Architectural Support to Exploit Commutativity in Shared-Memory Systems

by

Guowei Zhang

B.S., Microelectronics B.S., Economics

Tsinghua University, China (2014)

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 2016

Author..... Signature redacted .....
Department of Electrical Engineering and Computer Science
May 19, 2016

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

Accepted by..... Signature redacted .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Architectural Support to Exploit Commutativity in Shared-Memory Systems

by

Guowei Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2016, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Parallel systems are limited by the high costs of communication and synchronization. Exploiting *commutativity* has historically been a fruitful avenue to reduce traffic and serialization. This is because commutative operations produce the same final result regardless of the order they are performed in, and therefore can be processed concurrently and without communication.

Unfortunately, software techniques that exploit commutativity, such as privatization and semantic locking, incur high runtime overheads. These overheads offset the benefit and thereby limit the applicability of software techniques. To avoid high overheads, it would be ideal to exploit commutativity in hardware. In fact, hardware already provides much of the functionality that is required to support commutativity. For instance, private caches can buffer and coalesce multiple updates. However, current memory hierarchies can understand only reads and writes, which prevents hardware from recognizing and accelerating commutative operations.

The key insight this thesis develops is that, with minor hardware modifications and minimal extra complexity, cache coherence protocols, the key component of communication and synchronization in shared-memory systems, can be extended to *allow local and concurrent commutative* operations. This thesis presents two techniques that leverage this insight to exploit commutativity in hardware.

First, COUP provides architectural support for a limited number of single-instruction commutative updates, such as addition and bitwise logical operations. COUP allows multiple private caches to simultaneously hold update-only permission to the same cache line. Caches with update-only permission can locally buffer and coalesce updates to the line, but cannot satisfy read requests. Upon a read request, COUP reduces the partial updates buffered in private caches to produce the final value.

Second, COMMTM is a commutativity-aware hardware transactional memory (HTM) that supports an even broader range of multi-instruction, semantically commutative operations, such as set insertions and ordered puts. COMMTM extends the coherence protocol with a reducible state 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 arbitrary user-defined commutative operations through labeled loads and stores. 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.

COUP and COMMTM reduce communication and synchronization in many challenging parallel workloads. At 128 cores, COUP accelerates state-of-the-art implementations of update-heavy algorithms by up to 2.4×, and COMMTM outperforms a conventional eager-lazy HTM by up to 3.4× and reduces or eliminates wasted work due to transactional aborts.

Thesis Supervisor: Daniel Sanchez
Title: Assistant Professor

Contents

Acknowledgments

First and foremost, I would like to thank my research advisor, Professor Daniel Sanchez. Two years ago, I came to MIT and started working in computer architecture. Since then, Daniel has been a constant source of excellent advice and patient encouragement, guiding me to overcome stress before conference presentations or fight against crashed simulations. He helped me get started in this area and directed me towards important and challenging problems. His rich knowledge and positive attitude helped me so much that I cannot imagine how this thesis would have happened without his guidance.

I am thankful for the support of our research group. I am especially grateful to my great co-authors: Webb Horn and Virginia Chiu. Working with them was such an enjoyable experience. I also learned a lot from the rest of our group: Professor Joel Emer, Nathan Beckmann, Harshad Kasture, Mark Jeffrey, Suvinay Subramanian, Po-An Tsai, Anurag Mukkara, and Maleen Abeydeera. It is my great pleasure to receive their valuable feedback on my work and discuss all kinds of fun topics.

I would also like to thank Professor David Perreault, my academic advisor, who helped me select courses and pursue the S.M. degree. Additionally, I thank EECS and Math Professors Michael Sipser, Daniel Sanchez, Joel Emer, Gregory Wornell, Stefanie Jegelka, and Samuel Madden. I was fortunate to learn from their excellent lectures.

Last but not least, I thank my friends, both in the U.S. and in China, and my family for their support and encouragement. Particularly, I owe thanks to my mom Yanxia Liu, who always succeeds in alleviating my pressure and making me confident. Finally, I thank my girlfriend Huan Shao for her constant love across the twelve-hour time difference.

Introduction

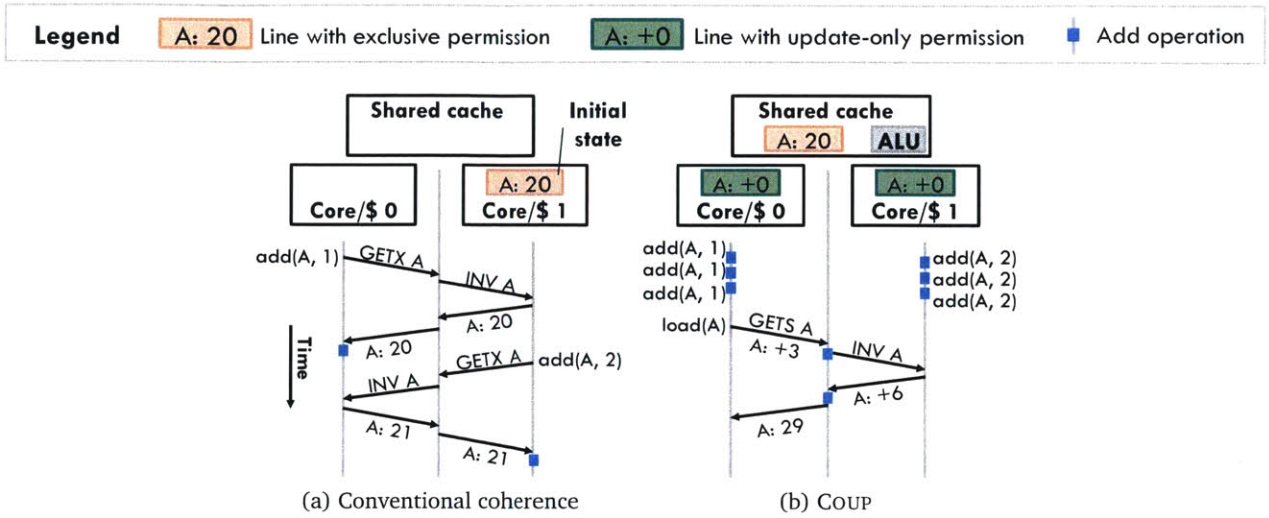
1

This thesis focuses on exploiting commutativity in hardware to minimize communication and synchronization in shared-memory systems. Parallel systems are limited by the high costs of communication and synchronization. For instance, at 28nm, a 64-bit floating-point multiply-add operation requires 20pJ, while shipping its operands across the chip consumes 300pJ, 15× higher [33, 53, 101]. Moreover, these overheads are increasing with system size, and conventional systems will not scale to handle the growing communication and synchronization. Much recent work has focused on reducing the cost of synchronization, but a more promising approach is to eliminate synchronization altogether.

Historically, exploiting *commutativity* has been a fruitful approach to do so. Commutative operations produce the same final result regardless of the order they are performed in. These operations need not read the data they update, so they do not introduce true data dependencies and therefore can be processed concurrently and coalesced locally before the data is read. Hence, updates can proceed concurrently, *without communication or synchronization*.

Many software techniques have exploited commutativity to improve parallelism, but unfortunately they incur high runtime overheads, which significantly offset their benefit and limit their applicability. For example, *privatization* is a software approach that lowers the cost of commutative updates by using thread-local variables [12, 29, 77]: each thread updates its local variable, and reads require reducing the per-thread variables. Privatization suffers from both slow transition between read-only and update-only phases and large memory footprint [52], and therefore can only be applied selectively. Moreover, in speculative parallelization techniques, *semantic locking* [91, 111] allows multiple *semantically commutative* operations on the same object to acquire its corresponding lock in a compatible mode, so they execute concurrently. But such software techniques incur high runtime overheads, typically 2-6× in software transactional memory [20].

Ideally, it would be desirable to implement commutativity support in hardware to avoid high overheads. In fact, hardware already provides much of the functionality that is required to exploit commutativity. For example, private caches can be used to buffer and coalesce multiple updates. However, exploiting commutativity in hardware is still challenging. The key reason is that current memory hierarchies support only two primitive operations: reads and writes. Therefore, updates must be expressed as read-modify-write sequences, even when they are commutative. As a result, hardware is unable to recognize and accelerate commutative operations, and thus frequent updates to shared data incur significant communication and synchronization. For example, consider a shared counter that is updated by multiple cores. On each update, the updating core first fetches an exclusive copy of the counter's cache line into its private cache, invalidating all other copies, and modifies it locally using an atomic operation such as fetch-and-add, as shown in Figure 1.1a. Each update incurs significant *traffic* and *serialization*: traffic to fetch the line and invalidate other copies, causing the line to ping-pong among updating cores; and serialization because only one core can perform an update at a time.



(a) Conventional coherence (b) COUP

Figure 1.1: Example comparing the cost of commutative updates under two schemes. Two cores add values to a single memory location, A. (a) With conventional coherence protocols, A's fetches and invalidations dominate the cost of updates. (b) With COUP, caches buffer and coalesce updates locally, and reads trigger a reduction of all local updates to produce the actual value.

