Architectural Support for Commutativity in Hardware Speculation

by

Virginia Chiu

S.B., Massachusetts Institute of Technology (2016)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
Daniel Sanchez
Assistant Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Architectural Support for Commutativity in Hardware Speculation

by

Virginia Chiu

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Hardware speculative execution schemes (e.g., hardware transactional memory (HTM)) enjoy low run-time overheads but suffer from limited concurrency because they detect conflicts at the level of reads and writes. By contrast, software speculation schemes can reduce conflicts by exploiting that many operations on shared data are *semantically commutative*: they produce semantically equivalent results when reordered. However, software techniques often incur unacceptable run-time overheads.

To bridge this dichotomy, this thesis presents CommTM, an HTM that exploits semantic commutativity. CommTM extends the coherence protocol and conflict detection scheme to allow multiple cores to perform user-defined commutative operations to shared data concurrently and without conflicts. CommTM preserves transactional guarantees and can be applied to arbitrary HTMs.

This thesis details CommTM's implementation and presents its evaluation. The evaluation uses a series of microbenchmarks that covers commonly used operations and a suite of full transactional memory applications. We see that CommTM scales many operations that serialize on conventional HTMs, such as counter increments, priority updates, and top-K set insertions. As a result, at 128 cores on full applications, CommTM outperforms a conventional eager-lazy HTM by up to 3.4× and reduces or eliminates aborts.

Thesis Supervisor: Daniel Sanchez
Title: Assistant Professor

Acknowledgements

I worked closely with Guowei Zhang on the work presented in this thesis, and I am deeply grateful for all the help and guidance he provided. Guowei laid the foundation for this project; without him, none of this work would exist, and I'd probably still be banging my head against the wall figuring out how to use zsim!

I'd also like to thank Professor Sanchez, who advised Guowei and I throughout the project. His thoughtful and pragmatic advice kept us on track and drove our work forward. In all our meetings where I came in with a problem, I always left with a solution.

To my closest friends at MIT: thanks for the constant support, and for teaching me that I don't need to rough out everything alone.

To my mother and father: thanks for the unending love, the occasional nag, and for always believing in me!

Contents

List of Figures

List of Tables

Chapter 1

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 transactional memory, 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 [11]); 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. Thus, semantically commutative operations executed in concurrent transactions need not cause a conflict, improving concurrency. 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 actual order of elements in `s` does not matter (a set is an unordered data structure), both representations are semantically equivalent, and insertions into sets commute. Semantic commutativity is common in other contexts beyond this simple example [15, 24, 33, 40]. Semantic commutativity was first exploited in the 1980s [49], and is now common in databases [7, 33], parallelizing compilers [35, 40], and runtimes [24, 35] (Chapter 2).

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 understand reads and writes. 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 overhead [24, 25].

To bridge this dichotomy, this thesis presents CommTM, a commutativity-aware HTM (Chapter 3). 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-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 (Chapter 3)

that achieves the same precision as software *semantic locking* [24, 49]. We then extend CommTM with *gather requests* (Chapter 4), which allow software to redistribute reducible data among caches, achieving much higher concurrency in important use cases.

We evaluate CommTM with microbenchmarks that cover commonly used operations (Section 5.2) and full TM applications (Section 5.3). Microbenchmarks show that CommTM scales on a variety of commutative operations, such as priority updates, linked list enqueues, and top-K set insertions, which allow no concurrency in conventional HTMs. 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.

Chapter 2

Background

In this chapter, we discuss prior work in exploiting semantic commutativity and in commutativity-aware cache coherence.

2.1 Semantic Commutativity

Semantic commutativity is frequently used in software conflict detection schemes [24, 25, 33, 35, 39, 49]. Most work in this area focuses on techniques that reason about operations to abstract data types. Prior work has proposed a wide variety of conflict detection implementations [20, 24, 35, 39, 49]. Not all commutativity-aware conflict detection schemes are equally precise: simple and general-purpose techniques, such as semantic locking [24, 39, 49], flag some semantically-commutative operations as conflicts, while more sophisticated schemes, like gatekeepers [24], incur fewer conflicts but have higher overheads and are often specific to particular patterns.

In this work we focus on semantic locking [39, 49], 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 can 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 coalesce commutative updates. With gather requests (Chapter 4), CommTM allows more concurrency than semantic locking.

2.2 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 [52] 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 new states that do not trigger coherence actions on updates, avoiding conflicts. However, Coup does not work in a transactional context (only for single-instruction atomic 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 much 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.

Chapter 3

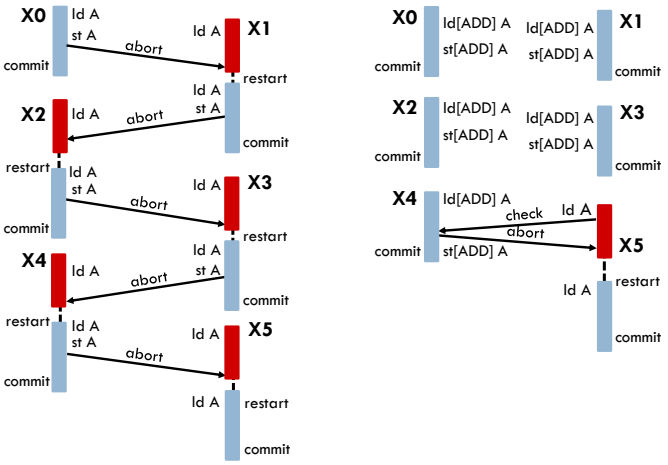
CommTM

We now present the CommTM commutativity-aware HTM. The key idea behind CommTM is to extend 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 are reordered 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 coherence protocols and HTM designs.

3.1 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*



(a) Conventional HTM (b) CommTM

Figure 3-1: Example comparing (a) a conventional HTM and (b) CommTM. Transactions X0–X4 increment a shared counter, and X5 reads it.

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 set insertions. Figure 3-1 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 multiplex these labels in Section 3.4. 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 memory accesses form a commutative operation, CommTM introduces labeled memory instruc-

tions. 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. Note that 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 [30]).

User-defined reductions: Finally, CommTM requires the program to specify a per-label reduction handler that merges reducible-state cache lines. This function takes the address of the cache line and the data from a reducible cache line 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 multi-instruction commutative operations done through labeled loads and stores, reduction handlers are *not transactional*. Moreover, to ease their implementation, we restrict the types of accesses they can make. Specifically, while reduction 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).

3.2 CommTM Implementation

3.2.1 Eager-Lazy HTM Baseline

To make our discussion concrete, we present CommTM in the context of a specific eager-lazy HTM baseline. We simulate an HTM with eager conflict detection and

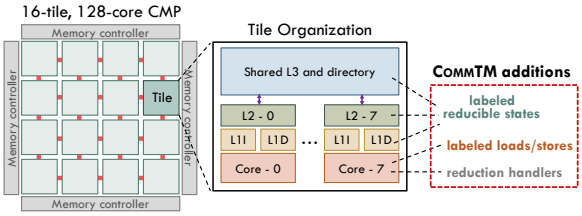


Figure 3-2: Baseline CMP and main CommTM additions.

lazy (buffer-based) version management, as in LTM [4] and Intel's TSX [51]. We assume a multicore system with per-core private L1s and L2s, and a shared L3, as shown in Figure 3-2. Cores buffer speculatively-updated data in the L1 cache; the L2 has non-speculative data only. Evicting the speculative data in L1s 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 [29]: 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 [10].

3.2.2 Coherence protocol

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

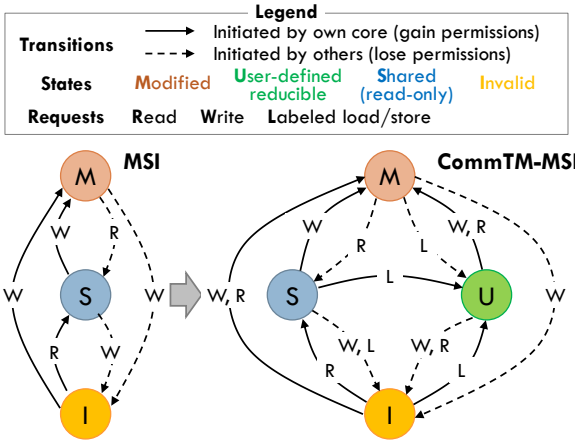


Figure 3-3: State-transition diagrams of MSI and CommTM protocols. For clarity, diagrams omit actions that do not cause a transition.

