Exploiting Semantic Commutativity in Hardware Speculation

Guowei Zhang Virginia Chiu Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{zhanggw, vchiu, sanchez}@csail.mit.edu

***Abstract*—Hardware speculative execution schemes such as hardware transactional memory (HTM) enjoy low run-time overheads but suffer from limited concurrency because they rely on reads and writes to detect conflicts. By contrast, software speculation schemes can exploit semantic knowledge of concurrent operations to reduce conflicts. In particular, they often exploit that many operations on shared data, like insertions into sets, are *semantically commutative*: they produce semantically equivalent results when reordered. However, software techniques often incur unacceptable run-time overheads.**

To solve this dichotomy, we present COMMTM, an HTM that exploits semantic commutativity. COMMTM extends the coherence protocol and conflict detection scheme to support user-defined commutative operations. Multiple cores can perform commutative operations to the same data concurrently and without conflicts. COMMTM preserves transactional guarantees and can be applied to arbitrary HTMs.

COMMTM scales on many operations that serialize in conventional HTMs, like set insertions, reference counting, and top-K insertions, and retains the low overhead of HTMs. As a result, at 128 cores, COMMTM outperforms a conventional eager-lazy HTM by up to 3.4× and reduces or eliminates aborts.

I. INTRODUCTION

Many software and hardware techniques, such as transactional memory (TM) or speculative multithreading, rely on speculative execution to parallelize programs with atomic regions. Multiple atomic regions are executed concurrently, and a *conflict detection* technique flags potentially unsafe interleavings of memory accesses (e.g., in TM, those that may violate serializability). Upon a conflict, one or more regions are rolled back and reexecuted to preserve correctness.

Ideally, conflict detection should (1) be *precise*, i.e., allow as many safe interleavings as possible to maximize concurrency, and (2) incur minimal run-time costs. Software and hardware conflict detection techniques satisfy either of these properties but sacrifice the other: software techniques can leverage program semantics to be highly precise, but they incur high run-time overheads (e.g., 2-6× in software TM [10]); meanwhile, hardware techniques incur small overheads, but are imprecise because they rely on reads and writes to detect conflicts.

In particular, software conflict detection techniques often leverage *semantic commutativity* of transactional operations to reduce conflicts. Two operations are semantically commutative when reordering them produces results that are semantically equivalent, even if the concrete resulting states are different. Semantically commutative operations executed in concurrent transactions need not conflict. For example, consider two consecutive insertions of different values `a` and `b` to a set `s` implemented as a linked list. If `s.insert(a)` and `s.insert(b)` are reordered, the concrete representation of these elements in set `s` will be different (either `a` or `b` will be in front). However, since the order of elements in `s` does not matter (a set is an unordered data structure), both representations are semantically equivalent. Therefore, insertions into sets commute. Software techniques can leverage such semantic commutativity to perform set insertions concurrently instead of serializing them. Semantic commutativity is common in other contexts beyond this simple example [13, 25, 34, 41]. Semantic commutativity was first exploited in the 1980s [51], and is now common in databases [6, 34], and parallelizing compilers and runtimes [25, 36, 41] (Sec. II).

By contrast, hardware conflict detection schemes do not exploit commutativity. The key reason is that hardware schemes leverage the coherence protocol to detect conflicts cheaply, and current protocols only support reads and writes. For instance, in the example above, concurrent transactions that insert into the same set conflict because they read and write the set's head pointer, and are serialized. This lack of precision can significantly limit concurrency, to the point that prior work finds that commutativity-aware software TM (STM) outperforms hardware TM (HTM) despite its higher overheads [25, 26].

To solve this dichotomy, we present COMMTM, a commutativity-aware HTM (Sec. III). COMMTM extends the coherence protocol with a *reducible* state. Lines in this state must be tagged with a user-defined *label*. Multiple caches can hold a given line in the reducible state with the same label, and transactions can implement commutative operations through labeled loads and stores that keep the line in the reducible state. These commutative operations proceed concurrently, without triggering conflicts or incurring any communication. A non-commutative operation (e.g., a conventional load or store) triggers a user-defined reduction that merges the different cache lines and may abort transactions with outstanding reducible updates. For instance, in the example above, multiple transactions can perform concurrent `insert` operations by acquiring the set's descriptor in *insert-only* mode and appending elements to their local linked lists. A normal read triggers an *insert-reduction* that merges the local linked lists.

COMMTM bears interesting parallels to prior commutativity-

978-1-5090-3508-3/16/$31.00

aware STMs. There is a wide spectrum of STM conflict detection schemes that trade precision for additional overheads. Similarly, we explore several variants of COMMTM that trade precision for hardware complexity. First, we present a basic version of COMMTM (Sec. III) that achieves the same precision as software *semantic locking* [25, 51]. We then extend COMMTM with *gather requests* (Sec. IV), which allow software to redistribute reducible data among caches, achieving much higher concurrency in important use cases.

We evaluate COMMTM with microbenchmarks (Sec. VI) and full TM applications (Sec. VII). Microbenchmarks show that COMMTM scales on a variety of commutative operations that allow no concurrency in conventional HTMs, such as set insertions, reference counting, ordered puts, and top-K insertions. At 128 cores, COMMTM improves full-application performance by up to 3.4×, lowers private cache misses by up to 45%, and reduces or even eliminates transaction aborts.

II. BACKGROUND

A. Semantic Commutativity

Semantic commutativity is frequently used in software conflict detection schemes [20, 25, 26, 34, 36, 40, 51]. Most work in this area focuses on techniques that reason about operations to abstract data types. Not all commutativity-aware conflict detection schemes are equally precise: simple and general techniques, like semantic locking [25, 40, 51], flag some commutative operations as conflicts, while more sophisticated schemes, like gatekeepers [25], incur fewer conflicts but have higher overheads and are often specific to particular patterns.

In this work we focus on semantic locking [40, 51], also known as abstract locking. Semantic locking generalizes read-write locking schemes (e.g., two-phase locking): transactions can acquire a lock protecting a particular object in one of a number of modes; multiple semantically-commutative methods acquire the lock in a compatible mode and proceed concurrently. Semantic locking requires additional synchronization on the actual accesses to shared data, e.g., logging or reductions.

COMMTM allows at least as much concurrency as semantic locking, with the added benefit of reducing communication by using caches to buffer and coalesce commutative updates. With gather requests (Sec. IV), COMMTM allows more concurrency than semantic locking.

B. Commutativity-Aware Cache Coherence

Unlike software conflict detection schemes, hardware schemes detect conflicts using read-write dependences. The reason is that they rely on the coherence protocol, which operates in terms of reads and writes. Recently, Coup [54] has shown that the coherence protocol can be extended to support local and concurrent commutative updates. Coup allows multiple caches to simultaneously hold update-only permission to the same cache line. Caches with update-only permission can buffer commutative updates (e.g., additions or bit-wise operations), but cannot satisfy read requests. Upon a read request, Coup reduces the partial updates buffered in private caches to produce the final value.

Like Coup, COMMTM modifies the coherence protocol to support a new state that does not trigger coherence actions on updates, avoiding conflicts. However, Coup does not work in a transactional context (only for single-instruction updates) and is restricted to a small set of *strictly commutative* operations, i.e., those that produce the same bit pattern when reordered. Instead, COMMTM supports the broader range of multi-instruction, semantically commutative operations. Moreover, COMMTM shows that there is a symbiotic relationship between semantic commutativity and speculative execution: COMMTM relies on transactions to make commutative multi-instruction sequences atomic, so semantic commutativity would be hard to exploit without speculative execution; and COMMTM accelerates speculative execution much more than Coup does single-instruction commutative updates, since apart from reducing communication, COMMTM avoids conflicts.

III. COMMTM

We now present the COMMTM commutativity-aware HTM. COMMTM extends the coherence protocol and conflict detection scheme to allow multiple private caches to simultaneously hold data in a user-defined *reducible* state. Transactions can use *labeled memory operations* to read and update these private, reducible lines locally without triggering conflicts. When another transaction issues an operation that does not commute given the current reducible state and label (i.e., a normal load or store or a labeled operation with a different label), COMMTM transparently performs a user-defined reduction before serving the data. This approach *preserves transactional guarantees*: semantically-commutative operations proceed concurrently to improve performance, but non-commutative operations cannot observe reducible lines with partial updates.

We first introduce COMMTM's programming interface and ISA. We then present a concrete COMMTM implementation that extends an eager-lazy HTM baseline. Finally, we show how to generalize COMMTM to support other HTM designs.

A. CommTM Programming Interface and ISA

COMMTM requires simple program changes to exploit commutativity: defining a *reducible state* to avoid conflicts among commutative operations, using *labeled memory accesses* to perform each commutative operation within a transaction, and implementing *user-defined reduction handlers* to merge partial updates to the data.