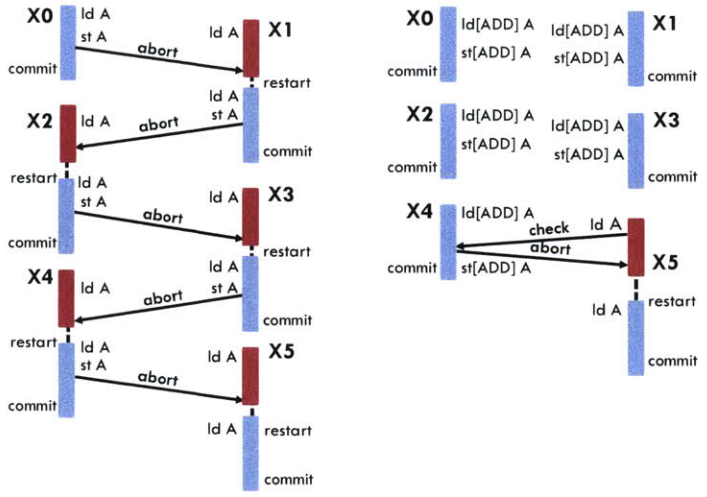
1.1 Contributions

The key insight of this thesis is that with minor modifications and minimal additional complexity, cache coherence protocols, the key component of communication and synchronization in shared-memory systems, can be extended to process commutative updates locally and concurrently. We present two hardware techniques that leverage this insight to reduce communication and improve concurrency. Specifically, we make the following contributions:

COUP (Chapter 3) is a general technique that extends coherence protocols to allow local and concurrent commutative updates. Specifically, COUP decouples read and write permissions, and introduces a limited number of commutative-update primitive operations, in addition to reads and writes. With COUP, multiple caches can acquire a line with *update-only* permission, and satisfy commutative-update requests locally, buffering and coalescing updates. On a read request, the coherence protocol gathers all the local updates and *reduces* them to produce the correct value before granting read permission. For example, multiple cores can concurrently add values to the same counter. Updates are held in their private caches as long as no core reads the current value of the counter. When a core reads the counter, all updates are added to produce the final value, as shown in Figure 1.1b.

COUP provides significant benefits at minimal cost. COUP introduces minor hardware overheads, preserves coherence and consistency, and imposes small verification costs. We identify several update-heavy parallel applications where current techniques have clear shortcomings, and discuss how COUP addresses them. Specifically, COUP supports a limited number of *single-instruction* commutative operations, such as addition and bitwise logical operations. We evaluate COUP under simulation, using single- and multi-socket systems. At 128 cores, COUP improves the performance of update-heavy benchmarks by 4%–2.4× and reduces traffic by up to 20×.

COMMTM (Chapter 4) is a commutativity-aware HTM that extends COUP to support an unlimited number of user-defined commutative operations that cannot be expressed as single instructions. Specifically,



(a) Conventional HTM (b) COMMTM

Figure 1.2: 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.

COMMTM supports a much broader range of *multi-instruction, semantically commutative* operations, such as set insertions and ordered puts. We find that commutativity and transactional memory are complementary: transactions benefit commutativity by guaranteeing the atomicity of multi-instruction operations, and commutativity benefits transactions by avoiding unnecessary conflicts and wasted work, as shown in Figure 1.2.

COMMTM extends the coherence protocol with a *reducible* state that generalizes COUP's update-only 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.

We explore several variants of COMMTM that trade precision for hardware complexity. We first present a basic version of COMMTM that achieves the same precision as software *semantic locking* [59, 111]. We then extend COMMTM with *gather requests*, which allow software to redistribute reducible data among caches, achieving much higher concurrency in important use cases.

We evaluate COMMTM with microbenchmarks and full TM applications. Microbenchmarks show that COMMTM scales on a variety of commutative operations, such as set insertions, reference counting, ordered puts, and top-K 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.

In summary, COUP and COMMTM enable parallel systems to exploit commutativity without the overheads of software techniques. This allows parallel systems to approach the minimal communication and synchronization truly required by the algorithms they run.

Background 2

Commutativity in Parallel Systems 2.1

Commutativity [111] has been widely exploited in parallel systems, such as databases [12, 77], parallelizing compilers [81, 92], and runtimes [59, 81]. Operations are commutative when they produce the same final result regardless of the order they are performed in. Thus, commutative operations introduce neither fundamental data dependencies nor necessary communication. As a result, they can be processed concurrently and coalesced locally to reduce traffic and serialization.

Commutative updates are common in many cases. *Strictly commutative* operations produce exactly the same final state when reordered. For instance, consider two additions to a counter. Reordering them does not affect the final result at all, and therefore integer addition is strictly commutative. Besides addition, multiplication, minimization, maximization and bitwise logical operations are all strictly commutative.

An even broader concept is *semantic commutativity* [111]. Semantically commutative operations produce results that are semantically equivalent when reordered, even if the concrete resulting states are different. 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). Thus, these set insertions are not strictly commutative. 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 semantically commute. Other examples include ordered puts and top-K insertions.

Reducing Communication and Synchronization in Non-speculative Parallelism 2.2

Conventional shared-memory programs update shared data using atomic operations for single-word updates, or normal reads and writes with synchronization (e.g., locks or transactions) for multi-word updates. Many software and hardware optimizations seek to reduce the cost of updates. Though often presented in the context of specific algorithms or implementations, we observe these techniques can be classified into two categories: they are either *privatization*-based or *delegation*-based.

Privatization-based Techniques 2.2.1

Privatization is a software technique that exploits commutativity to reduce the costs of updates. Privatization schemes buffer updates in thread-private storage, and require reads to reduce these thread-private updates to produce the correct value. Privatization is most commonly used to implement reduction variables efficiently, often with language support (e.g., reducers in MapReduce [36], OpenMP pragmas, and Cilk Plus hyperobjects [42]). Privatization is generally used when updates are frequent and reads are rare.

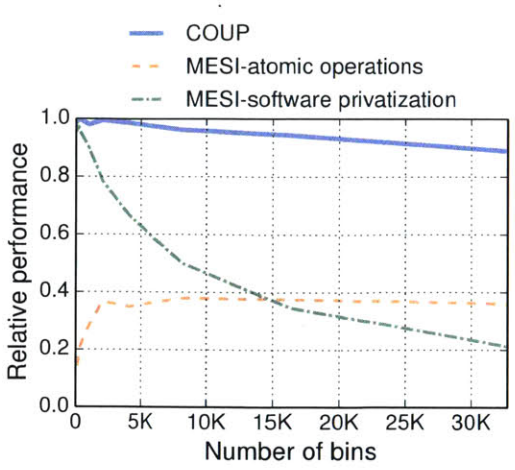


Figure 2.1: Performance of parallel histogram implementations using atomics, software privatization, and COUP. More bins reduce contention and increase privatization overheads, favoring atomics. COUP does not suffer these overheads, so it outperforms both software implementations.

Privatization is limited to commutative updates, and works best when data goes through long update-only phases without intervening reads. However, privatization has two major sources of overhead. First, software reductions are much slow, making finely-interleaved reads and updates inefficient. Second, with N threads, privatized variables increase the memory footprint by a factor of N. This makes naïve privatization impractical in many contexts (e.g., reference counting). Dynamic privatization schemes [29, 77, 115] can lessen space overheads, but add time overheads and complexity.

These overheads often make privatization underperform conventional updates. For instance, Jung et al. [52] propose parallel histogram implementations using both atomic operations and privatization. These codes process a set of input values, and produce a histogram with a given number of bins. Jung et al. note that privatization is desirable with a few output bins, but works poorly with many bins, as the reduction phase dominates execution time and hurts locality. Figure 2.1 shows this tradeoff. It compares the performance of histogram implementations using atomic fetch-and-add, privatization, and COUP (Chapter 3), when running on 64 cores (see Section 3.6 for methodology details). In this experiment, all schemes process a large, fixed number of input elements. Each line shows the performance of a given implementation as the number of output bins (x-axis) changes from 32 to 32 K. Performance is reported relative to COUP's at 32 bins (higher numbers are better). While the costs of privatization impose a delicate tradeoff between both implementations in software, COUP robustly outperforms both. In fact, COUP is the *hardware counterpart of privatization*.

2.2.2 Delegation-based Techniques

Delegation-based techniques send updates to a single location to reduce data movement and hence the cost of updates. Unlike privatization-based techniques, delegation-based techniques do not leverage commutativity. This limits their performance benefits, but makes them applicable for non-commutative operations.

Specifically, in software, delegation schemes send updates to a single thread [18, 19]. They divide shared data among threads and send updates to the corresponding thread, using shared-memory queues [18] or active messages [96, 107]. Delegation is common in architectures that combine shared memory and message passing [96, 113] and in NUMA-aware data structures [18, 19]. Although delegation reduces data movement and synchronization, it still incurs global traffic and serialization.