(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 Figure 3-3, 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 in detail.

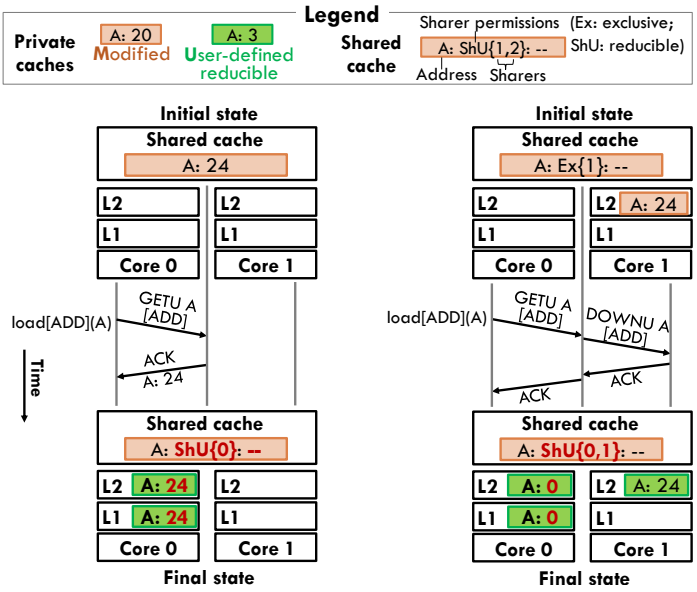
CommTM's U state is similar to Coup's update-only state [52]. However, CommTM requires substantially different support from Coup in nearly all other aspects: whereas Coup requires new update-only instructions for each commutative operation, CommTM allows programs to implement arbitrary commutative operations, 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.

3.2.3 Transactional execution

Labeled memory operations within transactions cause lines to enter the U state. We first discuss of permissions change in the absence of transactional conflicts, then explain how conflict detection changes.

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 Figure 3-4a.
2. If there are sharers in S, the directory invalidates them, then serves the data.
3. If there are 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 detail in Section 3.2.3.
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 Figure 3-4b.



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

Figure 3-4: 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.

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 the user-defined identity element (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. Figure 3-5 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 as 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 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 down-

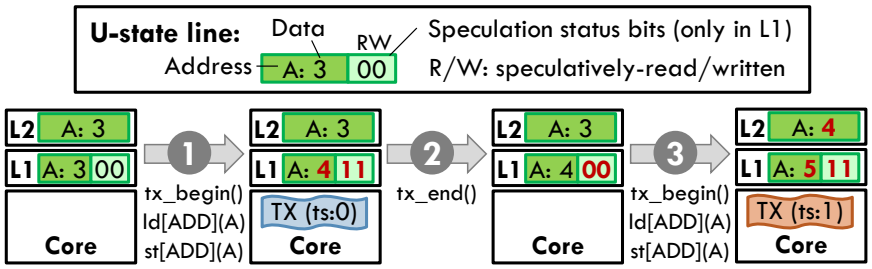
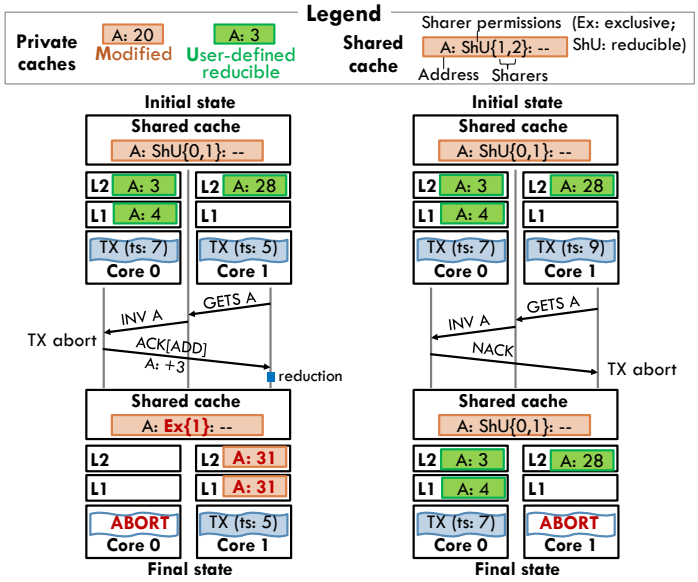


Figure 3-5: Value management for U-state lines is similar to M. L1 tag bits record whether the line is speculatively read or written (using the state and label to infer whether from labeled or unlabeled instructions). Upon commit, spec-R/W bits are reset to zero. Before being written by another transaction, dirty U-state lines are written back to the L2.



(a) INV from lower-timestamp transaction

(b) INV from higher-timestamp transaction

Figure 3-6: 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.

grade 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 transaction's *labeled set*. We leverage the existing L1's status bits to track the labeled set, as shown in Figure 3-5.

CommTM is orthogonal to the conflict resolution protocol. We leverage our baseline's timestamp-based resolution approach: each transaction is assigned an

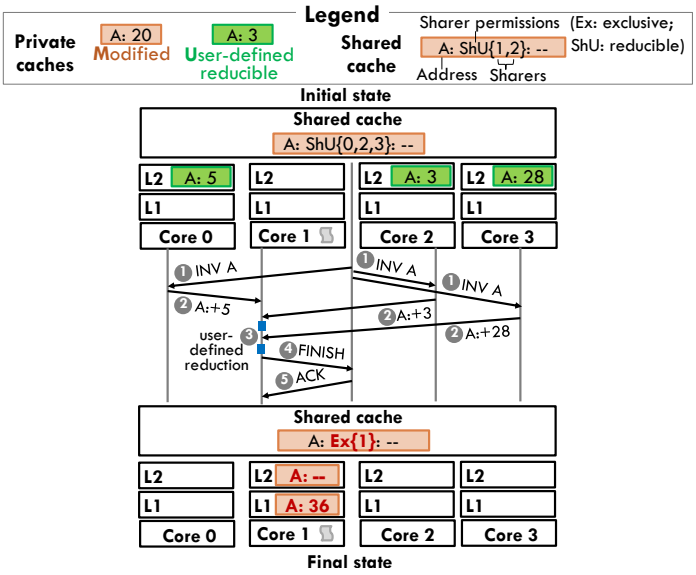


Figure 3-7: Core 0 issues unlabeled or differently-labeled request, causing a full reduction of A's U-state data, held in several private caches.

unique timestamp, and requests from each transaction include its timestamp. 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. Figure 3-6 shows both of these cases in detail for a line in the labeled set.

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 reductions at the core that issues the reduction-triggering request. Specifically, each core features a *shadow hardware thread* dedicated to perform reductions. Figure 3-7 shows the steps of a reduction in detail: 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 helper hardware context to reductions ensures that they are performed quickly, but adds implementation cost. Alternatively, we could handle reductions through user-level interrupts of the main thread [27, 43, 50], or use a low-performance helper core [5, 14].

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 Figure 3-6b. 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 re-executed, 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 within 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 (Section 5.2 and Section 5.3).

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 Section 3.1, we enforce that reduction handlers cannot trigger reductions themselves (this restriction is easy to satisfy in all the reduction handlers we study). Second, to avoid a protocol deadlock caused by reductions, we dedicate an extra virtual network for forwarded U-state data. This adds moderate buffering requirement to on-chip network routers [34], which must already support 3-6 virtual networks in conventional protocols [8, 32, 45]. 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. We should note that both the corner cases and the deadlock-avoidance strategies we adopt are similar to those in architectures with hardware support for active messages, where these topics are well studied [2, 27, 43, 48] (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.

3.2.4 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 write-back. 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.

3.3 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 (requiring only 1–5 transient states by merging S and U) [52].
- One extra virtual network for forwarded U data, which adds few KBs of router buffers across the system [17].
- 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.