In this section, we use a very simple example to introduce COMMTM's API: concurrent increments to a shared counter. Counter increments are both strictly and semantically commutative; we later show how COMMTM also supports more involved operations that are semantically commutative but not strictly commutative, such as top-K insertions. Fig. 1 shows how COMMTM allows multiple transactions to increment the same counter concurrently without triggering conflicts.

User-defined reducible state and labels: COMMTM extends the conventional exclusive and shared read-only states with a reducible state. Lines in this reducible state must be tagged with a *label*. The architecture supports a limited number of labels

(e.g., 8). The program should allocate a different label for each set of commutative operations; we discuss how to virtualize these labels in Sec. III-D. Each label has an associated, user-defined *identity value*, which may be used to initialize cache lines that enter the reducible state. For example, to implement commutative addition, we allocate one label, `ADD`, to represent deltas to shared counters, and set its identity value to zero.

Labeled load and store instructions: To let the program denote what accesses form a commutative operation, COMMTM introduces labeled memory instructions. A labeled load or store simply includes the label of its desired reducible state, but is otherwise identical to a normal memory operation. For instance, commutative addition can be implemented as follows:

```
void add(int* counter, int delta) {
  tx_begin();
  int localValue = load[ADD](counter);
  int newLocalValue = localValue + delta;
  store[ADD](counter, newLocalValue);
  tx_end();
}
```

`load[ADD]` and `store[ADD]` inform the memory system that it may grant reducible permission with the `ADD` label to multiple caches. This way, multiple transactions can perform commutative additions locally and concurrently. This sequence is performed within a transaction to guarantee its atomicity (this code may also be called from another transaction, in which case it is handled as a conventional nested transaction [31]).

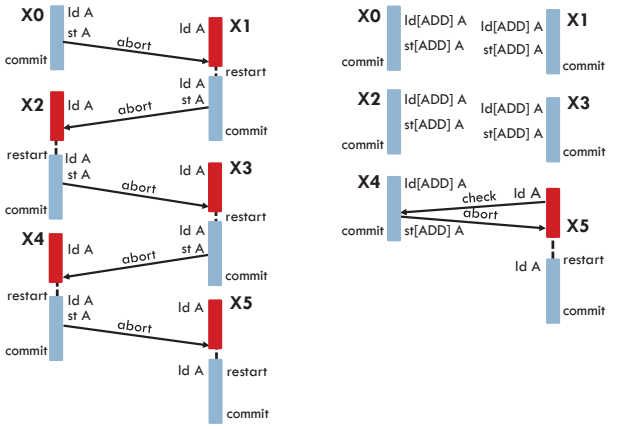
User-defined reductions: Finally, COMMTM requires the program to specify a per-label reduction handler that merges reducible cache lines. This function takes the address of the cache line and the data to merge into it. For example, the reduction operation for addition is:

```
void add_reduce(int* counterLine, int[] deltas) {
  for (int i = 0; i < intsPerCacheLine; i++) {
    int v = load[ADD](counterLine[i]);
    int nv = v + deltas[i];
    store[ADD](counterLine[i], nv);
  }
}
```

Unlike commutative operations done through labeled loads and stores, reduction handlers are *not transactional*. Moreover, to ease COMMTM's implementation, we restrict the types of accesses that reduction handlers can make. Specifically, while handlers can access arbitrary data with read-only and exclusive permissions, they should not trigger additional reductions (i.e., they cannot access other lines in reducible state).

Arbitrary object sizes: COMMTM operates at cache-line granularity, so smaller or larger objects need additional conventions.

To support objects smaller than a cache line, COMMTM requires data to be aligned to object-size boundaries. For example, each 64-byte line can hold up to 8 8-byte counters, each aligned at a 8-byte boundary. Because a reduction of arbitrary data with the identity value leaves the data unchanged, padding is unnecessary. Reduction handlers simply assume that lines are full of aligned values, and reduce all data in the line. For example, the reduction handler above tries to reduce all 8 possible counters; if the line has only one counter, data



(a) Conventional HTM (b) COMMTM

Fig. 1. Example comparing (a) a conventional HTM and (b) COMMTM. Transactions X0–X4 increment a shared counter, and X5 reads it. While conventional HTMs serialize all transactions, COMMTM allows commutative operations (additions in X0–X4) to happen concurrently, serializing non-commutative operations (the load in X5) only.

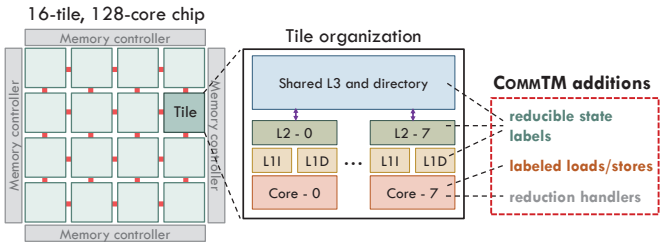


Fig. 2. Baseline system and main COMMTM additions.

surrounding the counter will be reduced with identity elements (zeros) and will remain unchanged.

To support objects larger than a cache line, COMMTM relies on indirection, using the reducible cache line to hold pointers to the object's partial updates. As we will see in Sec. VI, this naturally arises with data structures like sets or linked lists.

B. CommTM Implementation

B.1. Eager-Lazy HTM Baseline

To make our discussion concrete, we present COMMTM in the context of a specific HTM baseline. This HTM uses eager conflict detection and lazy (buffer-based) version management, as in LTM [4] and Intel's TSX [53]. We assume a multicore system with per-core private L1s and L2s, and a shared L3, as shown in Fig. 2. Cores buffer speculatively-updated data in the L1 cache; the L2 has non-speculative data only. Evicting speculatively-accessed data from the L1 causes the transaction to abort. The HTM uses the coherence protocol to detect conflicts eagerly. Transactions are timestamped, and timestamps are used for conflict resolution [30]: on a conflict, the earlier transaction wins, and aborted transactions use randomized backoff to avoid livelock. This conflict resolution scheme frees eager-lazy HTMs from common pathologies [9].

B.2. Coherence protocol

COMMTM extends the coherence protocol with an additional state, *user-defined reducible (U)*. For example, Fig. 3 shows

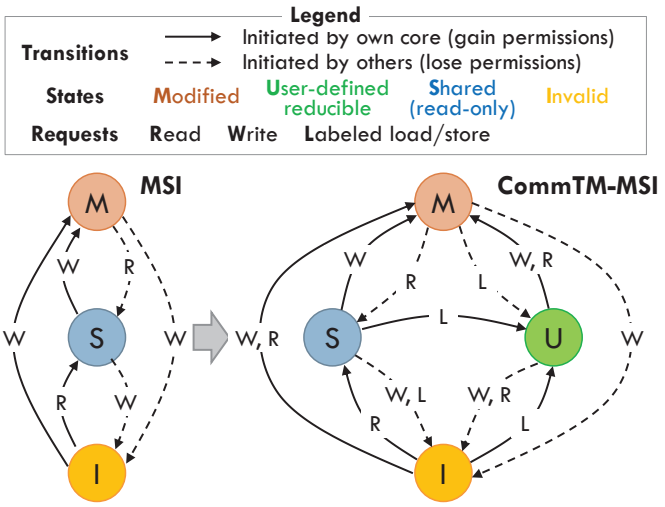


Fig. 3. State-transition diagrams of MSI and COMMTM protocols. For clarity, diagrams omit actions that do not cause a transition (e.g., R requests in S).

how COMMTM extends MSI with the U state. Lines enter U in response to labeled loads and stores, and leave U through reductions. Each U-state line is labeled with the type of reducible data it contains (e.g., ADD). Lines in U can satisfy loads and stores whose label matches the line's.

Other states in the original protocol retain similar functionality. For example, in Fig. 3, M can satisfy all memory requests (conventional and labeled), S can only satisfy conventional loads, and I cannot satisfy any requests. In the rest of the section we will show how lines transition among these states.

