions perform revaluation or we could utilize an allocator where the internal state is accessed via transactions. Both alternatives have significant performance drawbacks.

5 Evaluation

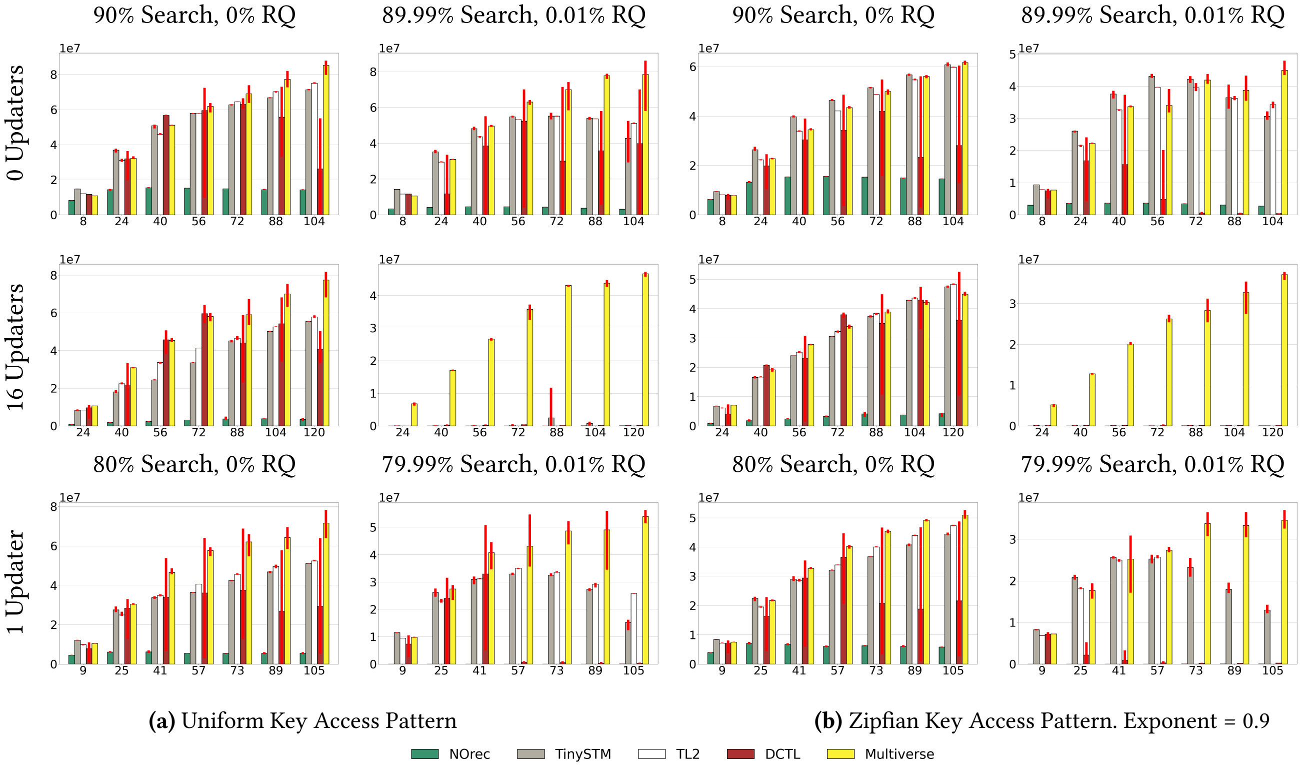
We implemented Multiverse in C++. The implementation is publicly available [8][1]. The code was compiled with GCC 10.3.0 with an optimization level of -O2. We compare against several existing STMs which also guarantee opacity. Specifically TL2 [12], DCTL [25], NOrec [10] and TinySTM [15]. These STMs are described in Section 6. For TL2 and TinySTM we use the author's public implementations. For NOrec we used same implementation as in [7]. While DCTL has no public implementation, the authors have a public implementation of a similar algorithm [26] which we could easily adapt into DCTL. We used the same benchmark as [6] for our evaluation. We ran all experiments on a single AMD EPYC 7662 processor which has 64 cores and 128 hardware threads. All results report the average of 5 trials. The measurement period of each trial is 20 seconds. In this paper we show a representative sample of our results using an (a,b)-tree. In the extended version of this paper we show additional experiments featuring other data structures including an internal AVL tree, an external binary search tree and a hashmap.