3.4 Generalizing CommTM

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 [52]. In fact, we use and extend MESI in our evaluation.

Multiplexing labels: Large applications with many data types may have more semantically-commutative operations than hardware provides. In this case, we can assign the same label to two or more operations 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. This allows CommTM to scale to large applications with a small number of labels in hardware.

Lazy conflict detection: While we focus on eager conflict detection, CommTM applies to HTMs with lazy (commit-time) conflict detection, such as TCC [13, 19] or Bulk [12, 36]. This would simply require acquiring lines in S or U without restrictions (triggering non-speculative reductions if needed, but without flagging conflicts), holding speculative updates (both commutative and non-commutative), and making them public when the transaction commits. Commits 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 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 other contexts beyond TM where speculative execution is required, e.g., thread-level speculation.

3.5 CommTM vs Semantic Locking

Just as eager conflict detection is the hardware counterpart to two-phase locking [6, 21], CommTM as described so far is the hardware counterpart to semantic locking (Section 2.1). 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 [24]. An attempt to acquire the lock in a different mode triggers a conflict. Each label in CommTM can be seen as a locking mode, and just like reads and writes implicitly acquire read and write locks to the cache line, labeled accesses implicitly acquire the lock in the mode specified by the label, triggering conflicts if needed. Furthermore, CommTM is architected to reduce communication by holding commutative updates to the line in private caches.

Chapter 4

Avoiding Needless Reductions with Gather Requests

While semantic locking is general, not all semantically-commutative operations are amenable to semantic locking, and more sophisticated software conflict detectors allow more operations to commute [24]. 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*.

4.1 Motivation

Consider a *bounded non-negative counter* that supports `increment` and `decrement` operations. `increment` always succeeds, but `decrement` returns a failure when the initial value of the counter is already zero. `increment` always commutes, but `decrement` only commutes 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
    if (load(counter) == 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.

4.2 Gather Requests

Figure 4-1 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`:

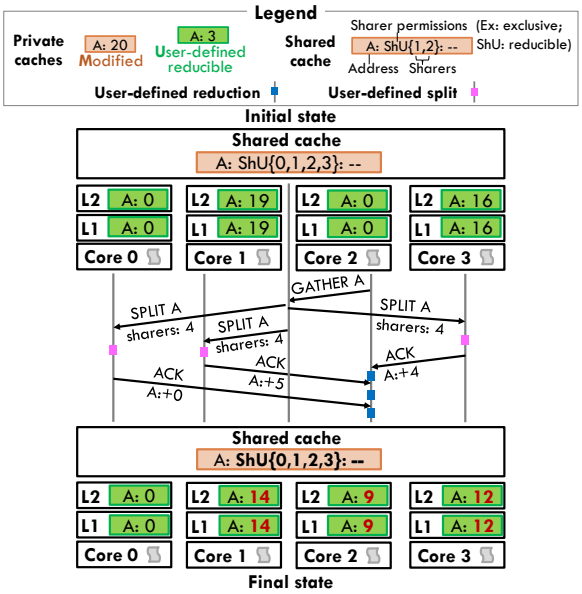


Figure 4-1: Gather requests collect and reduce U-state data from other caches. In this example, core 2 initiates a gather to satisfy a local decrement. User-defined splitters at other cores donate part of their local deltas to core 2.

```
bool decrement(int* counter) {
  tx_begin();
  int value = load[ADD](counter);
  if (value == 0) {
    value = load_gather[ADD](counter);
    if (value == 0)
      if (load(counter) == 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(v / numSharers);
    fwdLine[i] = donation;
    store[ADD](counterLine[i], v - donation);
  }
}
```

Figure 4-1 shows how a gather request rebalances the data 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 is a wide array of options to enhance the expressiveness of gather operations. 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.

Chapter 5

Evaluation

In this chapter, we discuss our evaluation of CommTM. We present the experimental methodology in Section 5.1, the microbenchmarks in Section 5.2, and the full TM applications in Section 5.3.

5.1 Experimental Methodology

We perform microarchitectural, execution-driven simulation using zsim [42]. We evaluate a 16-tile CMP with 128 simple cores and a three-level memory hierarchy, shown in Figure 3-2, with parameters given in Table 5.1. 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 [51] 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 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.

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

Table 5.1: Configuration of the simulated system.

5.2 CommTM on Microbenchmarks

We use microbenchmarks to explore CommTM's capabilities and its impact on update-heavy operations. We see that often the baseline HTM serializes transactions while CommTM achieves significant speedup.

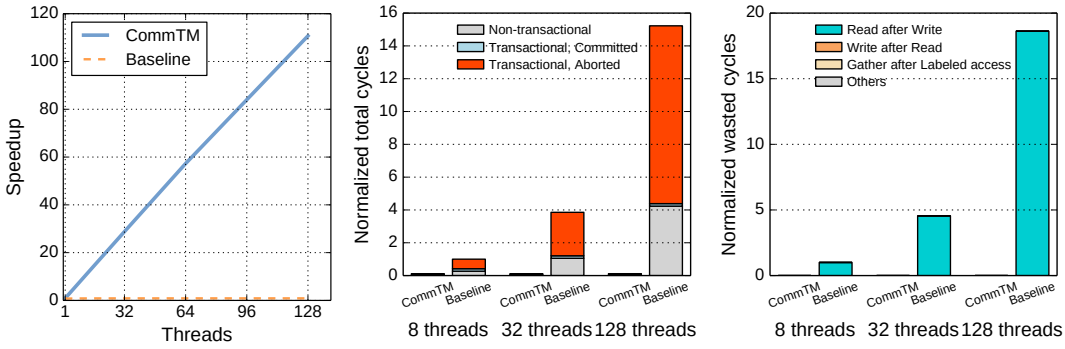
5.2.1 Counter Increments

In this microbenchmark, we evaluate the baseline HTM and CommTM's performance when issuing increments to a counter. Counters are our simplest case, but prior work reports that counter updates are a major cause of aborts in real applications [16, 41]. Incrementing a counter is a strictly commutative operation.

We implement a counter as presented in Chapter 3. On CommTM, we use labeled accesses tagged with `ADD` and define a reduction handler that combines all local updates when a normal load is issued. On the baseline HTM, threads update the counter with a read-modify-write sequence inside a transaction.

Threads perform a total of 10 million increments to a single counter, with work evenly distributed among all threads. Figure 5-1a shows the speedups of the baseline HTM and CommTM from 1–128 threads. Speedups are relative to the performance of a sequential execution in the baseline HTM. We see that while the baseline HTM achieves no speedup, CommTM achieves near-linear speedup, scaling to 110× at 128 threads.

Figure 5-1b and Figure 5-1c give more insight into this result. Figure 5-1b shows the



(a) Speedup (b) Breakdown of total cycles (c) Breakdown of wasted cycles

Figure 5-1: Counter increments microbenchmark results.

breakdown of total cycles spent by all threads. Each cycle is either non-transactional or transactional, and transactional cycles are divided into useful (committed) and wasted (aborted) cycles. The graph shows the breakdown of cycles for both CommTM and the baseline HTM on 8, 32, and 128 threads. Cycles are normalized to the baseline's at 8 threads. Lower bars are better. Figure 5-1c shows the cause of the aborted transactional cycles.

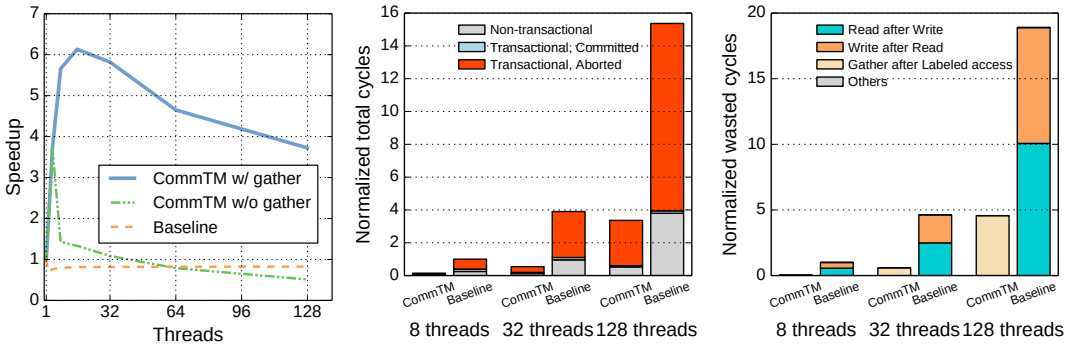
These figures show that the baseline HTM wastes most of its cycles in aborted transactions, and all of them are read after write aborts. CommTM removes the unnecessary dependencies and eliminates all wasted cycles.