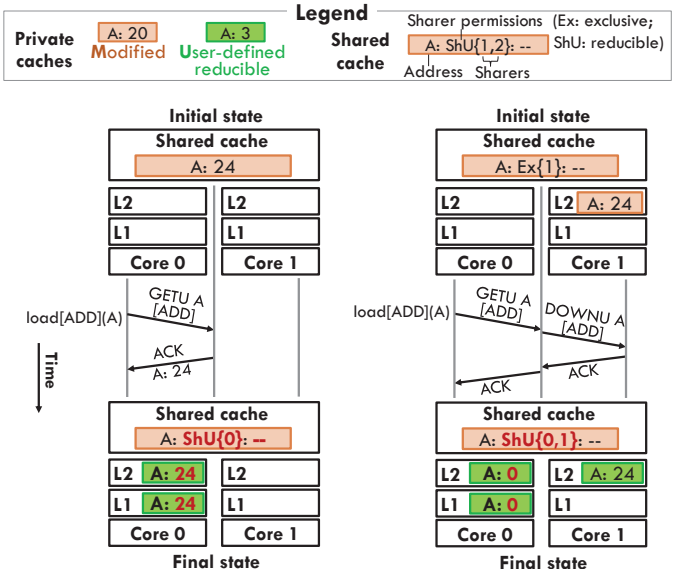
COMMTM's U state is similar to Coup's update-only state [54]. However, COMMTM requires substantially different support from Coup in nearly all other aspects: whereas Coup requires new update-only instructions for each commutative operation, COMMTM allows programs to implement arbitrary commutative operations, exploiting transactional memory to make them atomic; whereas Coup implements fixed-function reduction units, COMMTM allows arbitrary reduction functions; and whereas Coup focuses on reducing communication in a non-transactional context, COMMTM reduces both transactional conflicts and communication.

B.3. Transactional Execution

Labeled memory operations within transactions cause lines to enter the U state. We first discuss how state transitions work in the absence of transactional conflicts, then explain how conflict detection handles U-state lines.

On a labeled request to a line with invalid or read-only permissions, the cache issues a GETU request and receives the line in U. There are five possible cases:

1) If no other private cache has the line, the directory serves the data directly, as shown in Fig. 4a.
2) If there are one or more sharers in S, the directory invalidates them, then serves the data.
3) If there are one or more sharers in U with a different label from the request's, the directory asks them to forward the data to the requesting core, which performs a reduction to produce the data. Reductions are discussed in Sec. III-B4.



(a) No other sharers (b) Downgrade from M to U

Fig. 4. Serving labeled memory accesses: (a) the first GETU requester obtains the data; and (b) another cache with the line in M is downgraded to U and retains the data, while the requester initializes the line with the identity value. Each diagram shows the initial and final states in the shared and private caches.

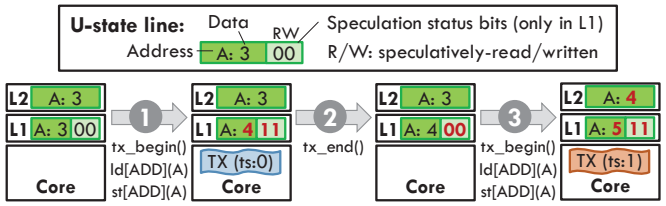
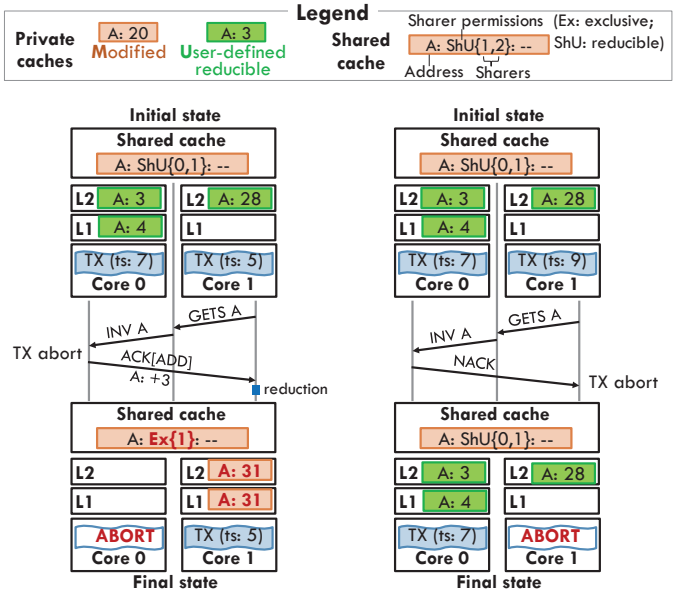


Fig. 5. Value management for U-state lines is similar to that of M-state lines. L1 tag bits record whether the line is speculatively read or written (using the state and label to infer whether from labeled or unlabeled instructions). Upon commit, spec-R/W bits are reset to zero. Before being written by another transaction, dirty U-state lines are written back to the L2.

4) If there are one or more sharers in U with the same label, the directory grants U permission, but does not serve any data.
5) If there is an exclusive sharer in M, the directory downgrades that line to U and grants U to the requester without serving any data, as shown in Fig. 4b.

In cases 1–3, the requester receives both U permission and the data; in cases 4 and 5, the requester does not receive any data, and instead initializes its local line with user-defined identity elements (e.g., zeros for ADD). Labeled operations must be aware that data may be scattered across multiple caches. In all cases, COMMTM preserves a key invariant: reducing the private versions of the line produces the right value.

Speculative value management: Value management for lines in U that are modified is nearly identical to that of lines in M. Fig. 5 shows how a line in U is read, modified, and, in the absence of conflicts, committed: ① Both normal and labeled writes are buffered in the L1 cache, and non-speculative values are stored in the private L2. ② When the transaction commits, all dirty lines in the L1 are marked non-speculative. ③ Before a dirty line in the L1 is speculatively written by a new transaction, its value is forwarded to the L2. Thus, if the transaction is



(a) Invalidation request from lower-timestamp transaction

(b) Invalidation request from higher-timestamp transaction

Fig. 6. Invalidations to lines in the labeled set cause a conflict. In this example, core 0 receives an invalidation request to a U-state line in its transaction's labeled set: (a) if requester has a lower timestamp, abort and forward data; and (b) if requester has a higher timestamp, NACK invalidation.

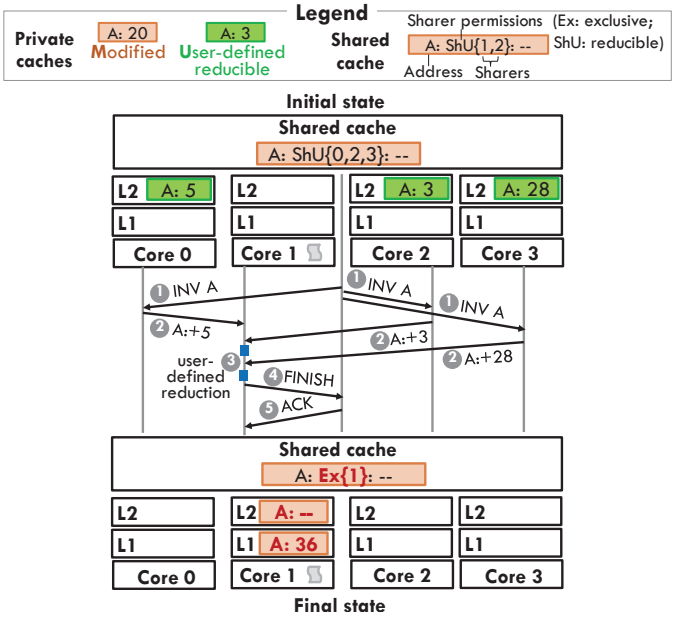


Fig. 7. Reductions are triggered by non-commutative requests. In this example, core 1 issues an unlabeled or differently-labeled request, causing a full reduction of line A's U-state data, held in several private caches.

aborted, its speculative updates to data in both M and U can be safely discarded, as the L2 contains the correct value.

Conflict detection and resolution: COMMTM leverages the coherence protocol to detect conflicts. In our baseline, conflicts are triggered by invalidation and downgrade requests to lines read or modified by the current transaction (i.e., lines in the transaction's read- or write-sets). Similarly, in COMMTM, invalidations to lines that have received a labeled operation from the current transaction trigger a conflict. We call this set of lines the transaction's *labeled set*. We leverage the existing L1 status bits to track the labeled set, as shown in Fig. 5.

COMMTM is orthogonal to the conflict resolution protocol. We leverage the timestamp-based approach of our baseline HTM: each transaction is assigned a unique timestamp that is included in all memory requests. On an invalidation to a line in the transaction's read, write, *or labeled set*, the core compares its transaction's timestamp and the requester's. If the receiving transaction is younger (i.e., has a higher timestamp), it honors the invalidation request and aborts; if it is older than the requester, it replies with a `NACK`, which causes the requester to abort. Fig. 6 shows both cases for a line in the labeled set.

B.4. Reductions

COMMTM performs reductions transparently to satisfy non-commutative requests. There is a wide range of implementation choices for reductions, as well as important considerations for deadlock avoidance.