Tunable Parameters. For TL2 we use the GV4 global clock implementation. DCTL requires specifying the number of aborts before falling back to a starvation free mode for which we use 100 (the maximum used in [25]). For Multiverse we use $\mathcal{K}_1$=100, $\mathcal{K}_2$=16, $\mathcal{K}_3$=28, $\mathcal{S}$=10, $\mathcal{L}$=10 and $\mathcal{P}$=10%. For both Multiverse and DCTL we use the same linear backoff as in [26]. Unmentioned parameters use default values.

Experimental Setup. We use our own benchmark because prior TM benchmarks are not compelling if we are interested in supporting long queries. The goal of our experimental setup is to test a plausibly realistic workload where

---

[1] The code can be found at: https://zenodo.org/records/18099743



(a) Uniform Key Access Pattern

(b) Zipfian Key Access Pattern. Exponent = 0.9

Figure 6. Throughput for (a,b)-tree, a=4 b=16, prefilled to 1 million keys. Y-axis: average ops/sec. X-axis: number of threads. In all workloads the remaining percentage of work is equal parts insert and delete. RQ size is 10k (1% of prefill size).

an algorithm must be able to perform large range queries (RQs) reliably in the presence of updates in order to obtain high performance overall.

This *sounds* easy: Why not use workloads in which threads have some probability of performing a search, insert, delete or RQ? It turns out such workloads can obscure performance problems with RQs, artificially inflating the performance of inadequate algorithms. Suppose updates always abort RQs. A thread attempting a RQ that is repeatedly aborting can effectively wait until other threads that were completing conflicting operations happen to roll the dice and determine that they must now also perform a RQ. If, eventually, all threads perform RQs at the same time, they will all succeed.

To avoid this issue, we add dedicated *updater* threads. These threads keep RQ algorithms "honest"—they must deal with contention in order to perform RQs consistently. Since these dedicated updater threads can continue to perform operations even if an algorithm has no ability to perform RQs, we do not count the throughput of the dedicated updaters towards the overall throughput (otherwise we would reward algorithms with poor RQ support).

This approach still requires care. Setting the percentage of RQs too high will slow down other operations dramatically. For example, if an algorithm can perform 1000 RQs/sec, and 10% of all operations are RQs then, in expectation, for each RQ performed, only 9 other operations can be performed. In our example, this means that the total throughput will be at most 10x the speed of RQs or 10k ops/sec. In such a workload, what we would measure is almost entirely the cost of performing RQs. For this reason, we evaluate workloads with less than 1% RQs.

In our experiments, dedicated updaters increase the *effective update rate*, i.e., the percentage of operations that are updates *in total.* Since the number of dedicated updaters is fixed, the effective update rate depends on the number of worker threads. This can be measured. At high thread counts, without RQs, workloads with 16 dedicated updaters and worker threads performing 10% updates actually have effective update rates close to 20%. When the workload includes RQs, the effective update rate can vary greatly depending on whether or not the TM actually supports efficient RQs. For example, with 0.01% RQs and 16 dedicated updaters, 99% of all operations performed by TinySTM are updates with the majority of those being performed by dedicated updaters.

Preserving Short Query Performance. For workloads that do not feature RQs, versioning is typically not necessary. In these workloads, we want Multiverse to maintain the performance of the state of the art STM. As demonstrated in columns 1 and 3 of Figure 6, Multiverse achieves this goal achieving throughput comparable or better to all of the TMs that we test. This is possible because our versioned transactions are only executed on-demand.

Supporting Long Running Queries. For workloads with some percentage of RQs, versioning is useful. As seen in the first row of Figure 6, when there are no dedicated updaters to cause conflicts with the RQs, all of the TMs can achieve reasonable throughput but Multiverse still performs better. On the other hand, in the second row of Figure 6 we show that when there are some dedicated updaters and RQs, Multiverse significantly outperforms the other TMs. In many cases the throughput of the other TMs is too low to display. Even worse, it is common for the other TMs to have transactions reach their maximum allowed aborts and quit. In the third row of Figure 6 we have worker threads perform a larger amount of update operations so we reduce the number of dedicated updaters. Even in this case with only a single dedicated updater Multiverse outperforms the other TMs especially for workloads with RQs.

DCTL Starvation Freedom. One will notice that the variance in DCTL is very high. In many cases for workloads without RQs, DCTL's maximum throughput is similar or better compared to the other TMs. However, its minimum performance is often near zero. This is a result of its irrevocable transactions which must claim locks on reads (which can abort other transactions). Since only a single transaction can be irrevocable at any time, this can lead to many or all transactions waiting to execute on the irrevocable path.