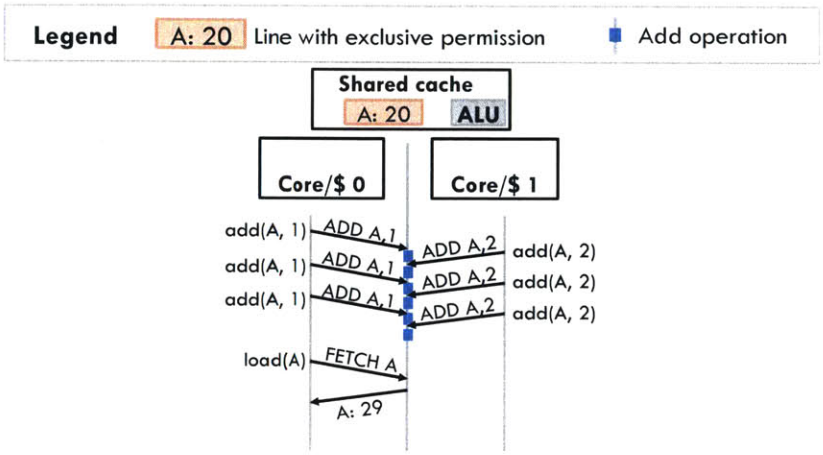


Figure 2.2: With remote memory operations, cores send updates to a fixed location, the shared cache in this case.

Remote memory operations (RMOs) [43, 49, 99, 118] are the hardware counterpart of delegation. Rather than caching lines to be updated, update operations are sent to a fixed location, as shown in Figure 2.2. The NYU Ultracomputer [43] proposed implementing atomic fetch-and-add using adders in network switches, which could coalesce multiple requests on their way to memory. The Cray T3D [55], T3E [99], and SGI Origin [65] implemented RMOs at the memory controllers, while TilePro64 [49] and recent GPUs [112] implement RMOs in shared caches. Prior work has also proposed adding caches to memory controllers to accelerate RMOs [118] and data-parallel RMOs [6].

Although RMOs reduce update costs for both commutative and non-commutative operations, they suffer from two issues. First, while RMOs avoid ping-ponging cache lines, they still require sending every update to a shared, fixed location, causing global traffic. RMOs are also limited by the throughput of the single updater. For example, in Figure 2.2, frequent remote-add requests drive the shared cache's ALU near saturation. Second, strong consistency models are challenging to implement with RMOs, as it is harder to constrain memory operation order. For example, TSO requires making stores globally visible in program order, which is feasible with local store buffers, but much more complicated when stores are also performed by remote updaters. As a result, most implementations provide weakly-consistent RMOs. Timestamp-based order validation [62, §5] allows strong consistency with RMOs, but it is complicated.

Reducing Communication and Synchronization in Speculative Parallelism 2.3

Many software and hardware techniques, such as transactional memory (TM) or speculative multi-threading, rely on speculative execution to parallelize programs with atomic regions. For example, transactional memory lets programmers define transactions, regions of code that are executed atomically. For instance, in the following function, the read-modify-write sequences of account balances are placed in a transaction, which must appear atomic to ensure correctness.

```
void transfer(Account& from, Account& to, int amount) {
  atomic {
    to.balance += amount;
    from.balance -= amount;
  }
}
```

Speculative execution techniques run multiple atomic regions 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 runtime costs. Software and hardware conflict detection techniques satisfy either of these properties but sacrifice the other: On the one hand, software techniques can leverage program semantics to be highly precise, but they incur high runtime overheads. On the other hand, hardware techniques incur small overheads, but leave a great amount of concurrency unexploited.

2.3.1 Software Techniques

Software conflict detection schemes often exploit semantic commutativity [59, 60, 77, 81, 91, 111]. 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 [45, 59, 81, 91, 111]. Not all commutativity-aware conflict detection schemes are equally precise: simple and general-purpose techniques, such as *semantic locking* [59, 91, 111], flag some semantically-commutative operations as conflicts, while more sophisticated schemes, like gatekeepers [59], incur fewer conflicts but have higher overheads and are often specific to particular patterns.

Specifically, semantic locking [91, 111], also known as abstract 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. For instance, the deposit operations to the same account are commutative. Therefore, multiple deposit operations can hold the lock of the account in *addition* mode at the same time, and can proceed concurrently without blocking one another. Semantic locking requires additional synchronization on the actual accesses to shared data, e.g., logging or reductions. However, such software techniques incur high runtime overheads (e.g. 2-6× in software TM [20]).

2.3.2 Hardware Techniques

Hardware can implement speculative execution at minimal costs by reusing many existing components: private caches to buffer speculative data, and the coherence protocol to detect conflicting speculative accesses. In fact, after an intensive period of research [21, 44, 46, 72, 84, 85], hardware transactional memory has been quickly adopted in commercial processors [76, 109]. Likewise, hardware already has much of the functionality that is necessary to support commutativity. However, exploiting commutativity in hardware conflict detection is still challenging because conventional coherence protocols can reason only reads and writes. Therefore, commutative updates to the same data, e.g. deposits to the same account, trigger unnecessary conflicts, as shown in Figure 1.2a. 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 [59, 60].

COMMTM bridges the gap between software and hardware and solves this precision-overhead dichotomy. By extending the coherence protocol and the conflict detection scheme, COMMTM provides commutativity support for hardware speculation, and avoids the overheads of software techniques.

2.4 Summary

Table 2.1 summarizes the characteristics of COUP, COMMTM and other related software and hardware techniques. In general, COUP makes single-instruction commutative updates as inexpensive as normal

Techniques	Hardware or software?	Reduces speculative conflicts?	Exploits strict commutativity?	Exploits semantic commutativity?	Reduces communication?
Delegation	SW	✗	✗	✗	✗
RMO	HW	✗	✗	✗	✗
Privatization	SW	✗	✓	✓	✓
COUP	HW	✗	✓	✗	✓
Semantic locking	SW	✓	✓	✓	✗
COMMTM	HW	✓	✓	✓	✓

Table 2.1: Summary of COUP, COMMTM and related work.

reads, and COMMTM allows multi-instruction commutative operations to be processed concurrently and without conflicts.

COUP 3

In this chapter, we present COUP, a general technique that extends coherence protocols to allow *local and concurrent commutative updates*. COUP allows multiple caches to acquire a line with *update-only* permission, and satisfy commutative-update requests locally, buffering and coalescing updates. Upon a read, the coherence protocol collects all the local updates and *reduces* them to produce the correct value before granting read permission.

COUP confers significant benefits over RMOs, especially when data receives several consecutive updates before being read. Moreover, COUP maintains full cache coherence and does not affect the memory consistency model. This makes COUP easy to apply to current systems and applications. Note that COUP's advantages come at the cost of a more restricted set of operations: COUP is limited to commutative updates, while RMOs support non-commutative operations such as fetch-and-add and compare-and-swap.

COUP also completes a symmetry between hardware and software schemes to reduce the cost of updates. Just as remote memory operations are the hardware counterpart to delegation, COUP *is the hardware counterpart to privatization*. COUP has two benefits over software privatization. First, transitions between read-only and update-only modes are much faster, so COUP remains practical in many scenarios where software privatization requires excessive synchronization. Second, privatization's thread-local copies increase memory footprint and add pressure to shared caches, while COUP does not.

We demonstrate COUP's utility by applying it to improve the performance of *single-word update operations*, which are currently performed with expensive atomic read-modify-write instructions.

Overall, we make the following contributions:

- We present COUP, a technique that extends coherence protocols to support concurrent commutative updates (Section 3.1 and Section 3.2). We show that COUP preserves coherence and consistency (Section 3.3), and imposes small verification costs (Section 3.4).
- We identify several update-heavy parallel applications where current techniques have clear shortcomings (Section 3.5), and discuss how COUP addresses them.
- We evaluate COUP under simulation, using single- and multi-socket systems (Section 3.6). At 128 cores, COUP improves the performance of update-heavy benchmarks by 4%–2.4×, and reduces traffic by up to 20×.

In summary, COUP shows that extending coherence protocols to leverage the semantics of commutative updates can substantially improve performance without sacrificing the simplicity of cache coherence.

COUP Example: Extending MSI 3.1

We first present the main concepts and operation of COUP through a concrete, simplified example. Consider a system with a single level of private caches, kept coherent with the MSI protocol. This system has a single shared last-level cache with an in-cache directory. It implements a single commutative-update

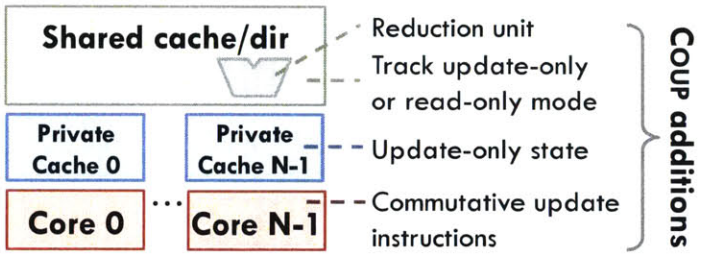


Figure 3.1: Summary of additions and modifications needed to support COUP.

operation, addition. Finally, we restrict this system to use single-word cache blocks. We will generalize COUP to other protocols, operations, and cache hierarchies in Section 3.2.