We choose to perform each reduction at the core that issues the reduction-triggering request. Specifically, each core features a *shadow hardware thread* dedicated to perform reductions. Each core has a small (e.g., 2-entry) buffer to hold lines waiting to be reduced. The head entry of the buffer is memory-mapped to a fixed physical page, and this page is mapped (read-only) to the shadow thread's address space. Fig. 7 shows the steps of a reduction: ① When the directory receives a reduction-triggering request, it sends invalidation requests to all the cores with U-state permissions. ② Each of the cores receiving the invalidation forwards the line to the requester. ③ When each forwarded line arrives at the requester, the shadow thread runs the reduction handler, which merges it with the current line (if the requester does not have the line in U yet, it transitions to U on the first forwarded line it receives). ④ After all lines have been received and reduced, the requester transitions to M, ④ notifies the directory, and ⑤ serves the original request.

Dedicating a hardware thread to reductions ensures that they are performed quickly, but adds implementation cost. Alternatively, COMMTM could perform reductions by interrupting the requesting thread, e.g., through user-level interrupts [28, 44, 52]. Because reductions are not transactional, this would require implementing transaction pause/unpause [31, 55].

NACKed reductions: When a reduction happens to a line that has been speculatively updated by a transaction, the core receiving the invalidation may NACK the request, as shown in Fig. 6b. In this case, the requesting core still reduces the values it receives, but aborts its transaction afterwards, retaining its data in the U state. When reexecuted, the transaction will retry the reduction, and will eventually succeed thanks to timestamp-based conflict resolution.

For simplicity, non-speculative requests have no timestamp and cannot be NACKed. Finally, even though the request they seek to serve may come from a transaction, *reductions are not speculative*: reduction handlers always operate on non-speculative data and have no atomicity guarantees. Transactional reductions would be more complex, and they are

unnecessary in all the use cases we study (Secs. VI and VII).

Deadlock avoidance: Because the memory request that triggers the reduction blocks until the reduction is done, and reduction handlers may themselves issue memory accesses, there are subtle corner cases that may lead to deadlock and must be addressed. First, as mentioned in Sec. III-A, we enforce that reduction handlers cannot trigger reductions themselves (this restriction is easy to satisfy in all the reduction handlers we study). Second, to prevent reductions from causing protocol deadlocks, we dedicate an extra virtual network for forwarded U-state data. This adds moderate buffering requirements to on-chip network routers [35], which must already support 3-6 virtual networks in conventional protocols [7, 33, 46]. Third, we reserve a way in all cache levels for data with permissions other than U. Misses from reductions always fill data in that way, which ensures that they will not evict data in U, which would necessitate a reduction.

With these provisos, memory accesses caused by reductions cannot cause a cyclic dependence with the access they are blocking, avoiding deadlock. Note that both the corner cases and the deadlock-avoidance strategies we adopt are similar to those in architectures that support active messages, where these topics are well studied [2, 28, 44, 50] (a forward response triggered by a reduction is similar to an active message).

Handling unlabeled operations to speculatively-modified labeled data: Finally, CommTM must handle a transaction that accesses the same data through labeled and unlabeled operations (e.g., it first adds a value to a shared counter, and then reads it). Suppose that an unlabeled access to data in U causes a reduction (i.e., if the core's U-state line was not the only one in the system). If the data was speculatively modified by our own transaction, we cannot simply incorporate this data to the reduction, as the transaction may abort, leaving CommTM unable to reconstruct the non-speculative value of the data. For simplicity, in this case we abort the transaction and perform the reduction with the non-speculative state, re-fetched from the core's L2. When restarted, labeled loads and stores are performed as conventional loads and stores, so the transaction does not encounter this case again. Though we could avoid this abort through more sophisticated schemes (e.g., performing speculative and non-speculative reductions), we do not observe this behavior in any of our use cases.

B.5. Evictions

Evictions of lines in U from private caches are handled as follows: if no other private caches have U permissions for the line apart from the one that initiates the eviction, the directory treats this as a normal dirty writeback. When there are other sharers, the directory forwards the data to one of the sharers, chosen at random, which reduces it with its local line.

If the chosen core is performing a transaction that touches this data, for simplicity, the transaction is aborted.

Finally, evictions of lines in U from the shared cache cause a reduction at one of the cores sharing the line. Since the last-level cache is inclusive, this eviction aborts all transactions that have accessed the line.

C. Putting it all Together: Overheads

In summary, our CommTM implementation introduces moderate hardware overheads:

- Labeled load and store instructions in ISA and cores.
- Cache at all levels need to store per-tag label bits. Supporting eight labels requires 3 bits/line, introducing 0.6% area overhead for caches with 64-byte lines.
- Extended coherence protocol and cache controllers. While we have not verified CommTM's protocol extensions, they are similar to Coup's, which has reasonable verification complexity (by merging S and U, Coup requires no extra stable states and only 1–5 transient states [54]).
- One extra virtual network for forwarded U data, which adds few KBs of router buffers across the system [15].
- One shadow hardware thread per core to perform reductions. In principle, this is the most expensive addition (an extra thread increases core area by about 5% [22]). However, commercial processors already support multiple hardware threads, and the shadow thread can be used as a normal thread if the application does not benefit from CommTM.

D. Generalizing CommTM

CommTM can be applied to other contexts beyond our particular implementation.

Other protocols: While we have used MSI for simplicity, CommTM can easily extend other invalidation-based protocols, such as MESI or MOESI, with the U state [54]. In fact, we use and extend MESI in our evaluation.

Virtualizing labels: Large applications with many data types may have more semantically-commutative operations than hardware has labels. With moderate toolchain support, programmers should be able to define and use as many commutative operations as they need. At link time, the linker can map these operations to a small number of labels. Multiple operations may share the same label under two conditions. First, it should not be possible for both commutative operations to access the same data. There are many cases where this is naturally guaranteed, for instance, on operations on different types (e.g., insertions into sets and lists). Second, U-state lines need to have enough information (e.g., the data structure's type) to allow reduction handlers to perform the right operation. If too many labels are still needed, it is always safe to turn labeled loads and stores into unlabeled ones (e.g., using profile-guided optimization to preserve the most profitable labels).

Lazy conflict detection: While we focus on eager conflict detection, CommTM applies to HTMs with lazy (commit-time) conflict detection, such as TCC [12, 19] or Bulk [11, 37]. This would simply require acquiring lines in S or U without restrictions (triggering non-speculative reductions if needed, but without flagging conflicts), buffering speculative updates (both commutative and non-commutative), and making them public when the transaction commits. Commits would then abort all executing transactions with non-commutative updates. For example, a transaction that triggers a reduction and then commits would abort all transactions that accessed the line

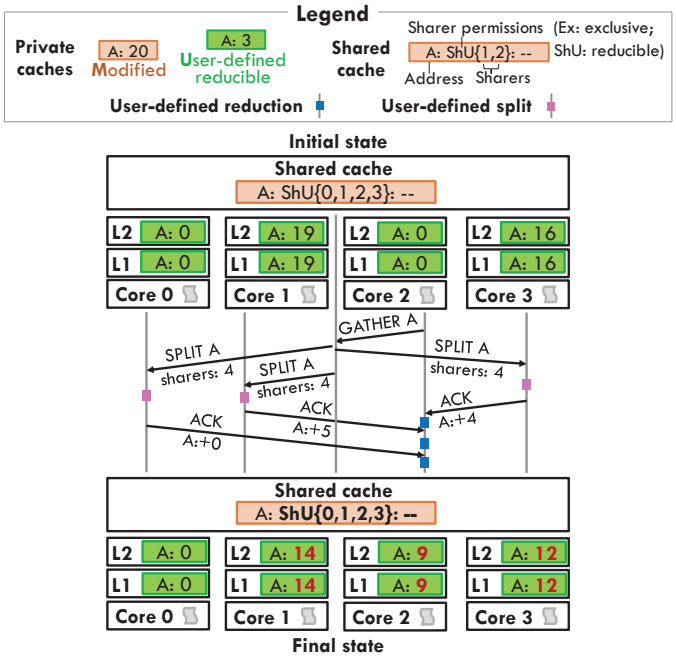


Fig. 8. Gather requests collect and reduce U-state data from other caches. In this example, core 2 issues a gather request to satisfy a decrement operation. User-defined splitters at other cores donate part of their local deltas to core 2. For instance, core 3 splits its initial value, 16, into 12, which it retains, and 4, which it donates.

while in U, but transactions that read and update the line while in U would not abort each other.

Other contexts: CommTM's techniques could be used in contexts beyond TM that require speculative execution of atomic regions, such as architectural support for implicit parallelism [18, 24, 49] or deterministic multithreading [17].