Time Varying Workloads. To understand the benefit of our different TM modes, we experiment with time varying workloads. Specifically, we split each trial into 4 intervals of 5 seconds and we change the workload in each interval. In these experiments we measure throughput over time using an additional background thread which captures the throughput every 200ms. We also separately show implementations of Multiverse where we disable mode switching and force the initial mode to Mode Q or Mode U respectively.

We use a workload with 2 repeated intervals where interval 1 and 3 have no RQs and no dedicated updaters and interval 2 and 4 have 0.01% RQs with a large RQ size of 100k and 4 dedicated updaters. All 4 intervals have 10% insert and 10% deletes with the remaining work being searches (point queries). Figure 7 shows the results of this experiment. In interval 1 our Mode Q only implementation performs noticeably better than our Mode U only implementation and the opposite is true once we introduce RQs. Our implementation with mode switching enabled achieves performance comparable to the better of the mode restricted implementations in each interval despite us not investing much effort into finely tuning our parameters for mode switching.

One might expect all TMs to increase in throughput when RQs are removed from the workload but this does not occur. When we change intervals, newly generated work conforms to the new workload however, any queries that started in the previous interval must finish before the thread can continue under the new workload (this matches the reality of varying workloads in the real world). In other words, a thread running a large RQ will continue to attempt the RQ until it succeeds. This means that algorithms that do not support the larger RQs are likely to be stuck retrying those RQs forever even after an interval change.

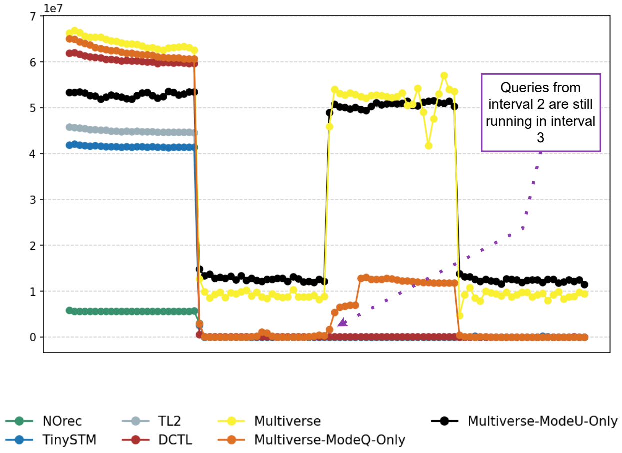


Figure 7. Throughput over time for an (a,b)-tree using 64 worker threads for a time-varying workload with 4 intervals. Intervals 1 and 3 have no RQs and no dedicated updaters. Interval 2 and 4 have 0.01% RQs with a RQ size of 100k and 4 dedicated updaters. All intervals have 10% insert, 10% deletes and the remaining work is searches (point queries). Y-axis: average ops/sec. X-axis: time (seconds).

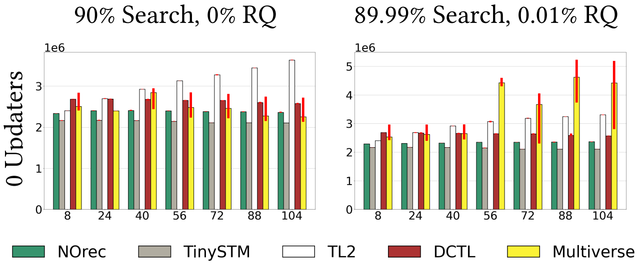


Figure 8. Maximum memory usage for the same (a,b)-tree from Figure 6 using a uniform key access pattern. Y-axis: max resident memory in KB. X-axis: number of threads.

Memory Usage. Figure 8 shows the maximum memory usage for the same (a,b)-tree from row one of Figure 6. In general, it is expected that multiversioned algorithms will require more memory compared to unversioned algorithms. However, as a result of our dynamic multiversioning approach, we only pay the cost of additional memory requirements when multiple versions are actually needed. For workloads without RQs the maximum memory used by Multiverse is comparable to (and sometimes lower than) DCTL.