3.1.1 Structural changes

COUP requires modest changes to hardware structures, summarized in Figure 3.1 and described below.

Commutative-update instructions: In most ISAs, COUP needs additional instructions that let programs convey commutative updates, as conventional atomic instructions (e.g., fetch-and-add) return the latest value of the data they update. In this case, we add a *commutative-addition* instruction, which takes an address and a single input value, and does not write to any register.

Some ISAs may not need additional instructions. For instance, the recent Heterogeneous System Architecture (HSA) includes atomic-no-return instructions that do not return the updated value [2]. While these instructions were likely introduced to reduce the cost of RMOs, COUP could use them directly.

Update-only permission: COUP extends MSI with an additional state, *update-only (U)*, and a third type of request, *commutative update (C)*, in addition to conventional reads (R) and writes (W). We call the resulting protocol MUSI. Figure 3.2 shows MUSI's state-transition diagram for private caches. MUSI allows multiple private caches to hold read-only permission to a line and satisfy read requests locally (S state); multiple private caches to hold update-only permission to a line and satisfy commutative-update requests locally (U state); or at most a single private cache to hold exclusive permission to a line and satisfy all types of requests locally (M state). By allowing M to satisfy commutative-update requests, interleaved updates and reads to *private* data are as cheap as in MSI.

MUSI's state-transition diagram shows a clear symmetry between S and U: all transitions caused by R/C requests in and out of S match those caused by C/R requests in and out of U. We will exploit this symmetry in Section 3.4 to simplify our implementation.

Directory state: Conventional directories must track both the sharers of each line (using a bit-vector or other techniques [23, 94, 116]), and, if there is a single sharer, whether it has exclusive or read-only permission. In COUP, the directory must track whether sharers have exclusive, read-only, or update-only permission. The sharers bit-vector can be used to track both multiple readers or multiple updaters, so MUSI requires only one extra bit per directory tag.

Reduction unit: Though cores can perform local updates, the memory system must be able to perform reductions. Thus, COUP adds a reduction unit to the shared cache, consisting of an adder in this case.

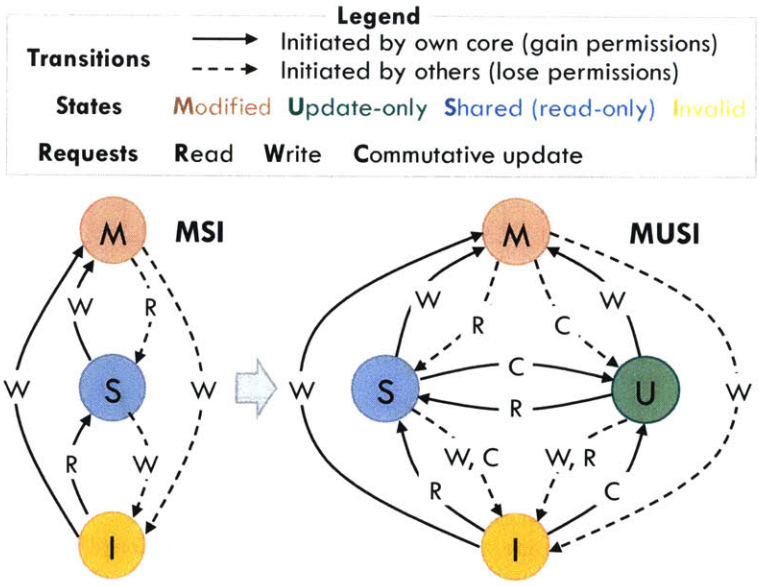


Figure 3.2: State-transition diagrams of MSI and MUSI protocols. For clarity, diagrams omit actions that do not cause a state transition (e.g., R requests in S).

Protocol operation

3.1.2

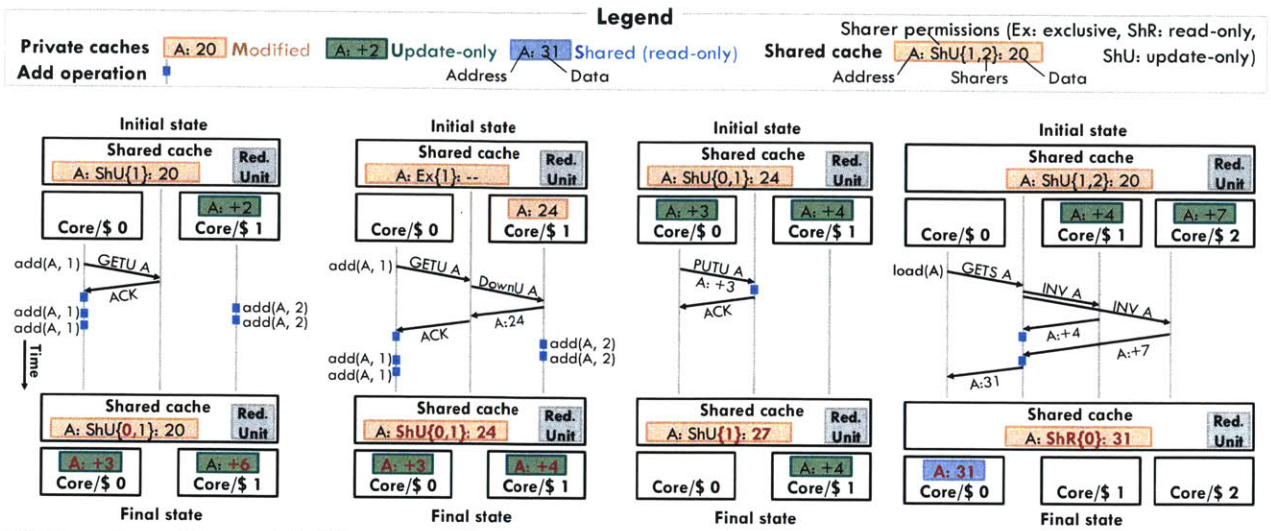
Performing commutative updates: Both the M and U states provide enough permissions for private caches to satisfy update-only requests. In M, the private cache has the actual data value; in U, the cache has a partial update. In either case, the core can perform the update by atomically reading the data from the cache, modifying it (by adding the value specified by the commutative-add instruction) and storing the result in the cache. The cache cannot allow any intervening operations to the same address between the read and the write. This scheme can reuse the existing core logic for atomic operations. We assume this scheme in our implementation, but note that alternative implementations could treat commutative updates like stores to improve performance (e.g., using update buffers similar to store buffers and performing updates with an ALU at the L1).

Entering the U state: When a cache has insufficient permissions to satisfy an update request (I or S states), it requests update-only permission from the directory. The directory invalidates any copies in S, or downgrades the single copy in M to U, and grants update-only permission to the requesting cache, which transitions to U. Thus, there are two ways a line can transition into the U state: by requesting update-only permission to satisfy a request from its own core, as shown in Figure 3.3a; or by being downgraded from M, as shown in Figure 3.3b.

When a line transitions into U, its contents are always initialized to the *identity element*, 0 for commutative addition. This is done even if the line had valid data. This avoids having to track which cache holds the original data when doing reductions. However, reductions require reading the original data from the shared cache.

Leaving the U state: Lines can transition out of U due to either evictions or read requests.

Evictions initiated by a private cache (to make space for a different line) trigger a *partial reduction*, shown in Figure 3.3c: the evicting cache sends its partial update to the shared cache, which uses its reduction unit to aggregate it with its local copy.



(a) Upgrade to U caused by commutative-update request

(b) Downgrade caused by commutative-update request

(c) Partial reduction caused by eviction

(d) Full reduction caused by read request

Figure 3.3: MUSI protocol operation: (a) granting update-only (U) state; (b) downgrade from M to U due to an update request from another core; (c) partial reduction caused by an eviction from a private cache; and (d) full reduction caused by a read request. Each diagram shows the initial and final states in the shared and private caches.

The shared cache may also need to evict a line that private caches hold in U. This triggers a *full reduction*: all caches with update-only permission are sent invalidations, reply with their partial updates, and the shared cache uses its reduction unit to aggregate all partial updates and its local copy, producing the final value.

Finally, read requests from any core also trigger a full reduction, as shown in Figure 3.3d. Depending on the latency and throughput of the reduction unit, satisfying a read request can take somewhat longer than in conventional protocols. Hierarchical reductions can rein in reduction overheads with large core counts (Section 3.2). In our evaluation, we observe that reduction overheads are small compared to communication latencies.

3.2 Generalizing COUP

We now show how to generalize COUP to support multiple operations, larger cache blocks, other protocols, and deeper cache hierarchies.

Multiple operations: Formally, COUP can be applied to any *commutative semigroup* $(G, \circ)$.[1] For example, G can be the set of 32-bit integers, and $\circ$ can be addition, multiplication, and, or, xor, min, or max.