E. CommTM vs Semantic Locking

Just as eager conflict detection is the hardware counterpart to two-phase locking [5, 21], CommTM as described so far is the hardware counterpart to semantic locking (Sec. II-A). In semantic locking, each lock has a number of modes, and transactions try to acquire the lock in a given mode. Multiple transactions can acquire the lock in the same mode, accessing and updating the data it protects concurrently [25] (with some other synchronization to arbitrate low-level accesses, e.g., logging updates and performing reductions later). An attempt to acquire the lock in a different mode triggers a conflict. Each label in CommTM can be seen as a lock mode, and just like reads and writes implicitly acquire read and write locks to lines, labeled accesses implicitly acquire locks in the mode specified by their label, triggering conflicts if needed. Besides reducing conflicts, CommTM also reduces communication by buffering commutative updates in private caches.

IV. Avoiding Needless Reductions with Gather Requests

While semantic locking is general, not all commutative operations are amenable to semantic locking, and more sophisticated conflict detectors allow more operations to commute [25]. Similarly, we now extend CommTM to allow more concurrency than semantic locking. The key idea is that many operations are *conditionally commutative*: they only commute when the reducible data they operate on meets some conditions. With CommTM as presented so far, these conditions require normal reads, resulting in frequent reductions that limit concurrency. To solve this problem, we introduce *gather requests*, which allow moving partial updates to the same data across different private caches *without leaving the reducible state*.

Motivation: Consider a *bounded non-negative counter* that supports `increment` and `decrement` operations. `increment` always succeeds, but `decrement` fails when the initial value of the counter is already zero. `increment` always commutes, but `decrement` commutes only if the counter has a positive value. Bounded counters have many use cases, such as reference counting and resizable data structures.

In CommTM, we can exploit the fact that if the local value is positive, the global value must be positive. In this case, `decrement` can safely decrement the local value. However, if the local value is zero, `decrement` must perform a reduction to check whether the value has reached zero, as shown in this implementation:

```
bool decrement(int* counter) {
  tx_begin();
  int value = load[ADD](counter);
  if (value == 0) {
    // Trigger a reduction
    value = load(counter);
    if (value == 0) {
      tx_end();
      return false;
    }
  }
  store[ADD](counter, value - 1);
  tx_end();
  return true;
}
```

With frequent decrements, reductions will serialize execution even when the actual value of the counter is far greater than zero. Gather requests avoid this by allowing programs to observe partial updates in other caches and redistribute them without leaving U.

Gather requests: Fig. 8 depicts the steps of a gather request in detail. Gather requests are initiated by a new instruction, `load_gather`, which is similar to a labeled load. If the requester's line is in U, `load_gather` issues a gather request to the directory and reduces forwarded data from other sharers before returning the value.

The directory forwards the gather request to each (U-state) sharer. The core executes a *user-defined splitter*, a function analogous to a reduction handler that inspects its local value and sends a part of it to the requester. In our implementation, the directory forwards the number of sharers in gather requests, which splitters can use to rebalance the data appropriately.

Splitters reuse all the machinery of reduction handlers: they run on the shadow thread, are non-speculative, and split requests may trigger conflicts if their address was speculatively accessed.

Our bounded counter example can use gather requests as follows. First, we modify the `decrement` operation to use

load_gather:

```
bool decrement(int* counter) {
  tx_begin();
  int value = load[ADD](counter);
  if (value == 0) {
    value = load_gather[ADD](counter);
    if (value == 0) {
      value = load(counter);
      if (value == 0) {
        tx_end();
        return false;
      }
    }
  }
  store[ADD](counter, value - 1);
  tx_end();
  return true;
}
```

Second, we implement a user-defined splitter that gives a fraction $1/numSharers$ of its counter values, which, over time, will maintain a balanced distribution of values:

```
void add_split(int* counterLine, int* fwdLine,
               int numSharers) {
  for (int i = 0; i < intsPerCacheLine; i++) {
    int value = load[ADD](counterLine[i]);
    int donation = ceil(value / numSharers);
    fwdLine[i] = donation;
    store[ADD](counterLine[i], value - donation);
  }
}
```

Fig. 8 shows how a gather request rebalances counter values and allows a `decrement` operation to proceed while maintaining lines in U. Note how, after the gather request, the requester's local value (9) allows it to perform successive `decrement`s locally. In general, we observe that, although gather requests incur global traffic and may cause conflicts, they are rare, so their cost is amortized across multiple operations.

There are many ways to make gather operations more expressive. For example, we could enhance `load_gather` to query a subset of sharers, or to provide user-defined arguments to splitters. However, we have not found a need for these mechanisms for the operations we evaluate. We leave an in-depth exploration of these and other mechanisms to enhance CommTM's precision to future work.

V. Experimental Methodology

We perform microarchitectural, execution-driven simulation using zsim [43]. We evaluate a 16-tile chip with 128 simple cores and a three-level memory hierarchy, shown in Fig. 2, with parameters given in Table I. Each core has private L1s and a private L2, and all cores share a banked L3 cache with an in-cache directory.

We compare the baseline HTM and CommTM. Both HTMs use Intel TSX [53] as the programming interface, but do not use the software fallback path, which the conflict resolution protocol makes unnecessary. We add encodings for `labeled_load`, `labeled_store`, and `load_gather`, with labels embedded in the instructions.

We evaluate CommTM under microbenchmarks (Sec. VI) and full TM applications (Sec. VII). We run each benchmark to completion, and report results for its parallel region. To achieve statistically significant results, we introduce small amounts of non-determinism [3], and perform enough runs to achieve 95% confidence intervals $\leq 1\%$ on all results.

TABLE I
CONFIGURATION OF THE SIMULATED SYSTEM.

Cores	128 cores, x86-64 ISA, 2.4 GHz, IPC-1 except on L1 misses
L1 caches	32 KB, private per-core, 8-way set-associative, split D/I
L2 caches	128 KB, private per-core, 8-way set-associative, inclusive, 6-cycle latency
L3 cache	64 MB, fully shared, 16 4 MB banks, 16-way set-associative, inclusive, 15-cycle bank latency, in-cache directory
Coherence	MESI/CommTM, 64 B lines, no silent drops
NoC	4×4 mesh, 2-cycle routers, 1-cycle 256-bit links
Main mem	4 controllers, 136-cycle latency

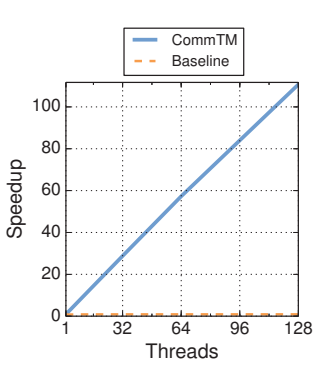


Fig. 9. Speedup of counter microbenchmark.

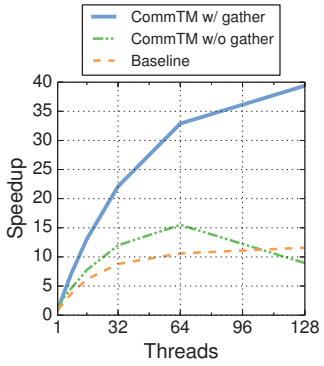


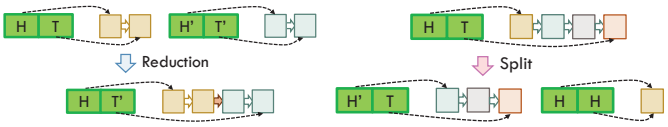
Fig. 10. Speedup of reference-counting microbenchmark.

VI. CommTM on Microbenchmarks

We use microbenchmarks to explore CommTM's capabilities and its impact on update-heavy operations.

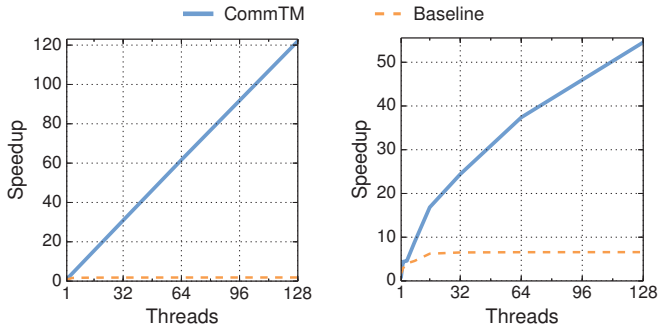
Counter increments: In this microbenchmark, threads perform 10 million increments to a single counter, implemented as presented in Sec. III. Fig. 9 shows that CommTM achieves linear scalability, while the baseline HTM serializes all transactions. While counters are our simplest use case, prior work reports that counter updates are a major cause of aborts in real applications [14, 42].