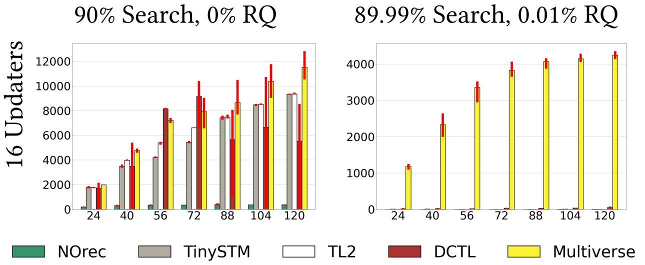


Figure 9. Throughput per joule consumed by the CPU package for the same (a,b)-tree workload from Figure 6 using a uniform key access pattern. Y-axis: average ops/sec per joule. X-axis: number of threads.

Power Consumption. Power Consumption is another interesting metric to consider. We compare the energy efficiency of the TMs via `perf` by measuring the power consumption of the CPU package in joules (specifically measuring the `energy-pkg` hardware event). Note that it is not possible to measure the energy consumption of a specific program or core [20, 24]. Figure 9 shows average throughput per joule of energy for the same (a,b)-tree from row two of Figure 6. Multiverse is able to leverage increased memory usage to efficiently support RQs resulting in up to 50x improved energy efficiency compared to the next best TM.

6 Related Work

Transactional Locking II (TL2) [12] is one of the most well known STMs. TL2 is an opaque unversioned STM that relies on a global clock and per-address versioned locks. By default, TL2 uses *buffered writes* and deferred *commit time locking*. Transactions that write increment the global clock at commit time after locking the write set. Numerous optimizations of TL2 exist, especially new approaches for managing the global clock. The most recent of these optimizations was presented in a new STM, Deferred Clock Transactional Locking (DCTL), [25]. DCTL uses encounter time locking and increments the global clock only on aborts. It has a starvation free mode where a single transaction can become irrevocable. Multiple non-irrevocable transactions can execute concurrently with a single irrevocable transaction. TLRW [13] is an STM that relies on byte-locks which are a form of read-write locks. TLRW is unversioned but it does support irrevocable transactions. It has been shown to be competitive with TL2 for single chip machines in some workloads. Unfortunately TLRW has no public implementation. TinySTM [15] is another well known opaque unversioned STM. Like TL2, it relies on a global clock and per-address versioned locks. TinySTM uses encounter time locking. No Ownership Records (NORec) [10] is an unversioned STM that does not use per-address versioned locks. It uses a single global sequence lock, commit time locking and value based validation. These STMs do not have proper support for long running read-only transactions.

Verlib [5] is the state of the art MVCC mechanism. Notably, Verlib can sometimes avoid adding indirection when updating addresses. Verlib was incorporated into a sort of restricted STM that can be used only for implementing optimistic data structures in [4]. Unlike Multiverse this restricted TM does not guarantee opacity. Other non-opaque multiversion STMs have also been proposed [22, 27]. Selective Multi-Versioning (SMV) [23] is an opaque STM that uses an approach somewhat similar to Verlib. We would have liked to compare against SMV [23], but it is implemented in Java and heavily relies on garbage collection. To our knowledge there is no public non-Java implementation and it is not straightforward to implement it in C++, since we would need to solve the memory management problem. SMV uses commit time locking. In SMV updates always add new versions but old versions can be quickly removed. Active transactions in SMV append a descriptor to a global list. A descriptor has a timestamp and keeps references to the old versions of objects that the transaction modified to prevent their reclamation by the garbage collector. Unlike Verlib which uses a numeric global timestamp, SMV uses the list of descriptors to serve as a global clock which requires different synchronization.

7 Conclusion

In this work we presented Multiverse, a new opaque STM that combines the best of both unversioned STM and MVCC. Multiverse features dynamic multiversioning and uses multiple TM modes which adapt the behaviour of the TM to fit the needs of the workload. Our experimental evaluation of Multiverse demonstrated that our TM can match or exceed the performance of existing unversioned STMs even in common case workloads that do not feature long running read-only transactions while still significantly outperforming them for workloads that do feature long running reads.

Acknowledgments

We thank the reviewers for their helpful comments and suggestions. We also thank Naama Ben-David for her involvement in numerous discussions during the early development of the algorithm. Naama's ideas inspired various initial versions of Multiverse. Her insights and feedback were instrumental in overcoming several issues and helped to guide the evolution of the algorithm.

This work was supported by: the Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: CRDPJ 539431-19, the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, NSERC Discovery Launch Supplement: DGECR-2019-00048, NSERC Discovery Program grant: RGPIN-2019-04227, and the University of Waterloo.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. *Journal of the ACM (JACM)* 40, 4 (1993), 873–890.