Supporting multiple operations in the system requires minor changes. First, additional instructions are needed to convey each type of update. Second, reduction units must implement all supported operations. Third, the directory and private caches must track, for each line in U state, what type of operation is being performed. Fourth, COUP must serialize commutative updates of different types, because they do not commute in general (e.g., + and $*$ do not commute with each other). This can be

[1] $(G, \circ)$ is a commutative semigroup iff $\circ : G \times G \rightarrow G$ is a binary, associative, commutative operation over elements of set G, and G is closed under $\circ$.

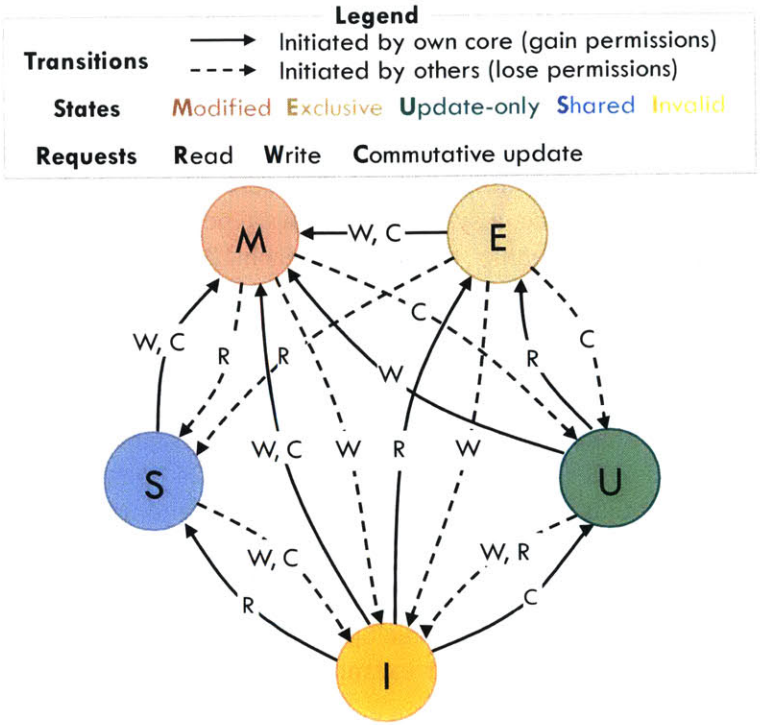


Figure 3.4: State-transition diagram of MEUSI. Just as MESI grants E to a read request if a line is unshared, MEUSI grants M to an update request if a line is unshared. For clarity, the diagram omits actions that do not cause a state transition (e.g., C requests in U).

accomplished by performing a full reduction every time the private cache or directory receives an update request of a type different from the current one.

Larger cache blocks: Supporting multi-word blocks is trivial if $(G, \circ)$ has an *identity element* (formally, this means $(G, \circ)$ is a commutative monoid). The identity element produces the same value when applied to any element in G. For example, the identity elements for addition, multiplication, *and*, and *min* are 0, 1, all-ones, and the maximum representable integer, respectively.

All the operations we implement in this work have an identity element. In this case, it is sufficient to initialize every word of the cache block to the identity element when transitioning to U. Reductions perform element-wise operations even on words that have received no updates. Note this holds *even if those words do not hold data of the same type*, because applying $\circ$ on the identity element produces the same output, so it does not change the word's bit pattern. Alternatively, reduction units could skip operating on words with the identity element.

In general, not all operations may have an identity element. In such cases, the protocol would require an extra bit per word to track uninitialized elements.

Finally, note we assume that data is properly aligned. Supporting commutative updates to unaligned data would require more involved mechanisms to buffer partial updates. If the ISA allows unaligned accesses, they can be performed as normal read-modify-writes.

Other protocols: COUP can extend protocols beyond MSI. Figure 3.4 shows how MESI [79] is extended to MEUSI, which we use in our evaluation. Note that update requests enjoy the same optimization that

E introduces for read-only requests: if a cache requests update-only permission for a line and no other cache has a valid copy, the directory grants the line directly in M.

Deeper cache hierarchies: COUP can operate with multiple intermediate levels of caches and directories. COUP simply requires a reduction unit at each intermediate level that has multiple children that can issue update requests. For instance, a system with private per-core L1s and L2s and a fully shared L3 needs reduction units only at L3 banks. However, if each L2 was shared by two or more L1Ds, a reduction unit would be required in the L2s as well.

Hierarchical organizations lower the latency of reductions in COUP, just as they lower the latency of sending and processing invalidations in conventional protocols: on a full reduction, each intermediate level aggregates all partial updates from its children before replying to its parent. For example, consider a 128-core system with a fully-shared L4 and 8 per-socket L3s, each shared by 16 cores. In this system, a full reduction of a line shared in U state by all cores has $8 + 16 = 24$ operations in the critical path—far fewer than the 128 operations that a flat organization would have, and not enough to dominate the cost of invalidations.

Other contexts: We focus on single-word atomic operations and hardware cache coherence, but note that COUP could apply to other contexts. For example, COUP could be used in software coherence protocols (e.g., in distributed shared memory).

3.3 Coherence and Consistency

COUP maintains cache coherence and does not change the consistency model.

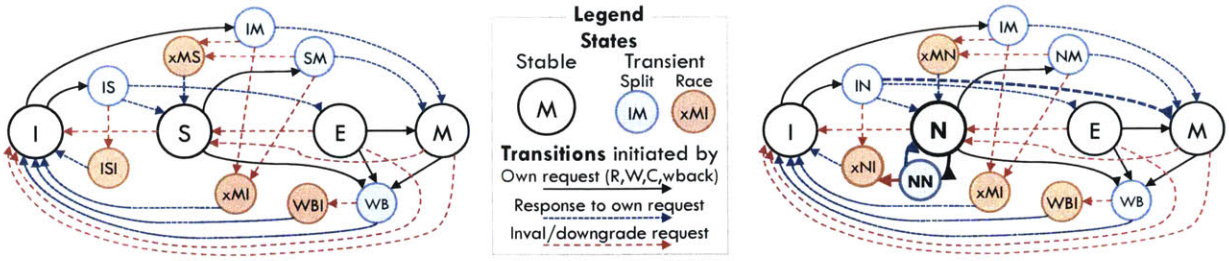
Coherence: A memory system is coherent if, for each memory location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and that obeys two invariants [32, §5.1.1]:[2]

1. Operations issued by each core occur in the order in which they were issued to the memory system by that core.
2. The value returned by each read operation is the value written to that location in the serial order.

In COUP, a location can be in exclusive, read-only, or update-only modes. The baseline protocol that COUP extends already enforces coherence in and between exclusive and read-only modes. In update-only mode, multiple cores can concurrently update the location, but because updates are commutative, *any serial order* we choose produces the same execution result. Thus, the first invariant is trivially satisfied. Moreover, transitions from update-only to read-only or exclusive modes propagate all partial updates and make them visible to the next reader. Thus, the next reader always observes the last value written to that location, satisfying the second property. Therefore, COUP maintains coherence.

Consistency: As long as the system restricts the order of memory operations as strictly for commutative updates as it does for stores, COUP does not affect the consistency model. In other words, it is sufficient for the memory system to consider commutative updates as being equivalent to stores. For instance, by having store-load, load-store, and store-store fences apply to commutative updates as well, systems with relaxed memory models need not introduce new fence instructions.

[2]Others reason about coherence using the *single-writer, multiple-reader* and the *data-value* invariants [105], which are sufficient but not necessary. COUP does not maintain the single-writer, multiple-reader invariant.



(a) MESI

(b) MEUSI with generalized N state

Figure 3.5: COUP implementation: (a) full state-transition diagram for the L1 cache on the baseline two-level MESI protocol; (b) corresponding MEUSI state-transition diagram. The non-exclusive state, N, generalizes S and U, and requires only an extra transient state and four transitions over MESI.

Implementation and Verification Costs 3.4

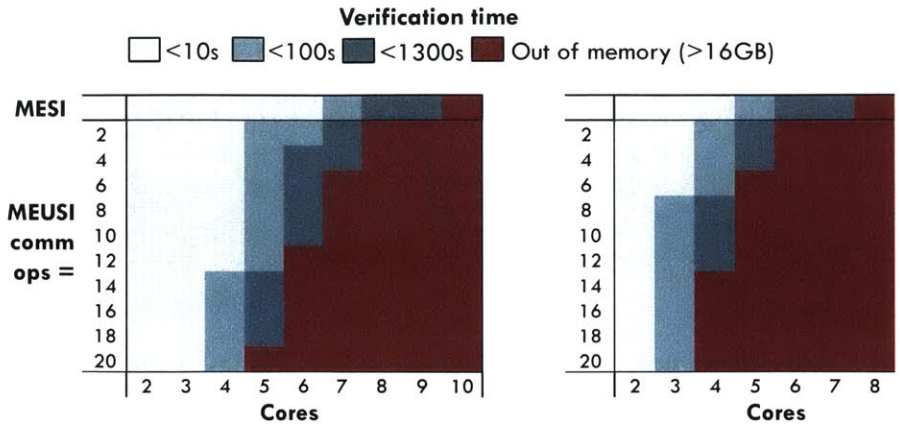
While we have presented COUP in terms of stable states, realistic protocols implement coherence transactions with additional transient states and are subject to races, which add complexity and hinder verification. By studying full implementations of MESI and MEUSI, we show that COUP requires a minimal number of transient states and adds modest verification costs.