5.2.2 Reference Counting

Reference counting is a technique used in garbage collection algorithms for storing the number of references, pointers, or handles to a resource. Increments and decrements to a reference counter are strictly commutative operations easily exploited by CommTM.

We implement a reference counter using the non-negative bounded counter described in Chapter 4, with and without gather requests. The reference counter supports two primitive operations: `increment` and `decrement`.

In the microbenchmark, threads acquire and release 1 million references in total, incrementing and decrementing the counter. Each thread starts with three references to the object and holds up to five references. Threads behave probabilistically: each thread increments the counter with probability 1.0, 0.7, 0.5, 0.5, 0.3, and 0.0 if it



(a) Speedup (b) Breakdown of total cycles (c) Breakdown of wasted cycles

Figure 5-2: Reference counting microbenchmark results.

holds 0, 1, 2, 3, 4, and 5 local references, respectively, and decrements it otherwise.

Figure 5-2a shows that the baseline HTM achieves no speedup, and CommTM without gather requests provides speedup at low thread counts, but at higher thread counts, frequent reductions result in serialized transactions. CommTM with gather requests scales to 6.1× at 16 threads and 3.7× at 128 threads. The decreased scalability at higher thread counts is due to more frequent gather requests and splits, as shown in Figure 5-2b and Figure 5-2c (aborts are all caused by gather after labeled access violations). Although CommTM does not achieve significant speedup here, this greatly improves on prior work: Coup scales to 64× on a similar reference counting microbenchmark using 1024 counters, whereas we report a peak of 6× speedup using only one counter [52].

5.2.3 Linked Lists

We use an unordered linked list microbenchmark to show how CommTM can exploit semantic commutativity. Unordered linked lists are used a number of applications, such as in set implementations, as hash table buckets, or as work-sharing queues.

Our linked list implementation is singly linked and allows two primitive operations: **enqueue** and **dequeue**. **enqueue** appends a node to the list and **dequeue** removes a node and returns it. When order is unimportant, these operations are semantically (but not strictly) commutative.

To allow commutative operations, we implement the linked list with a *descriptor*

that contains the linked list's head and tail pointers. We allocate the label `LIST`, and only the descriptor is accessed using labeled loads and stores (elements in the linked list are accessed using normal loads and stores). When multiple threads are operating on the same linked list, each thread holds a local, reducible descriptor that contains its own head and tail pointers; these pointers represent a segment of the whole linked list. Our implementation of the descriptor is shown below.

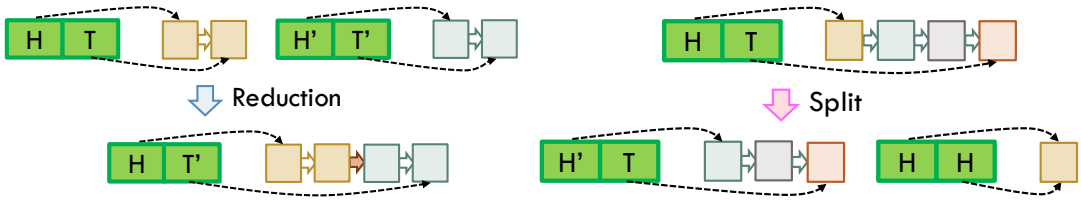
```
struct List {
    Node* head;
    Node* tail;
    int pad[6]; // Only one descriptor on the line
}__attribute__((__aligned__(64)));
```

Threads perform `enqueues` and `dequeues` on their local linked list descriptors. An `enqueue` requires no synchronization. If a thread tries to dequeue from an empty local segment, then the thread uses `load_gather` to take elements from other threads and rebuild a local list. We show our implementation of `enqueue` and `dequeue` below.

```
void enqueue(List* list, Node* newNode) {
    Node** tailAddr = getTailAddr(list);
    Node** headAddr = getHeadAddr(list);
    txn_begin();
    Node* tail = load[LIST](tailAddr);
    if (tail != NULL) tail->next = newNode;
    else store[LIST](headAddr, newNode);
    store[LIST](tailAddr, newNode);
    txn_end();
}
Node* dequeue(List* list) {
    Node** tailAddr = getTailAddr(list);
    Node** headAddr = getHeadAddr(list);
    txn_begin();
    Node* head = load[LIST](headAddr);
    if (head == NULL) {
        head = load_gather[LIST](headAddr);
        if (head == NULL) return NULL;
    }
    Node* newHead = head->next;
    store[LIST](headAddr, newHead);
    if (newHead == NULL) store[LIST](tailAddr, newHead);
    txn_end();
    return head;
}
```

Each splitter donates the head element of its local segment and our user-defined reduction handler merges their descriptors. These procedures are shown visually in Figure 5-3 and in code in Listing 5.1.

We compare the baseline HTM to CommTM on two workloads: in the first case threads perform only enqueues, in the second case threads perform 50% enqueues and 50% dequeues (randomly interleaved). In both cases, threads perform a total of 10 million operations, evenly distributed among all threads. In the baseline HTM, the



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

Figure 5-3: Reducing and splitting a linked list descriptor.

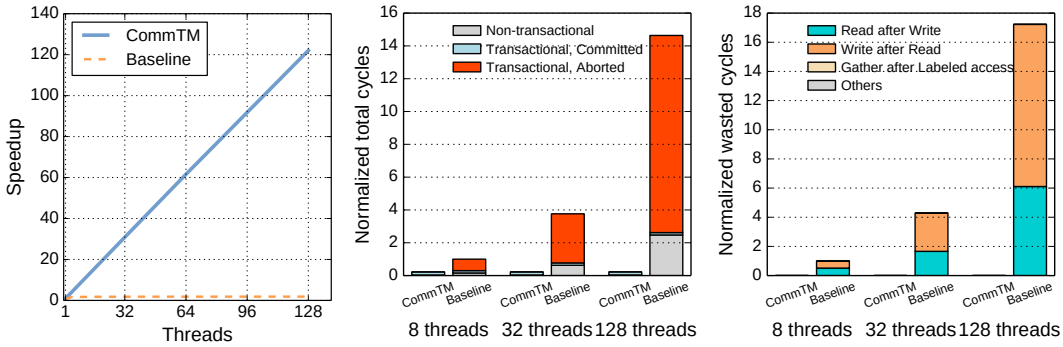
```
void list_reduce(List* list, List* delta) {
    Node** listTailAddr = getTailAddr(list);
    Node** listHeadAddr = getHeadAddr(list);
    Node* deltaTail = getTail(delta);
    Node* deltaHead = getHead(delta);

    if (deltatHead == null) break;
    Node* tail = load[LIST](listTailAddr);
    if (tail == null) store[LIST](headTailAddr, deltaHead);
    else tail->next = deltaHead;
    store[LIST](listTailAddr, deltaTail);
}


List* list_split(List* list) {
    List* donation;
    Node** tailAddr = getTailAddr(list);
    Node** headAddr = getHeadAddr(list);
    Node* head = load[LIST](headAddr);

    donation.setHead(head);
    donation.setTail(head);
    if (head != NULL) {
        Node* newHead = head->next;
        head->next = NULL;
        store[LIST](headAddr, newHead);
        if (newHead == NULL)
        store[LIST](tailAddr, newHead);
    }

    return donation;
}
```

Listing 5.1: Linked list reduction handler and splitter implementations.

head and tail pointers are allocated on different cache lines to avoid false sharing.