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[3] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading, Boston, USA.

[4] Guy E Blelloch, Zachary Kent, and Yuanhao Wei. 2025. TLF: Transactional Lock Fusion. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures*. Association for Computing Machinery, New York, NY, USA, 116–130.

[5] Guy E Blelloch and Yuanhao Wei. 2024. Verlib: Concurrent versioned pointers. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, United States, 200–214.

[6] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. *Non-Blocking Interpolation Search Trees with Doubly-Logarithmic Running Time*. Association for Computing Machinery, New York, NY, USA, 276–291. https://doi.org/10.1145/3332466.3374542

[7] Trevor Brown and Srivatsan Ravi. 2017. Cost of Concurrency in Hybrid Transactional Memory. In *31st International Symposium on Distributed Computing (DISC 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 91)*, Andréa Richa (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:16. doi:10.4230/LIPIcs.DISC.2017.9

[8] Gaetano Coccimiglio, Trevor Brown, and Srivatsan Ravi. 2025. *Multiverse: Transactional Memory with Dynamic Multiversioning (Extended Version)*. doi:10.5281/zenodo.18099743

[9] Gaetano Coccimiglio, Trevor Brown, and Ravi Srivatsan. 2026. Multiverse: Transactional Memory with Dynamic Multiversioning. doi:10.5281/zenodo.18189368

[10] Luke Dalessandro, Michael F Spear, and Michael L Scott. 2010. NOrec: streamlining STM by abolishing ownership records. *ACM Sigplan Notices* 45, 5 (2010), 67–78.

[11] Dave Dice, Timothy L Harris, Alex Kogan, Yossi Lev, and Mark Moir. 2014. Pitfalls of lazy subscription. In *Proceedings of the 6th Workshop on the Theory of Transactional Memory, Paris, France*.

[12] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *International Symposium on Distributed Computing*. Springer, Springer, Berlin, Heidelberg, 194–208.

[13] Dave Dice and Nir Shavit. 2010. TLRW: return of the read-write lock. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. Association for Computing Machinery, New York, NY, USA, 284–293.

[14] Robert Ennals and Robert Ennals. 2006. *Software transactional memory should not be obstruction-free*. Technical Report. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report.

[15] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. Association for Computing Machinery, New York, NY, United States, 237–246.

[16] Keir Fraser and Tim Harris. 2007. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)* 25, 2 (2007), 5–es.

[17] Rachid Guerraoui and Michał Kapałka. 2010. *Principles of transactional memory*. Vol. 4. Morgan & Claypool Publishers, California USA.

[18] Tim Harris, James Larus, and Ravi Rajwar. 2022. *Transactional memory*. Springer Nature, Berlin, Heidelberg.

[19] Maurice Herlihy and J Eliot B Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*. Association for Computing Machinery, New York, NY, USA, 289–300.

[20] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. 2018. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3, 2 (2018), 1–26.

[21] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. 2009. Anatomy of a scalable software transactional memory. In *Proc. 4th ACM SIGPLAN Workshop on Transactional Computing*. Association for Computing Machinery, New York, NY, United States.

[22] Li Lu and Michael L Scott. 2013. Generic multiversion STM. In *International Symposium on Distributed Computing*. Springer, Springer, Berlin, Heidelberg, 134–148.

[23] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. 2011. SMV: Selective multi-versioning STM. In *International Symposium on Distributed Computing*. Springer, Springer, Berlin, Heidelberg, 125–140.

[24] Guillaume Raffin and Denis Trystram. 2025. Dissecting the software-based measurement of CPU energy consumption: a comparative analysis. *IEEE Transactions on Parallel and Distributed Systems* 36 (2025), 96–107.

[25] Pedro Ramalhete and Andreia Correia. 2024. Scaling Up Transactions with Slower Clocks. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, United States, 2–16.

[26] Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2021. Efficient algorithms for persistent transactional memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, 1–15.

[27] Torvald Riegel, Christof Fetzer, and Pascal Felber. 2006. Snapshot isolation for software transactional memory. *TRANsACT06* 298 (2006).

[28] Nir Shavit and Dan Touitou. 1995. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. Association for Computing Machinery, New York, NY, USA, 204–213.

[29] Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, 31–46.

[30] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

Received 2025-09-01; accepted 2025-11-10