Reference counting: We implement a reference counter using the non-negative bounded counter described in Sec. IV, with and without gather requests. Threads acquire and release 1 million references in total, incrementing and decrementing 16 counters. Each thread starts with three references to each object and holds up to ten references. On every iteration, a thread selects a random object and increments or decrements its reference count probabilistically. The probability to increment the counter decreases linearly with the number of references the thread holds to the object, from 1.0 with no references to 0.0 with 10 references. Fig. 10 shows that CommTM without gather requests provides some speedup over the baseline TM with a few threads, but frequent reductions caused by threads having zero in their U-state line result in serialized transactions. By contrast, CommTM with gather requests scales to 39× at 128 threads. The sub-linear scalability is due to more frequent gather requests and splits at high thread counts.



(a) Reducing a list descriptor (b) Splitting a list descriptor

Fig. 11. A linked-list descriptor contains its head and tail pointers, and can be shared in U state by multiple caches. Each U-state copy represents a partial linked list. A reduction merges all partial lists and generates the resulting descriptor, and a split divides the partial list descriptor into two: one with the head element, which is donated, and the other with all other elements.



(a) 100% enqueues (b) 50% enqueues, 50% dequeues

Fig. 12. Speedup of linked list microbenchmark.

Linked lists: In this microbenchmark, threads enqueue and dequeue elements from a singly-linked list. When order is unimportant (e.g., if the list is used as a set, a hash table bucket, or a work-sharing queue), these operations are semantically (but not strictly) commutative. Fig. 11a shows how COMMTM makes these operations concurrent. Only the *descriptor* of a linked list, which contains its head and tail pointers, is accessed with labeled loads and stores (accesses to elements use normal loads and stores). This way, threads enqueue/dequeue elements to their local, reducible linked-list descriptors. Fig. 11a shows how the user-defined reduction handler merges two linked-list descriptors. Dequeues use `load_gather` if their local descriptor is empty, and each splitter donates the head element of its local list, as shown in Fig. 11b.

Fig. 12 compares the baseline HTM and COMMTM. In the baseline HTM, to avoid false sharing, head and tail pointers are allocated on different cache lines. Threads perform 10 million operations: all enqueues in Fig. 12a, or 50% enqueues and 50% dequeues (randomly interleaved) in Fig. 12b. The baseline HTM scales poorly in both cases, while COMMTM scales near-linearly on enqueues, and by 55× on mixed enqueues/dequeues (limited again by frequent gathers).

Ordered puts: Ordered puts or priority updates are frequent in databases [34] and are key in challenging parallel algorithms [47]. This semantically-commutative operation replaces an existing key-value pair with a new input pair if the new pair has a lower key. In COMMTM, we simply access the key-value pair with a labeled accesses, and define a reduction handler that merges key-value pairs by keeping the lowest one. Threads perform 10 million ordered puts using randomly-generated 64-bit keys and values. These fit within a cache line, but arbitrarily large key-value pairs are possible by using indirection (i.e., keeping pointers to the key and value in the reducible line). Fig. 13 shows that COMMTM scales near-linearly, while the baseline is 3.8× slower (in this case, the baseline scales to 31× because only smaller keys cause conflicting writes).

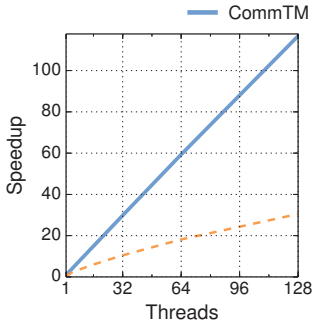


Fig. 13. Speedup of ordered put microbenchmark.

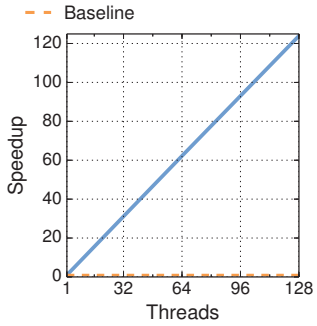


Fig. 14. Speedup of top-K insertion microbenchmark.

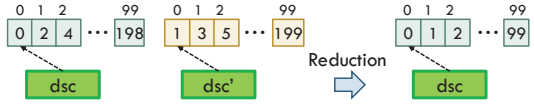


Fig. 15. Reduction of a top-K set descriptor with $K = 100$.

Top-K sets: A *top-K set*, common in databases, retains the K highest elements of a set [34]. We implement top-K sets similarly to linked lists: a descriptor contains a pointer to the top-K data (stored as a heap), and only the accesses to the descriptor use labeled loads and stores. Threads build up local top-K heaps, and reads trigger a reduction that merges all local heaps, as shown in Fig. 15.

Fig. 14 shows the performance of inserting 10 million elements to a top-1000 set. While the baseline HTM suffers significant serialization introduced by unnecessary read-write dependencies, COMMTM scales top-K set insertions linearly, yielding a 124× speedup at 128 threads.

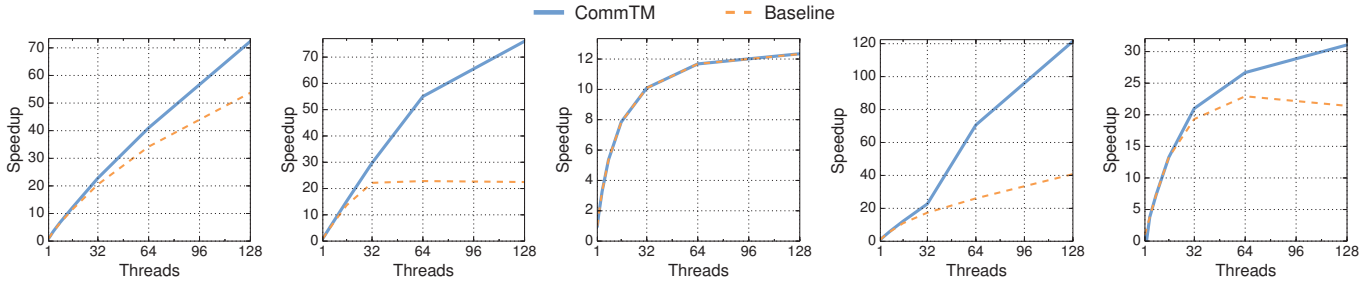
VII. COMMTM ON FULL APPLICATIONS

We evaluate COMMTM on five TM benchmarks: `boruvka` [25], which we implement from scratch, and `genome`, `kmeans`, `ssca2`, and `vacation`, which we adapt from STAMP [29]. Table II details their input sets and main characteristics. `boruvka` computes the minimum spanning tree of a graph. It utilizes several commutative operations: `OPUT` to record the minimum-weight edges connecting separate graph components, `MIN` to union two components, `MAX` to mark edges added to the minimum spanning tree, and `ADD` to calculate the weight of the resulting tree. `kmeans` performs commutative additions to shared cluster centroids. `ssca2` spends little time in commutative updates to shared, global graph metadata. Like Blundell el al. [8], we compile `genome` and `vacation` with resizable hash tables, which use conditionally-commutative updates to a bounded counter to determine when to resize.

Fig. 16 compares the performance and scalability of COMMTM and the baseline HTM. Each graph shows the speedups of the baseline HTM and COMMTM for a single application from 1 to 128 threads (x-axis). As before, all speedups are relative to the single-thread runtime in the baseline

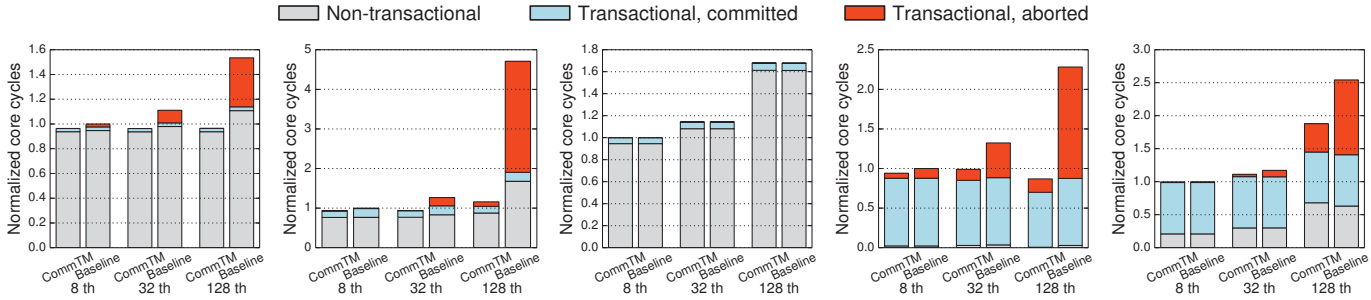
TABLE II
BENCHMARK CHARACTERISTICS.