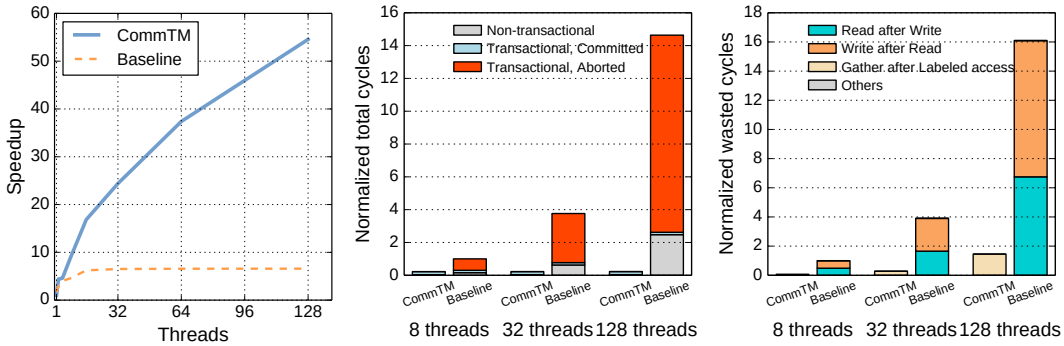
Figure 5-4 shows the result for 100% enqueues. In this case, the baseline HTM serializes all transactions because of contention on the tail pointer. On the other hand, CommTM avoids this problem and scales near linearly.

Figure 5-5 shows the results for 50% enqueues and 50% dequeues. Once again, the baseline sees poor scaling. However, the baseline HTM performs better on mixed enqueues/dequeues than on 100% enqueues because a dequeue on an empty list is simply a read operation. CommTM scales to 54× at 128 threads. Like the reference counting microbenchmark, the scaling at higher thread counts is limited by frequent gather requests and splits at higher thread counts. These gather requests are caused



(a) Speedup (b) Breakdown of total cycles (c) Breakdown of wasted cycles% dequeues

Figure 5-4: 100% enqueues microbenchmark results.



(a) Speedup (b) Breakdown of total cycles (c) Breakdown of wasted cycles

Figure 5-5: 50% enqueues 50% dequeues microbenchmark results.

by threads performing dequeues on a local empty linked list segment. Regardless, in both cases, CommTM significantly outperforms the baseline HTM.

5.2.4 Ordered Puts

Ordered put or priority update is a primitive operation frequent in databases [33] and key in challenging parallel algorithms [46]. 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 store the key-value pair in a cache line along with a boolean value that indicates if a valid key-value pair is present.

```
struct Pair {
    bool valid;
    int key;
    int value;
    int pad[5]; // Only one pair on the line
}__attribute__((__aligned__(64)));
```

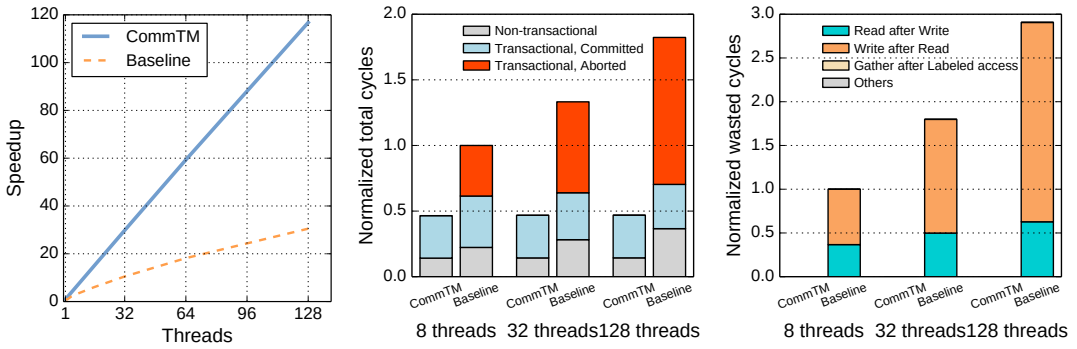
We access this line with labeled loads and stores tagged with `OPUT`, and define a

reduction handler that merges key-value pairs by keeping the lowest one. We show our implementation of the ordered put operation and the reduction handler below.

```
void ordered_put(Pair* pair, int newKey, int newValue) {
    bool* validAddr = getValidAddr(pair);
    int* keyAddr = getKeyAddr(pair);
    int* valueAddr = getValueAddr(pair);
    txn_begin();
    bool valid = load[ODPT](validAddr);
    int key = load[ODPT](keyAddr);
    if (key > newKey || !valid) {
        store[ODPT](keyAddr, newKey);
        store[ODPT](valueAddr, newValue);
    }
    txn_end();
}
void odpt_reduce(Pair* pair, Pair* delta) {
    bool* validAddr = getValidAddr(pair);
    int* keyAddr = getKeyAddr(pair);
    int* valueAddr = getValueAddr(pair);
    bool newValid = getValid(delta);
    int newKey = getKey(delta);
    int newValue = getValue(delta);

    int valid = load[ODPT](valueAddr);
    int key = load[ODPT](keyAddr);

    if (newValid) {
        if (!valid || valid && key > newKey) {
            store[ODPT](validAddr, newValid);
            store[ODPT](keyAddr, newKey);
            store[ODPT](valueAddr, newValue);
        }
    }
}
```

In the microbenchmark, threads perform a combined total of 10 million ordered puts, evenly distributed among all threads. The key-value pairs are 64-bit keys and values, and the keys gradually decrease. These fit within a cache line, but arbitrarily large key-value pairs are possible by using indirection.

Figure 5-6a shows that CommTM scales near-linearly, while the baseline is 3.8× slower (in this case, the baseline scales to 30× because only smaller keys cause



(a) Speedup (b) Breakdown of total cycles (c) Breakdown of wasted cycles

Figure 5-6: Ordered put microbenchmark results.

conflicting writes). Figure 5-6b and Figure 5-6c illustrates that CommTM drastically improves performance by eliminating all the unnecessary read-write dependency violations.

5.2.5 Top-K Set Insertions

A *top-K set*, common in databases, contains the K highest elements of a set [33]. Insertions to a top-K set are semantically commutative: although the elements may be inserted in a different order, the final result is semantically equivalent.

As we did with linked lists, we implement a top-K set using a reducible descriptor. The descriptor contains a pointer to the top-K data (stored as a heap) and a size field that tracks the number of elements in the top-K set.

```
struct TopKSet {
    uint size;
    int* elements;
    int pad[6]; //Ensure only the descriptor is on the cache line
}__attribute__((__aligned__(64)));
```

Threads build up local top-K heaps, and reads trigger a reduction that merges all local heaps, as shown in Figure 5-7 and in the below code.

```
void topkset_insertion(TopKSet* topk, int elt, uint k) {
    int** elementsAddr = getElementsAddr(topk);
    uint* sizeAddr = getSizeAddr(topk);
    txn_begin();
    int* elements = load[TOPK](elementsAddr);
    uint size = load[TOPK](sizeAddr);

    if (elements == NULL) { // Must create a new local heap
        elements = (int*) malloc(sizeof(int) * k+1);
        elements[0] = k;
        store[TOPK](elementsAddr, elements);
    }

    if (size < k) {
        heapInsert(elements, elt);
        store[TOPK](sizeAddr, size + 1);
    } else {
        int smallest = heapGetMin(elements);
        if (elt < smallest) {
            heapReplaceMin(elements, elt);
        }
    }
    txn_end();
}
```