We first implement MESI protocols for two- and three-level cache hierarchies. Our implementations work on networks with unordered point-to-point communication, and use two virtual networks without any message buffering at the endpoints. In the two-level protocol, the L1 coherence controller has 12 states (4 stable, 8 transient), and the L2 has 6 states (3 stable, 3 transient). Figure 3.5a shows the state-transition diagram of the more complex L1 cache. In the three-level protocol, the L1 has 14 states (4 stable, 10 transient), the L2 has 38 (9 stable, 29 transient), and the L3 has 6 (3 stable, 3 transient).

Generalized non-exclusive state: While we have introduced U as an additional state separate from S, both have a strong symmetry and many similarities. In fact, reads are just another type of commutative operation. We leverage this insight to simplify COUP's implementation by integrating S and U under a single, generalized *non-exclusive* state, N. This state requires minor extensions over the machinery already described in Section 3.2 to support multiple commutative updates.

Multiple caches can have a copy of the line in N, but all copies must be under the same operation type, which can be read-only or one of the possible commutative updates. An additional field per line tracks its operation type when in N. Non-exclusive and downgrade requests are tagged with the desired operation type. E and M can satisfy all types of requests; commutative updates cause an E→M transition. N can satisfy non-exclusive requests of the same type, but requests of a different type trigger an invalidation (if starting from read-only) or a reduction (if starting from a commutative-update type) and cause a type switch. Invalidations and reductions involve the same request-reply sequence, so they can use the same transient states.

Implementing two-level MEUSI this way requires 13 states in the L1 and 6 states in the L2. Compared to two-level MESI, MEUSI introduces only one extra L1 transient state. Figure 3.5b shows the L1's state-transition diagram, which is almost identical to MESI's. The new transient state, *NN*, is used when moving between operation types (e.g., from read-only to commutative-add or from commutative-and to commutative-or). Our three-level MEUSI protocol is also similar to three-level MESI: the L1 has 15 states (one more transient than MESI, *NN*), the L2 has 43 (five more transient states than MESI, which, similarly to *NN*, implement transitions between operation types), and the L3 has 6.



(a) Two-level protocols (b) Three-level protocols

Figure 3.6: COUP exhaustive verification costs for two- and three-level protocols. Costs grow much more quickly with the number of cores and levels than the number of commutative updates.

Verification costs: We use Murphi [38] to verify MESI and MEUSI. We adopt common simplifications to limit the state space, modeling caches with a single 1-bit line; self-eviction rules model a limited capacity. In three-level protocols, we model systems with a single L2 and a single L3, and simulate traffic from other L2s with L3-issued invalidation and downgrade rules. Even then, Murphi can only verify systems of up to 4-8 cores, a well-known limitation of this approach [119, 120].

MEUSI's verification costs grow more quickly with the number of cores and levels than the number of commutative operations. Figure 3.6 reports the verification times for two- and three-level MESI and MEUSI protocols supporting 2–20 commutative-update types. We run Murphi on a Xeon E5-2670, and limit it to 16 GB of memory. Murphi can exhaustively verify MESI up to 7-9 cores and MEUSI up to 3-7 cores depending on the number of levels and commutative updates. This shows that MEUSI can be effectively verified up to a large number of commutative updates. Moreover, just as protocol designers assume that modeling a few cores provide reasonable coverage, verifying up to a few commutative operations should be equally reasonable.

3.5 Motivating Applications

In this work, we apply COUP to accelerate single-word updates to shared data. To guide our design, we first study under what circumstances COUP is beneficial over state-of-the-art software techniques, and illustrate these circumstances with specific algorithms and applications.

As discussed in Section 2.2, COUP is the hardware counterpart to privatization. Privatization schemes create several replicas of variables to be updated. Each thread updates one of these replicas, and threads synchronize to reduce all partial updates into a single location before the variable is read.

In general, COUP outperforms prior software techniques if *either* of the following two conditions holds:

- Reads and updates to shared data are finely interleaved. In this case, software privatization has large overheads due to frequent reductions, while COUP can move a line from update-only mode to read-only mode at about the same cost as a conventional invalidation. Thus, privatization needs many updates per core and data value to amortize reduction overheads, while COUP yields benefits with as little as two updates per update-only epoch.
- A large amount of shared data is updated. In this case, privatization significantly increases memory footprint and puts more pressure on shared caches.

We now discuss several parallel patterns and applications that have these properties.

Separate Update- and Read-Only Phases 3.5.1

Several parallel algorithms feature long phases where shared data is either only updated or only read. Privatization techniques naturally apply to these algorithms.

Reduction variables: Reduction variables are objects that are updated by multiple iterations of a loop using a binary, commutative operator (a reduction operator) [87, 88], and their intermediate state is not read. Reduction variables are natively supported in parallel programming languages and libraries such as HPF [58], MapReduce [36], OpenMP [39], TBB [89], and Cilk Plus [42]. Prior work in parallelizing compilers has developed a wide array of techniques to detect and exploit reduction variables [51, 87, 88]. Reductions are commonly implemented using parallel reduction trees, a form of privatization. Each thread executes a subset of loop iterations independently, and updates a local copy of the object. Then, in the reduction phase, threads aggregate these copies to produce a single output variable.

Reduction variables can be small, for example when computing the mean or maximum value of an array. In these cases, the reduction variable is a single scalar, the reduction phase takes negligible time, and COUP would not improve performance much over software reductions.

Reduction variables are often larger structures, such as arrays or matrices. For example, consider a loop that processes a set of input values (e.g., image pixels) and produces a histogram of these values with a given number of bins. In this case, the reduction variable is the whole histogram array, and the reduction phase can dominate execution time [52], as shown in Figure 2.1. Yu and Rauchwerger [115] propose several adaptive techniques to lower the cost of reductions, such as using per-thread hash tables to buffer updates, avoiding full copies of the reduction variable. However, these techniques add time overheads and must be applied selectively [115]. Instead, COUP achieves significant speedups by maintaining a single copy of the reduction variable in memory, and overlapping the loop and reduction phases.

Reduction variables and other update-only operations often use floating-point data. For example, depending on the format of the sparse matrix, sparse matrix-vector multiplication can require multiple threads to update overlapping elements of the output vector [6]. However, floating-point operations are not associative or commutative, and the order of operations can affect the final result in some cases [106]. Common parallel reduction implementations are non-deterministic, so we choose to support floating-point addition in COUP. Implementations desiring reproducibility can use slower deterministic reductions in software [37].

Ghost cells: In iterative algorithms that operate on regular data, such as structured grids, threads often work on disjoint chunks of data and only need to communicate updates to threads working on neighboring chunks. A common technique is to buffer updates to boundary cells using ghost or halo cells [56], private copies of boundary cells updated by each thread during the iteration and read by neighboring threads in the next iteration. Ghost cells are another form of privatization, different from reductions in that they capture point-to-point communication. COUP avoids the overheads of ghost cells by letting multiple threads update boundary cells directly.

The ghost cell pattern is harder to apply to iterative algorithms that operate on irregular data, such as PageRank [78, 98]. In these cases, partitioning work among threads to minimize communication can be expensive, and is rarely done on shared-memory machines [98]. By reducing the cost of concurrent updates to shared data, COUP helps irregular iterative algorithms as well.

3.5.2 Interleaved Updates and Reads

Several parallel algorithms read and update shared data within the same phase. Unlike the applications in Section 3.5.1, software privatization is rarely used in these cases, as software would need to detect data in update-only mode and perform a reduction before each read. By contrast, COUP transparently switches cache lines between read-only and update-only modes in response to accesses, improving performance even with a few consecutive updates or reads.

Graph traversals: High-performance implementations of graph traversal algorithms such as breadth-first search (BFS) encode the set of visited nodes in a bitmap that fits in cache to reduce memory bandwidth [5, 26]. The first thread that visits a node sets its bit, and threads visiting neighbors of the node read its bit to find whether the node needs to be visited.

Existing implementations use atomic-*or* operations to update the bitmap [5], or use non-atomic load-*or*-store sequences, which reduce overheads but miss updates, causing some nodes to be visited multiple times [26]. In both cases, updates from multiple threads are serialized. In contrast, COUP allows multiple concurrent updates to bits in the same cache line.

Besides graph traversals, commutative updates to bitmaps are common in other contexts, such as recently-used bits in page replacement policies [31], buddy memory allocation [57], and other graph algorithms [63].

Reference counting: Reference counting is a common automatic memory management technique. Each object has a counter to track the number of active references. Threads increment the object's counter when they create a reference, and decrement and read the counter when they destroy a reference, When the reference count reaches zero, the object is garbage-collected.