	Input set	Uses gather?	Commutative operations
boruvka	usroads [16]	✗	Updating min-weight edges (64b-key `OPUT`); Unioning components (64b `MIN`); Marking edges (64b `MAX`); Calculating weight of MST (64b `ADD`)
kmeans	-m15 -n15 -t0.05 -i random-n16384-d24-c16 [29]	✗	Updating cluster centers(32b `ADD`, 32b `FP ADD`)
ssca2	-s16 -i1.0 -u1.0 -l9 -p9 [29]	✗	Modifying global information for a graph (32b `ADD`)
genome	-g4096 -s64 -n640000 [29]	✓	Remaining-space counter of a resizable hash table (bounded 64b `ADD`)
vacation	-n4 -q60 -u90 -r32768 -t8192 [29]	✓	Remaining-space counter of a resizable hash table (bounded 64b `ADD`)



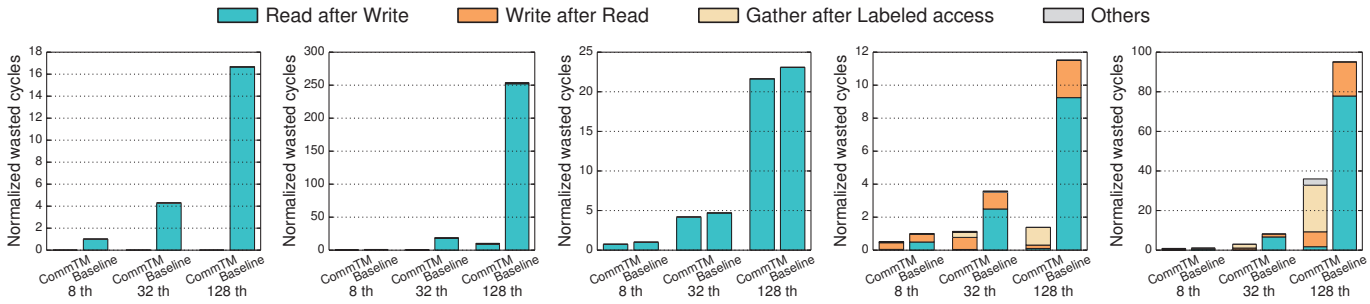
(a) boruvka (b) kmeans (c) ssca2 (d) genome (e) vacation

Fig. 16. Per-application speedups of COMMTM and baseline HTM on 1–128 threads (higher is better).



(a) boruvka (b) kmeans (c) ssca2 (d) genome (e) vacation

Fig. 17. Breakdown of core cycles for COMMTM and baseline HTM for 8, 32, and 128 threads (lower is better).



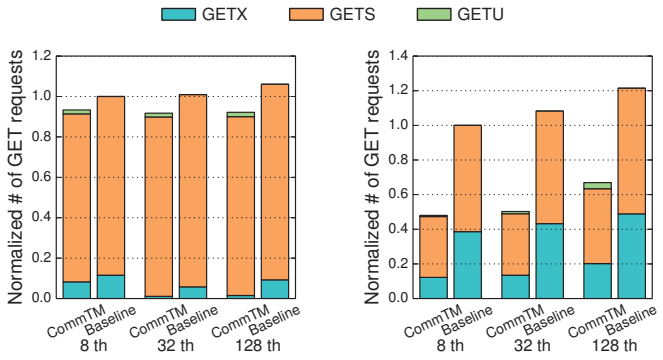
(a) boruvka (b) kmeans (c) ssca2 (d) genome (e) vacation

Fig. 18. Breakdown of wasted cycles for COMMTM and baseline HTM for 8, 32, and 128 threads (lower is better).

HTM. Fig. 16 shows that COMMTM always outperforms the baseline HTM, often significantly. At 128 threads, COMMTM outperforms the baseline by 35% on `boruvka`, 3.4× on `kmeans`, 0.2% on `ssca2`, 3.0× on `genome`, and 45% on `vacation`. Moreover, the gap between the baseline HTM and COMMTM often widens as the number of threads grows.

COMMTM is especially beneficial on update-heavy applications. For instance, `kmeans` introduces a large number of commutative updates within transactions. With conventional HTMs, these updates must be serialized. Thus, as the number of threads increases, serialized updates bottleneck the whole application. By contrast, COMMTM makes these updates local and concurrent, achieving significant speedup. COMMTM yields negligible improvements on applications that update shared data rarely, like `ssca2`.

Fig. 17 gives more insight into these results by showing the breakdown of cycles spent by all threads for each application. Each cycle is either non-transactional or transactional, and transactional cycles are divided into useful (committed) and wasted (aborted) cycles. Each graph shows the breakdown of cycles for both COMMTM and the baseline HTM on 8, 32, and 128 threads for a single application. Cycles are normalized



(a) boruvka (b) kmeans

Fig. 19. Breakdown of total number of GET requests between L2s and L3 for COMMTM and conventional HTM on 8, 32 and 128 threads (lower is better).

to the baseline's at 8 threads. Lower bars are better.

Fig. 17 shows that COMMTM substantially reduces wasted transactional cycles. At 128 threads, COMMTM reduces wasted cycles over the baseline by 25× on kmeans, 6.6% on ssca2, 8.3× on genome, and 2.6× on vacation. In boruvka, COMMTM eliminates all wasted transactional cycles. Fig. 17 also explains why COMMTM barely helps ssca2: contention is rare and therefore only a small fraction of cycles are spent on aborted transactions.

Fig. 18 further details the cause of wasted cycles. In the baseline HTM, wasted cycles are almost always caused by read-after-write dependency violations. For applications with ample semantic commutativity, such as boruvka and kmeans, most of these dependencies are superfluous and COMMTM avoids them entirely.

Beyond improving concurrency, COMMTM also reduces traffic, as applications with significant data reuse benefit substantially from buffering updates in private caches. Fig. 19 shows the breakdown of GET requests between the L2s and L3 for boruvka and kmeans, the two applications with a significant reduction in traffic. At 128 threads, COMMTM reduces L3 GET requests by 13% on boruvka and 45% on kmeans. This also explains why non-transactional cycles are lower in Fig. 17 (15% lower on boruvka and 48% on kmeans).

Finally, though COMMTM improves performance significantly, labeled memory operations are relatively rare. At 128 threads, the fraction of all labeled instructions, including loads, stores, and gathers, over all executed instructions are 0.13% on boruvka, 1.2% on kmeans, $5.9 \cdot 10^{-7}$ on ssca2, 0.042% on genome, and 0.057% on vacation. Though rare, their impact is substantial: on conventional HTMs, these operations cause conflicts that abort whole transactions, which include many other instructions, wasting a large amount of work.

VIII. Additional Related Work

Prior work in HTM has proposed a wide set of techniques to reduce the number of conflicts and their impact. These techniques are orthogonal to COMMTM, as they do not leverage commutativity, and detect conflicts through reads and writes.

Several HTMs, such as DATM [39], SONTM [5], Wait-n-GoTM [23], and OmniOrder [38], reduce aborts by letting transactions continue execution after they conflict and trying to commit them in the order imposed by the data dependence that caused the conflict. These designs improve performance when dependences are acyclic, but semantically-commutative updates often consist of read-modify-write chains that cause cyclic dependencies, which conflict-serializable HTMs must treat as conflicts. However, COMMTM avoids these conflicts.

SI-TM [27] relaxes serializability and implements snapshot isolation, which only flags write-write dependences as conflicts. SI-TM, like other schemes that weaken serializability [1, 48], can allow more concurrency on reads and writes to the same data but requires programs to be rewritten to work under a less intuitive concurrency model. SI-TM also relies on an expensive multiversioned main memory. Finally, SI-TM also cannot handle conflicting read-modify-write operations, which cause write-write conflicts (e.g., unlike COMMTM, SI-TM bottlenecks on kmeans [27]).

Other techniques focus on reducing the cost of mispeculation. ReSlice [45] reexecutes only the conflicting load and its dependent instructions, and RetCon [8] performs symbolic reexecution of simple, conflicting auxiliary updates (e.g., updates to shared counters that are not used elsewhere in the transaction). Unlike these schemes, COMMTM does not trigger conflicts to begin with, avoiding superfluous communication and serialization. COMMTM is also much cheaper than ReSlice and allows a broader range of non-peripheral operations than RetCon, such as enqueues and top-K insertions.

Finally, open-nested transactions [31, 32] can provide some of the benefits of commutativity. Unlike conventional (closed) nested transactions, which remain speculative until their parent commits, open-nested transactions commit when they end, and specify an abort handler to undo their effects if their parent later aborts. Open-nested transactions make their parents less vulnerable, but they still suffer from conflicts and serialization. By contrast, COMMTM supports concurrent and communication-free updates to the same data. Moreover, open nesting is practical only when operations are easy to undo, which is not always the case (e.g., top-K in Sec. VI).