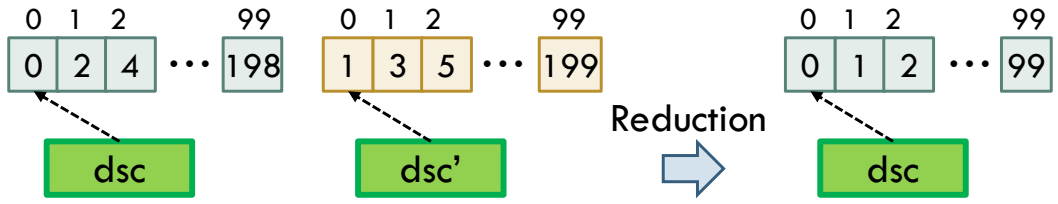
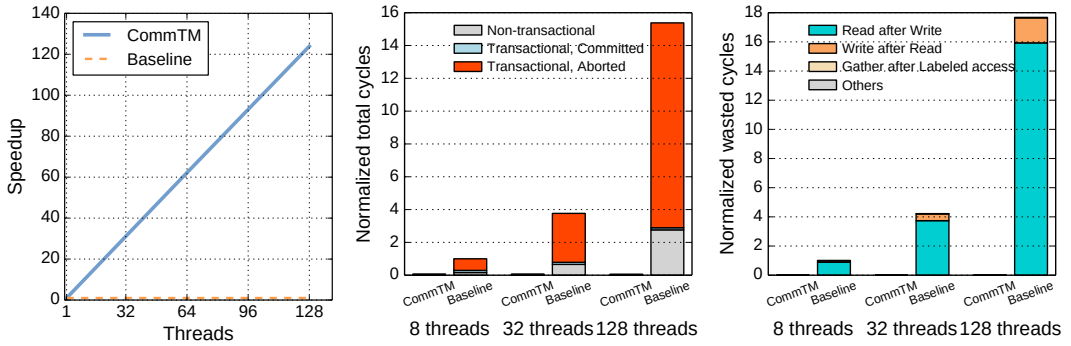


Figure 5-7: A top-K set descriptor with $K = 100$ (omitting implementation details).

```
void topk_reduce(TopKSet* topkset, TopKSet* delta) {
    uint* sizeAddr = getSizeAddr(topkset);
    int** elementsAddr = getElementsAddr(topkset);
    uint deltaSize = getSize(delta);
    int* deltaElements = getElements(delta);

        uint size = load[TOPK](sizeAddr);
        int* elements = load[TOPK](elements);
        uint k = elements[0];

        for (uint i = 1; i <= deltaElements; i++) {
            elt = deltaElements[i];
            if (size < k) {
                heapInsert(elements, elt);
            } else {
                int smallest = heapGetMin(elements);
                if (elt < smallest) {
                    heapReplaceMin(elements, elt);
                }
            }
        }
        store[TOPK](sizeAddr, size);
}
```



(a) Speedup (b) Breakdown of total cycles (c) Breakdown of wasted cycles

Figure 5-8: Top-K set insertions microbenchmark results.

Figure 5-8 shows the performance and cycle breakdowns 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 123× speedup at 128 threads.

	Input set	Uses gather?	Commutative operations
boruvka	usroads [18]	✗	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 [28]	✗	Updating cluster centers(32b `ADD`, 32b `FP ADD`)
ssca2	-s16 -i1.0 -u1.0 -l9 -p9 [28]	✗	Modifying global information for a graph (32b `ADD`)
genome	-g4096 -s64 -n640000 [28]	✓	Remaining-space counter of a resizable hash table (bounded 64b `ADD`)
vacation	-n4 -q60 -u90 -r32768 -t8192 [28]	✓	Remaining-space counter of a resizable hash table (bounded 64b `ADD`)

Table 5.2: Benchmark characteristics.

5.3 CommTM on Full Applications

We evaluate CommTM on several TM benchmarks: `boruvka` [24], and `genome`, `kmeans`, `ssca2`, and `vacation` from STAMP [28]. We discuss `boruvka` in Section 5.3.1, detail the STAMP benchmarks in Section 5.3.2, and present the results in Section 5.3.3. Table 5.2 summarizes the input sets and main characteristics of the benchmarks.

5.3.1 Boruvka's Algorithm

`boruvka` is an unordered algorithm that computes the minimum spanning tree of a connected undirected graph and the minimum spanning forest of a disconnected undirected graph. It computes this by applying successive *edge-contractions* to the smallest-weight edges that connect two separate *components*. These edges are added to the minimum spanning tree. A component is a group of vertices that are connected by edges added to the minimum spanning tree so far, and an edge-contraction connects two components to form a new component. The algorithm begins with each node as a singular component and ends after $\log_2 v$ iterations, where v is the number of vertices in the graph.

In most implementations, edge-contractions are managed using a *union-find* data structure. This data structure keeps track of a set of elements partitioned into a number of disjoint subsets. Each subset is represented by a fixed element, called the *representative*. The data structure supports the following two operations:

1. `find`: given an element, returns the representative of the subset to which the element belongs

2. `union`: given two elements, join their respective subsets.

We use the union-find data structure in `boruvka` by having the set of elements be the graph vertices and subsets be separate components. Below we give the code for a sequential implementation of `boruvka`.

```
int boruvka(Graph* g) {
    UnionFind uf = g->numVertices;
    for (int i = 1; i < g->numVertices; i = i + i) {
        for (int vertex = 0; vertex < g->numVertices; vertex++) {
            int minEdgeWeight = INT_MAX;
            bool addEdge = false;
            int edge, rep1, rep2;
            for (int j = 0; j < g->getOutDegree(vertex); j++) {
                int e = g->getEdge(vertex, j);
                if (g->getWeight(e) < minEdgeWeight) {
                    rep1 = uf.find(getSrc(e));
                    rep2 = uf.find(getDst(e));
                    if (rep1 != rep2) {
                        edge = e;
                        addEdge = true;
                    }
                }
            }
            if (addEdge) {
                g->mark(edge);
                uf.union(rep1, rep2);
            }
        }
    }

    int MSTweight = 0;
    for (int edge = 0; edge < g->numEdges; edge++) {
        if g->isMarked(edge)
            MSTweight += g->getWeight(edge);
    }
    return MSTweight;
}
```

There is ample opportunity to exploit semantic commutativity. The first opportunity is in edge selection: in each iteration we select the smallest weight edges; thus we can use `OPUT` (ordered puts) to perform the selection. The second opportunity is in edge-contraction. The union-find data structure has complex commutativity conditions [24], but if we give each vertex a unique integer label and use the vertex with the smallest label as the representative for its component, then a `union` can be performed as a `MIN` (a minimum operation). In our implementation, we also use `MAX` (a maximum operation) and `ADD` to mark edges added to the minimum spanning tree and to calculate the weight of the tree, respectively.

In order to exploit semantic commutativity, we must make modifications to the sequential implementation presented above. The sequential implementation has many interleaved reads and writes to union-find data. To decouple these reads and writes, we split them into two blocks separated by a barrier. The first block of the algorithm executes `find` operations, and the second executes `union` operations. Additionally,

interleaved reads and writes are inherent in the `union` operation: a `union` between two subsets requires reading their representatives. We eliminate this dependency by guaranteeing that we only pass representatives to `union`. Finally, we introduce an array that keeps track of the smallest weight edge that touches a given component. At the end of the edge-selection block in an iteration, the edges contained in this array are added to the tree.

Listing 5.2 shows the final implementation for `boruvka` on CommTM. With this implementation of `boruvka`, CommTM is able to achieve 35% speedup compared to the baseline HTM. We will further discuss this result in Section 5.3.3.

```
int boruvka(Graph* g) {
    UnionFind uf = g->numVertices;
    // keep track of edges we add in each iteration
    Pair** closest = new Pair*[g->numVertices];

    // threads operates on separate chunks of the graph in parallel
    int tid = thread_getId();
    int numThreads = thread_getNumThreads();
    int myStart = tid * g->numVertices / numThread;
    int myEnd = (tid + 1) * g->numVertices / numThread;

    for (int i = 1; i < g->numVertices; i = i + i) {
        for (int j = myStart; j < myEnd; j++)
            invalidatePair(closest[j]);

        thread_barrier();

        for (int vertex = myStart; vertex < myEnd; vertex++) {
            for (int j = 0; j < g->getOutDegree(vertex); j++) {
                int e = g->getEdge(vertex, j);
                rep1 = uf.find(getSrc(e));
                rep2 = uf.find(getDst(e));
                if (rep1 == rep2) continue;

                int edgeWeight = g->getWeight(e);
                int* data = {e, rep1, rep2};

                txn_begin();
                ordered_put(closest[rep1], e, data); // OPUT
                txn_end();

                txn_begin();
                ordered_put(closest[rep2], e, data); // OPUT
                txn_end();
            }
        }
        thread_barrier();

        for (int component = myStart; component < myEnd; component++) {
            if (!isValid(closest[component]) continue;
            int edge, rep1, rep2 = closest[component]->value;

            txn_begin();
            g->mark(edge); // MAX
            txn_end();

            txn_begin();
            uf.union(rep1, rep2); // MIN
            txn_end();
        }
        thread_barrier();
    }

    int MSTweight = 0;
    myStart = tid * g->numEdges / numThread;
    myEnd = (tid + 1) * g->numEdges / numThread;
    for (int edge = myStart; edge < myEnd; edge++) {
        if g->isMarked(edge) {
            txn_begin();
            add(&MSTweight, g->getWeight(edge)); // ADD
            txn_end();
        }
    }
    return MSTweight;
}
```

Listing 5.2: Final implementation of Boruvka's algorithm.