Using software techniques to reduce reference-counting overheads is a well-studied problem [28, 29, 40, 70]. Scalable Non-Zero Indicators (SNZIs) [40] reduce the cost of non-zero checks. SNZIs keep the global count using a tree of counters. Threads increment and decrement different nodes in the tree, and may propagate updates to parent nodes. Readers just need to check the root node to determine whether the count is zero. SNZIs make non-zero checks fast and allow some concurrency in increments and decrements, but add significant space and time overheads, and need to be carefully tuned.

Refcache [28] delays and batches reads to reference counts, which allows it to use privatization. Threads maintain a software cache of reference counter deltas, which are periodically flushed to the global counter. When the global counter stays at zero for a sufficiently long time, the true count is known to be zero and the object is deallocated. This approach reduces reference-counting overheads, but delayed deallocation hurts memory footprint and locality.

COUP enables shared reference counters with no space overheads and less coherence traffic than shared counters. COUP also allows delayed reference counting as in Refcache without a software cache (Section 3.6.4).

3.6 Evaluation

3.6.1 Methodology

Modeled systems: We perform microarchitectural, execution-driven simulation using zsim [95]. We evaluate single- and multi-socket systems with up to 128 cores and a four-level cache hierarchy, shown in Figure 3.7. Table 3.1 details the configuration of these systems. Each processor chip has 16 cores. Each core has private L1s and a private L2, and all cores in the chip share a banked L3 cache with an in-cache directory. The system supports up to 8 processor chips, connected in a dancehall topology to

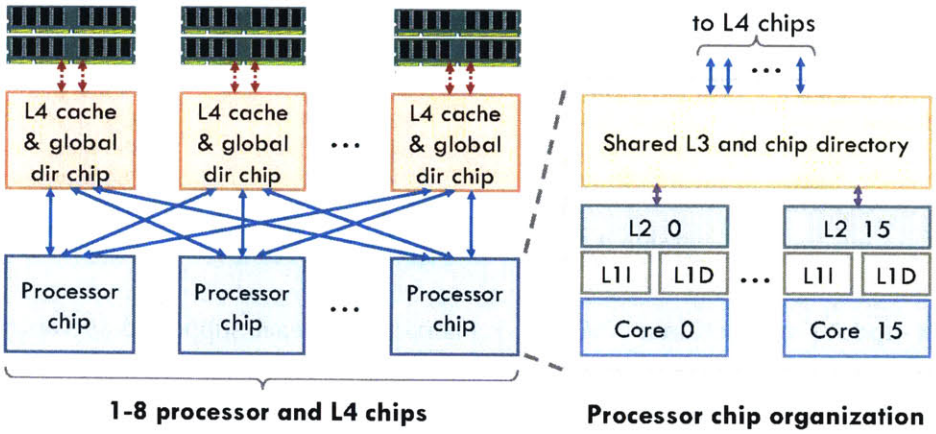


Figure 3.7: Architecture of the simulated system.

Processor chip	**Cores**	1–128 cores, 16 cores/processor chip, x86-64 ISA, 2.4 GHz, Nehalem-like OOO [95]
	L1 caches	32 KB, 8-way set-associative, split D/I, 4-cycle latency
	L2 caches	256 KB private per-core, 8-way set-associative, inclusive, 7-cycle latency
	L3 caches	32 MB, 8 banks, 16-way set-associative, inclusive, 27-cycle latency, in-cache directory
	Off-chip network	Dancehall topology, 40-cycle point-to-point links between each processor and L4 chip
	L4 & dir chip	128 MB, 8 banks/chip, 16-way set-associative, inclusive, 35-cycle latency, in-cache directory
	Coherence	MESI/MEUSI, 64 B lines, no silent drops
	Main memory	4 DDR3-1600-CL10 channels per L4 chip, 64-bit bus, 2 ranks/channel

Table 3.1: Configuration of the simulated system.

the same number of L4 chips. Each of these chips contains a slice of the L4 cache and global in-cache directory, and connects to a fraction of main memory. This organization is similar to the IBM z13 [110].

We compare MESI and MEUSI (Figure 3.4). With MEUSI, each L3 and L4 bank has a reduction unit. We perform hierarchical reductions as described in Section 3.2: on a full reduction, each L3 bank invalidates all its children, aggregates their partial updates, and sends a single response to the L4 controller.

COUP operations and data types: We add support for eight commutative-update types:

- Addition of 16, 32, and 64-bit integers, and 32 and 64-bit floating-point values.
- AND, OR, and XOR bitwise logical operations on 64-bit words.

We observe multiplication update-only operations are rare, so we do not support multiplication. We also observe *min* and *max* are often used with scalar reduction variables (e.g., to find the extreme values of an array). COUP would provide a negligible benefit for scalar reductions, as discussed in Section 3.5.1. Thus, we do not support *min* or *max*. Finally, we support a single word size for bitwise operations, because this suffices to express updates to bitmaps of any size (smaller or larger).

	Input set	Commutative operations	Sequential runtime
hist	GRiN [1], 512 bins	32b int add	2720 Mcycles
spmv	rma10 [35]	64b FP add	94 Mcycles
fluidanimate	simlarge [13]	32b FP add	5930 Mcycles
pgrank	Wikipedia (2007) [35]	64b int add	2850 Mcycles
bfs	cage15 [35, 67]	64b OR	5764 Mcycles

Table 3.2: Benchmark characteristics.

Commutative-update instructions: We add an instruction for each supported operation and data type. Each instruction takes two register inputs, with the address to be updated and the value to apply, and produces no register output. We encode these instructions using x86-64 no-ops that are never emitted by the compiler.

The x86 (TSO) memory model specifies that atomic instructions have an implicit store-load fence [100]; for consistency, we also add an implicit fence to commutative-update instructions. We implement conventional atomic operations and commutative updates using a four-μop sequence: load-linked, execute (in one of the appropriate execution ports), store-conditional, and store-load fence.

Reduction unit organization: Since functional units for the required operations are relatively simple, we assume a 2-stage pipelined, 256-bit ALU (4× 64-bit lanes). This ALU has a throughput of one full 64-byte cache line per two clock cycles, and a latency of three clock cycles per line. We explore the sensitivity to reduction unit throughput in Section 3.6.5.

Hardware overheads: In summary, our COUP implementation introduces modest overheads:

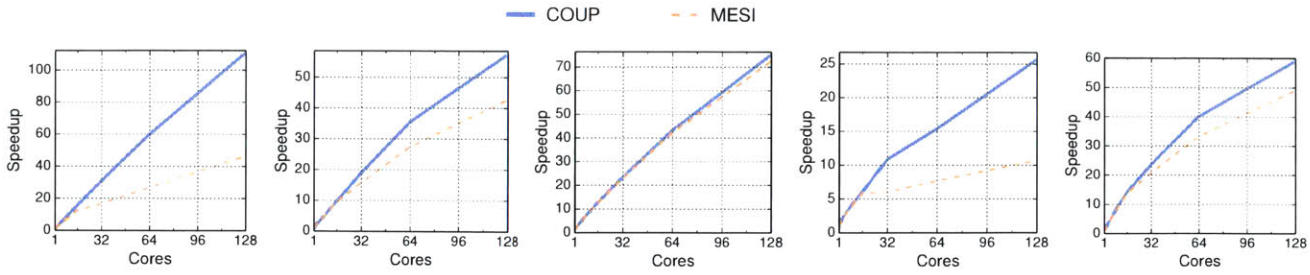
1. Eight additional commutative-update instructions.
2. Four bits per line to encode the non-exclusive operation type, either read-only or one of eight commutative-update types (Section 3.4).
3. One reduction unit per L3 and L4 bank.

Workloads: We use a set of five multithreaded benchmarks that cover the cases described in Section 3.5:

- **hist** is the TBB-based OpenCV [17] histogramming program (version 2.4.11).
- **spmv** is a sparse matrix-vector multiplication kernel, where the matrix is encoded in compressed sparse column (CSC) format. CSC requires multiple threads to perform scattered additions to the output vector. Other input formats, such as EBE, also cause scattered adds in matrix-vector multiplication [6].
- **fluidanimate**, from the PARSEC suite [13], is a regular iterative algorithm (Section 3.5.1). We optimize the default implementation, which uses locks to guard updates to shared cells, to use atomic operations instead.
- **pgrank** is a PageRank implementation similar to the shared-memory optimized version of Satish et al. [98].
- **bfs** is a parallel breadth-first search algorithm. Our implementation extends PBFS [67] with a visited bit-vector to reduce memory traffic (Section 3.5.2), similar to state-of-the-art approaches [5, 26].