IX. Conclusion

We have presented COMMTM, an HTM that exploits semantic commutativity to avoid conflicts that limit scalability in prior HTMs. COMMTM extends the coherence protocol and conflict detection scheme to allow multiple cores to perform user-defined commutative operations concurrently and without conflicts. COMMTM preserves transactional guarantees: COMMTM triggers reductions when non-commutative operations access the same data as commutative ones, so they never observe any partial state. COMMTM's basic scheme allows as much concurrency as semantic locking. Gather requests allow COMMTM to reduce conflicts even further.

We have shown that COMMTM bridges the precision-overhead dichotomy of hardware vs software conflict detection: COMMTM scales many operations that serialize in conventional HTMs, such as set insertions, reference counting, and top-K insertions, while retaining the low overhead of HTMs. As

a result, at 128 cores, COMMTM outperforms an eager-lazy HTM by up to 3.4× and reduces or even eliminates aborts.

Finally, beyond our specific implementation, a key contribution of our work is to recognize that hardware speculation can also benefit from conflict-detection techniques that have traditionally been considered software-only. Prior work has developed a rich set of conflict detectors that go beyond COMMTM's current capabilities. It would be interesting to see how many of these techniques can also be easily adapted by hardware. We leave this exploration to future work.

ACKNOWLEDGMENTS

We sincerely thank Maleen Abeydeera, Nathan Beckmann, Joel Emer, Mark Jeffrey, Harshad Kasture, Anurag Mukkara, Suvinay Subramanian, Po-An Tsai, and the anonymous reviewers for their helpful feedback. We are grateful to Heiner Litz for sharing his transactional memory implementation on zsim [27]. We thank to Colin Blundell and Milo Martin for sharing their RetCon benchmarks [8]. This work was supported in part by C-FAR, one of six SRC STARnet centers by MARCO and DARPA, and by NSF grant CAREER-1452994.

REFERENCES

[1] A. Adya, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson *et al.*, "The MIT Alewife machine: architecture and performance," in *ISCA-22*, 1995.

[3] A. R. Alameldeen and D. A. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, no. 4, 2006.

[4] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *HPCA-11*, 2005.

[5] U. Aydonat and T. S. Abdelrahman, "Hardware support for relaxed concurrency control in transactional memory," in *MICRO-43*, 2010.

[6] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi *et al.*, "Coordination avoidance in database systems," *VLDB*, vol. 8, no. 3, 2014.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt *et al.*, "The gem5 simulator," *ACM Comp. Arch. News*, vol. 39, no. 2, 2011.

[8] C. Blundell, A. Raghavan, and M. M. Martin, "RETCON: transactional repair without replay," in *ISCA-37*, 2010.

[9] J. Bobba, K. E. Moore, H. Volos, L. Yen *et al.*, "Performance pathologies in hardware transactional memory," in *ISCA-34*, 2007.

[10] C. Cascaval, C. Blundell, M. Michael, H. W. Cain *et al.*, "Software transactional memory: Why is it only a research toy?" *ACM Queue*, vol. 6, no. 5, 2008.

[11] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *ISCA-33*, 2006.

[12] H. Chafi, J. Casper, B. Carlstrom, A. McDonald *et al.*, "A scalable, non-blocking approach to transactional memory," in *HPCA-13*, 2007.

[13] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," in *SOSP-24*, 2013.

[14] C. Click, "Azul's experiences with hardware transactional memory," in *Transactional Memory Workshop*, 2009.

[15] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[16] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011.

[17] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: deterministic shared memory multiprocessing," in *ASPLOS-XIV*, 2009.

[18] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *ASPLOS-VIII*, 1998.

[19] L. Hammond, V. Wong, M. Chen, B. Carlstrom *et al.*, "Transactional memory coherence and consistency," in *ISCA-31*, 2004.

[20] M. Herlihy and E. Koskinen, "Transactional boosting: a methodology for highly-concurrent transactional objects," in *PPoPP*, 2008.

[21] M. D. Hill, "Is transactional memory an oxymoron?" *VLDB*, vol. 1, no. 1, 2008.

[22] G. Hinton, D. Sager, M. Upton, D. Boggs *et al.*, "The microarchitecture of the Pentium® 4 processor," in *Intel Technology Journal*, 2001.

[23] S. Jafri, G. Voskuilen, and T. Vijaykumar, "Wait-n-GoTM: Improving HTM performance by serializing cyclic dependencies," in *ASPLOS-XVIII*, 2013.

[24] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *MICRO-48*, 2015.

[25] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali, "Exploiting the commutativity lattice," in *PLDI*, 2011.

[26] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan *et al.*, "Optimistic parallelism requires abstractions," in *PLDI*, 2007.

[27] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson, "SI-TM: reducing transactional memory abort rates through snapshot isolation," in *ASPLOS-XIX*, 2014.

[28] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee *et al.*, "Exploiting two-case delivery for fast protected messaging," in *HPCA-4*, 1998.

[29] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC*, 2008.

[30] K. Moore, J. Bobba, M. Moravan, M. D. Hill, and D. A. Wood, "LogTM: log-based transactional memory." in *HPCA-12*, 2006.

[31] M. Moravan, J. Bobba, K. Moore, L. Yen *et al.*, "Supporting nested transactional memory in LogTM," in *ASPLOS-XII*, 2006.

[32] J. E. B. Moss, "Open nested transactions: Semantics and support," in *WMPI*, 2006.

[33] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb, "The Alpha 21364 network architecture," in *Hot Interconnects*, 2001.

[34] N. Narula, C. Cutler, E. Kohler, and R. Morris, "Phase reconciliation for contended in-memory transactions," in *OSDI-11*, 2014.

[35] L.-S. Peh and W. J. Dally, "A delay model and speculative architecture for pipelined routers," in *HPCA-7*, 2001.

[36] P. Prabhu, S. Ghosh, Y. Zhang, N. Johnson, and D. August, "Commutative set: A language extension for implicit parallel programming," in *PLDI*, 2011.

[37] X. Qian, W. Ahn, and J. Torrellas, "ScalableBulk: Scalable cache coherence for atomic blocks in a lazy environment," in *MICRO-43*, 2010.

[38] X. Qian, B. Sahelices, and J. Torrellas, "OmniOrder: Directory-based conflict serialization of transactions," in *ISCA-41*, 2014.

[39] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *MICRO-41*, 2008.

[40] R. F. Resende, D. Agrawal, and A. El Abbadi, "Semantic locking in object-oriented database systems," in *OOPSLA*, 1994.

[41] M. C. Rinard and P. C. Diniz, "Commutativity analysis: A new analysis framework for parallelizing compilers," in *PLDI*, 1996.

[42] W. Ruan, T. Vyas, Y. Liu, and M. Spear, "Transactionalizing legacy code: An experience report using GCC and memcached," in *ASPLOS-XIX*, 2014.

[43] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA-40*, 2013.

[44] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *ASPLOS-XV*, 2010.

[45] S. R. Sarangi, J. Liu, Wei Torrellas, and Y. Zhou, "ReSlice: Selective re-execution of long-retired misspeculated instructions using forward slicing," in *MICRO-38*, 2005.

[46] K. S. Shim, M. Lis, M. H. Cho, I. Lebedev, and S. Devadas, "Design tradeoffs for simplicity and efficient verification in the Execution Migration Machine," in *ICCD*, 2013.

[47] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons, "Reducing contention through priority updates," in *SPAA*, 2013.

[48] T. Skare and C. Kozyrakis, "Early release: Friend or foe?" in *WTW*, 2006.

[49] G. S. Sohi, S. E. Breach, and T. Vijaykumar, "Multiscalar processors," in *ISCA-22*, 1995.

[50] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *ISCA-19*, 1992.

[51] W. E. Weihl, "Commutativity-based concurrency control for abstract data types," *IEEE Trans. Comput.*, vol. 37, no. 12, 1988.

[52] H. Wong, A. Bracy, E. Schuchman, T. Aamodt *et al.*, "Pangaea: a tightly-coupled IA32 heterogeneous chip multiprocessor," in *PACT-17*, 2008.

[53] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel® transactional synchronization extensions for high-performance computing," in *SC13*, 2013.

[54] G. Zhang, W. Horn, and D. Sanchez, "Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems," in *MICRO-48*, 2015.

[55] C. Zilles and L. Baugh, "Extending hardware transactional memory to support non-busy waiting and non-transactional actions," in *TRANSACT*, 2006.