5.3.2 STAMP Applications

STAMP is a benchmark suite for transactional memory systems. We have selected the following applications from the suite to which CommTM can be applied: `kmeans`, `ssca2`, `genome`, and `vacation`. For each, we describe the algorithms and where commutavity can be exploited.

`kmeans` is a data mining application. The K-means clustering algorithm partitions objects in N-dimensional space into K clusters. The STAMP implementation has each thread process a partition of the objects iteratively. In the TM version, transactions are used to protect updates of shared cluster centroids. The length of transactions is short, the time spent in transactions is small, and read/write sets are small. The amount of contention depends on the value of K, as fewer clusters means that threads are more likely to be updating the same shared cluster centroid. The updates to cluster centroids are additions; we use CommTM to exploit the available commutativity.

`ssca2` is a scientific application that constructs an efficient graph representations using adjacency arrays and auxiliary arrays. It is commonly used in applications ranging from computational biology to security. The TM version has threads adding nodes to the graph in parallel and uses transactions to protect access to the adjacency arrays. Like `kmeans`, `ssca2` spends little time in short transactions and has small read/write sets. Contention is also low because a large number of graph nodes leads to infrequent concurrent updates of the same adjacency list. We apply CommTM to perform commutative additions to shared, global graph metadata.

`genome` is a bioinformatics application that performs gene sequencing. It has medium length transactions, medium read/write sets, and spends most if its time in transactions. In the first phase of the algorithm, it utilizes a hash table to create a set of unique DNA segments. We apply CommTM here by compiling `genome` with a resizable hash table (similar to Blundell el al. [9]). We do this by replacing the size field in the hash table with a bounded non-negative counter, and use conditionally-commutative updates to determine when to resize.

`vacation` is an online transaction processing application that emulates a travel

reservation system. During an execution, several client threads perform a number of sessions that interact with the travel system's database. To apply CommTM, we compile `vacation` to use resizable hash tables to manage the travel system's database. The resizable hash table is the same as that used in `genome`. Like `genome`, it has medium length transactions, medium read/write sets, and spends most if its time in transactions. Contention is low to moderate and depends on the number of client sessions that modify large portions of the database.

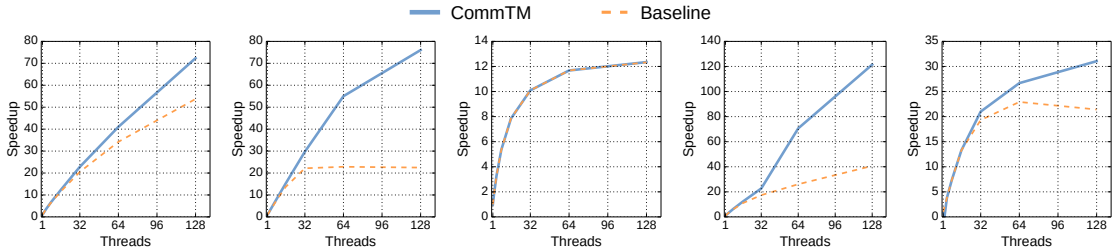
5.3.3 Results of Full Applications

We now present the results of the full application benchmarks. Figure 5-9 compares the performance and scalability of CommTM and the baseline HTM. We see that CommTM always outperforms 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 baseline HTM and CommTM often widens as the number of threads grows, demonstrating the better scalability of CommTM.

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. CommTM, however, makes these updates local and concurrent, achieving significant speedup. As the update contention decreases, the benefit of CommTM decreases. For applications such as `ssca2`, where there is little concurrent modification to shared data, CommTM yields a negligible improvement over the baseline HTM.

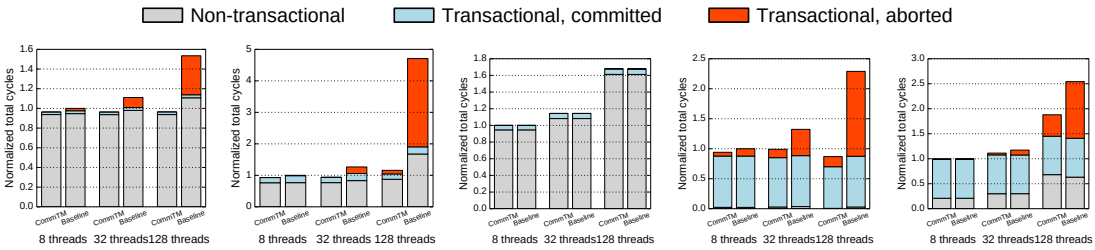
Figure 5-10 explains why CommTM has little impact on `ssca2`: contention is rare and thus only a small fraction of cycles are spent on aborted transactions.

Figure 5-11 shows that CommTM substantially reduces wasted transactional cycles. At 128 threads, CommTM's wasted cycles is lower than the baseline's by 25× on `kmeans`, 6.6% on `ssca2`. 8.3× on `genome`, and 2.6× on `vacation`. In `boruvka`, CommTM eliminates all aborts and hence eliminates all wasted transactional cycles.



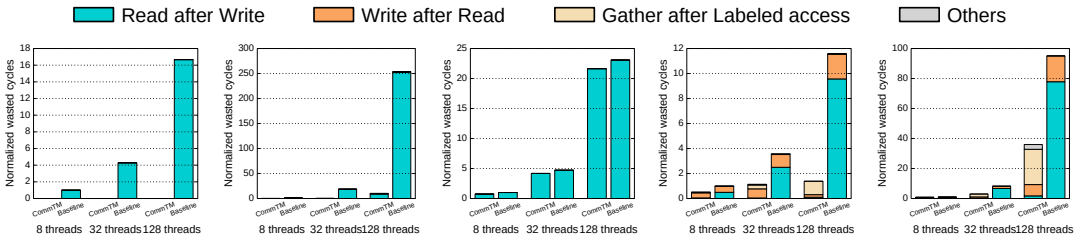
(a) boruvka (b) kmeans (c) ssca2 (d) genome (e) vacation

Figure 5-9: 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

Figure 5-10: Breakdown of total cycles for CommTM and baseline HTM for 8, 32, and 128 threads (lower is better).

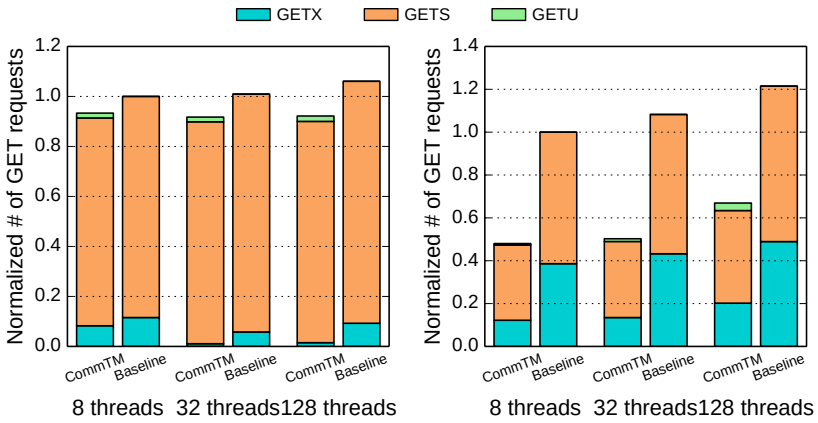


(a) boruvka (b) kmeans (c) ssca2 (d) genome (e) vacation

Figure 5-11: Breakdown of wasted cycles for CommTM and baseline HTM for 8, 32, and 128 threads (lower is better).

Figure 5-11 also 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. Figure 5-12 shows the breakdown of `GET` requests between 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 by 45%



(a) boruvka (b) kmeans

Figure 5-12: 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).

on `kmeans`. This also explains why non-transactional cycles are lower in Figure 5-10 (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 labeled loads, stores and gather requests, over all executed instructions are 0.13% on `boruvka`, 1.2% on `kmeans`, 0.000059% 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 (conventional) instructions, wasting a large amount of cycles.

Chapter 6

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 [38], SONTM [6], Wait-n-GoTM [23], and OmniOrder [37], 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 can substantially improve performance when dependences are acyclic, but semantically-commutative updates often consist of read-modify-write chains that cause cyclic dependencies.

SI-TM [26] relaxes serializability and implements snapshot isolation, which only flags write-write dependences as conflicts. SI-TM, like other schemes that weaken serializability [1, 47], 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 [26]).

Other techniques focus on reducing the cost of mispeculation. ReSlice [44] re-executes only the conflicting load and its dependent instructions, and RetCon [9] 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 operations than RetCon.

Finally, open-nested transactions [30, 31] 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. While open-nested transactions make their parents less vulnerable, the nested transactions still suffer from conflicts and serialization. By contrast, CommTM can support truly concurrent and communication-free updates to the same data. Moreover, open nesting is only practical when operations are easy to undo, which commutative operations may lack (e.g., top-K in Section 5.2).

Chapter 7

Conclusion

This thesis has presented CommTM, an HTM that exploits semantic commutativity to avoid conflicts that limit scalability in prior hardware speculation techniques. 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 or out-of-order updates. 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, achieving the high precision of software conflict detection 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.

Bibliography

[1] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[2] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L Johnson, David Kranz, John Kubiatowicz, B-H Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: architecture and performance. In *Proc. ISCA-22*, 1995.

[3] Alaa R Alameldeen and David A Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, (4), 2006.

[4] C Scott Ananian, Krste Asanović, Bradley C Kuszmaul, Charles E Leiserson, and Sean Lie. Unbounded transactional memory. In *Proc. HPCA-11*, 2005.

[5] Todd M Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. MICRO-32*, 1999.

[6] Utku Aydonat and Tarek S Abdelrahman. Hardware support for relaxed concurrency control in transactional memory. In *Proc. MICRO-43*, 2010.

[7] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3), 2014.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.

[9] Colin Blundell, Arun Raghavan, and Milo MK Martin. RETCON: transactional repair without replay. In *Proc. ISCA-37*, 2010.

[10] Jayaram Bobba, Kevin E Moore, Haris Volos, Luke Yen, Mark D Hill, Michael M Swift, and David A Wood. Performance pathologies in hardware transactional memory. In *Proc. ISCA-34*, 2007.

[11] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5), 2008.

[12] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. ISCA-33*, 2006.

[13] Hassan Chafi, Jared Casper, Brian D Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *Proc. HPCA-13*, 2007.

[14] Shimin Chen, Phillip B Gibbons, Michael Kozuch, and Todd C Mowry. Log-based architectures: using multicore to help software behave correctly. *ACM SIGOPS Operating Systems Review*, 45(1), 2011.

[15] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proc. SOSP-24*, 2013.

[16] Cliff Click. Azuls experiences with hardware transactional memory. In *Transactional Memory Workshop*, 2009.

[17] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[18] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1), 2011.

[19] Lance Hammond, Vicky Wong, Mike Chen, Brian D Carlstrom, John D Davis, Ben Hertzberg, Manohar K Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proc. ISCA-31*, 2004.

[20] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proc. PPoPP*, 2008.

[21] Mark D Hill. Is transactional memory an oxymoron? *Proceedings of the VLDB Endowment*, 1(1), 2008.

[22] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. The microarchitecture of the Pentium® 4 processor. In *Intel Technology Journal*, 2001.

[23] Syed Ali Raza Jafri, Gwendolyn Voskuilen, and TN Vijaykumar. Wait-n-GoTM: improving HTM performance by serializing cyclic dependencies. In *Proc. ASPLOS-XVIII*, 2013.

[24] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. Exploiting the commutativity lattice. In *Proc. PLDI*, 2011.

[25] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. Optimistic parallelism requires abstractions. In *Proc. PLDI*, 2007.

[26] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P Stevenson. SI-TM: reducing transactional memory abort rates through snapshot isolation. In *Proc. ASPLOS-XIX*, 2014.

[27] Kenneth Mackenzie, John Kubiatowicz, Matthew Frank, Wei-Jen Lee, Walter Lee, Anant Agarwal, and M Frans Kaashoek. Exploiting two-case delivery for fast protected messaging. In *Proc. HPCA-4*, 1998.

[28] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. IISWC*, 2008.

[29] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, David A Wood, et al. LogTM: log-based transactional memory. In *Proc. HPCA-12*, 2006.

[30] Michelle J Moravan, Jayaram Bobba, Kevin E Moore, Luke Yen, Mark D Hill, Ben Liblit, Michael M Swift, and David A Wood. Supporting nested transactional memory in LogTM. In *Proc. ASPLOS-XII*, 2006.

[31] J Eliot B Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, 2006.

[32] Shubhendu S Mukherjee, Peter Bannon, Steven Lang, Aaron Spink, and David Webb. The Alpha 21364 network architecture. In *Hot Interconnects 9, 2001.*, 2001.

[33] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *Proc. OSDI-11*, 2014.

[34] Li-Shiuan Peh and William J Dally. A delay model and speculative architecture for pipelined routers. In *Proc. HPCA-7*, 2001.

[35] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P Johnson, and David I August. Commutative set: A language extension for implicit parallel programming. In *Proc. PLDI*, 2011.

[36] Xuehai Qian, Wonsun Ahn, and Josep Torrellas. Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment. In *Proc. MICRO-43*, 2010.

[37] Xuehai Qian, Benjamin Sahelices, and Josep Torrellas. OmniOrder: Directory-based conflict serialization of transactions. In *Proc. ISCA-41*, 2014.

[38] Hany E Ramadan, Christopher J Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *Proc. MICRO-41*, 2008.

[39] Rodolfo F Resende, Divyakant Agrawal, and Amr El Abbadi. Semantic locking in object-oriented database systems. In *Proc. OOPSLA*, 1994.

[40] Martin C Rinard and Pedro C Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proc. PLDI*, 1996.

[41] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using GCC and memcached. In *Proc. ASPLOS-XIX*, 2014.

[42] Daniel Sanchez and Christos Kozyrakis. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proc. ISCA-40*, 2013.

[43] Daniel Sanchez, Richard M Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proc. ASPLOS-XV*, 2010.

[44] Smruti R Sarangi, Wei Liu Torrellas, Yuanyuan Zhou, et al. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proc. MICRO-38*, 2005.

[45] Keun Sup Shim, Marcin Lis, Myong Hyon Cho, Ilya Lebedev, and Srinivas Devadas. Design tradeoffs for simplicity and efficient verification in the Execution Migration Machine. In *Proc. ICCD*, 2013.

[46] Julian Shun, Guy E Blelloch, Jeremy T Fineman, and Phillip B Gibbons. Reducing contention through priority updates. In *Proc. SPAA*, 2013.

[47] Travis Skare and Christos Kozyrakis. Early release: Friend or foe. In *Workshop on Transactional Memory Workloads*, 2006.

[48] Thorsten Von Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. ISCA-19*, 1992.

[49] William E Weihl. Commutativity-based concurrency control for abstract data types. *Computers, IEEE Transactions on*, 37(12), 1988.

[50] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M Aamodt, Jamison D Collins, Perry H Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, and Hong Wang. Pangaea: a tightly-coupled IA32 heterogeneous chip multiprocessor. In *Proc. PACT-17*, 2008.

[51] Richard M Yoo, Christopher J Hughes, Koonchun Lai, and Ravi Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *Proc. SC13*, 2013.

[52] Guowei Zhang, Webb Horn, and Daniel Sanchez. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proc. MICRO-48*, 2015.