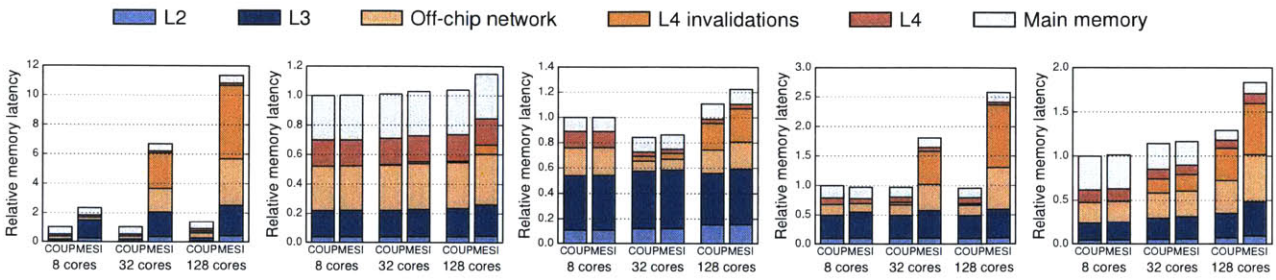
Table 3.2 details the input sets, commutative-update operations used, and sequential runtime of each benchmark.

All the baseline benchmark implementations use atomic operations. We also compare against a privatization-based variant of `hist` (implemented using TBB reductions) in Section 3.6.3, and develop reference-counting microbenchmarks to compare COUP against SNZI and Refcache in Section 3.6.4.



(a) hist (b) spmv (c) fluidanimate (d) pgrank (e) bfs

Figure 3.8: Per-application speedups of COUP and MESI on 1–128 cores (higher is better).



(a) hist (b) spmv (c) fluidanimate (d) pgrank (e) bfs

Figure 3.9: Breakdown of average memory access latency (AMAT) of COUP and MESI on 8, 32, and 128-core systems. AMAT is normalized to COUP's at 8 cores (lower is better).

We report results on 1–128 cores. We scale the number of processor and L4 chips on runs with more cores (e.g., 1-core runs use a single processor and L4 chip, 32-core runs use two of each, and so on), which also scales the bandwidth of the memory system and L4 capacity. To achieve statistically significant results, we introduce small amounts of non-determinism as proposed by Alameldeen and Wood [7], and perform enough runs to achieve 95% confidence intervals ≤1%.

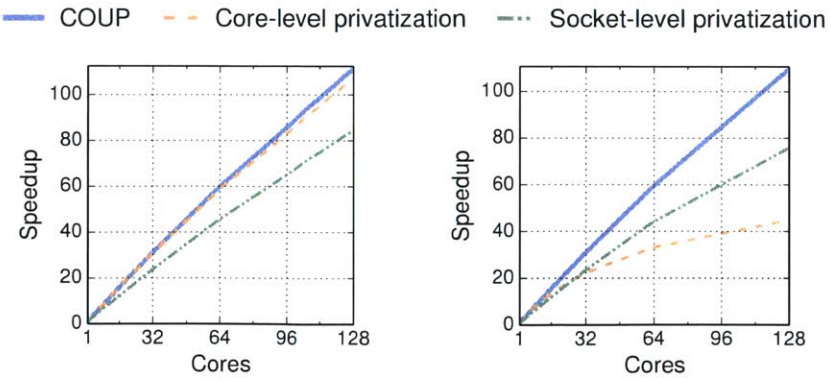
Comparison Against Atomic Operations 3.6.2

Figure 3.8 compares the performance and scalability of COUP and a conventional MESI protocol. Each graph shows results for a single application, and each line in the graph shows how performance scales for a particular scheme (MESI or COUP) as the number of cores grows from 1 to 128 (x-axis). All speedup numbers are relative to the runtime of the application on a single core under MESI. Higher numbers are better.

Figure 3.8 shows that COUP always outperforms MESI, often substantially. At 128 cores, COUP outperforms MESI by 2.4× on `hist`, 34% on `spmv`, 4.0% on `fluidanimate`, 2.4× on `pgrank`, and 20% on `bfs`. Moreover, the gap between MESI and COUP often widens as the number of cores grows, showing that COUP has better scalability than MESI.

COUP is especially beneficial for applications where shared data goes through long update-only phases. This is the case with `hist`, `spmv`, and `pgrank`. In `bfs`, where cache lines are constantly moving between U and S states as cores update and check the visited bit-vector (Section 3.5.2), COUP's advantage is lower but still significant. Finally, shared cells in `fluidanimate` experience long read-only and update-only phases, but only a fraction of cells are shared, and shared cells see few updates from neighboring threads on each update-only phase, so COUP provides a small speedup over MESI.

Figure 3.9 gives more insight into these results by showing the breakdown of average memory access latency (AMAT). Each graph shows results for a single application. Each set of two bars shows results



(a) 512 bins (b) 16K bins

Figure 3.10: Speedups of `hist` with COUP and both core- and socket-level privatization, using small (512) and large (16 K) numbers of bins.

for COUP and MESI for a given system size (8, 32, or 128 cores). The height of each bar is the average memory access latency of all loads, stores, and instruction fetches issued from the L1s, normalized to the AMAT that COUP achieves at 8 cores. Each bar is broken down into time spent at the L2, L3, off-chip network, L4, coherence invalidations from the L4, and main memory. This breakdown shows critical-path delays only (e.g., the time spent on invalidations is not the time spent on every invalidation, but the critical-path delay that L4 requests suffer because other sharers need to be invalidated or downgraded).

Figure 3.9 shows that COUP substantially reduces AMAT over MESI. At 128 cores, COUP's AMAT is lower than MESI's by 12.6× on `hist`, 10% on `spmv`, 12% on `fluidanimate`, 3.0× on `pgrank`, and 24% on `bfs`. COUP mainly does this by reducing invalidations and serialization. The effect of this reduction on the overall AMAT depends on how the application uses the memory system. For instance, COUP nearly eliminates invalidation traffic in `hist`, `spmv`, and `pgrank`. In `hist` and `pgrank`, invalidations are the dominant contributor to AMAT, so eliminating them has the largest impact. But AMAT in `spmv` is dominated by L4 and main memory accesses, so the overall impact of eliminating invalidations is smaller.

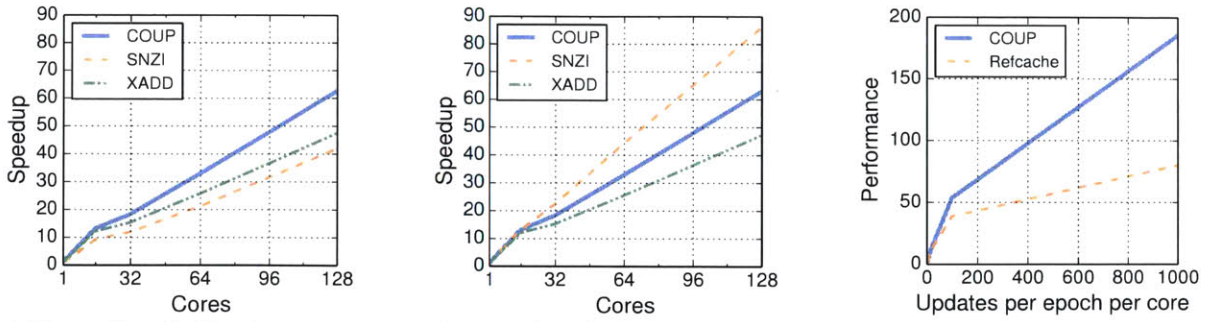
Beyond reducing AMAT, COUP also lowers traffic: at 128 cores, COUP incurs lower off-chip traffic than MESI by a factor of 20.2× on `hist`, 18% on `spmv`, 18% on `fluidanimate`, 4.9× on `pgrank`, and 20% on `bfs`.

Finally, even though COUP's benefits are significant, these benchmarks execute a relatively small fraction of commutative-update instructions: at 128 cores, commutative-update instructions are 1.0% of all executed instructions on `hist`, 2.4% on `spmv`, 0.96% on `fluidanimate`, 4.9% on `pgrank`, and 0.40% on `bfs`. Their impact is significant because, at large core counts, each atomic read-modify-write to a contended memory location can take several hundred cycles.

3.6.3 Case Study: Reduction Variables

All baseline benchmarks use atomic operations instead of privatization. To compare COUP with software privatization, we modify `hist` to make the histogram a reduction variable, and vary the number of bins (elements) in the histogram. We evaluate both core-level privatization, where each thread has its own variable, and socket-level privatization, where each socket has its own variable, shared and updated by all threads running in that socket using atomic operations. Socket-level privatization seeks to balance the overheads of the fully-shared and fully-privatized implementations.

Figure 3.10 compares the performance and scalability of COUP with core-level and socket-level privatization on `hist`. Figure 3.10a shows that, with a small number of bins, COUP outperforms core-level privatization by 3% and socket-level privatization by 38%. Core-level privatization works well



(a) Immediate dealloc, low count (b) Immediate dealloc, high count (c) Delayed dealloc

Figure 3.11: Performance of COUP on reference counting microbenchmarks: (a, b) immediate deallocation and (c) delayed deallocation.

in this case because each thread performs many updates to each histogram bin (128 on average), so reduction overheads are highly amortized.

In contrast, Figure 3.10b shows that, with a large number of bins, COUP outperforms core-level privatization by 2.5× and socket-level privatization by 51%. In this case, core-level privatization is dominated by the cost of reductions, as each thread performs a small number of updates to each histogram bin (2 on average).

Finally, privatization also increases footprint and adds pressure to shared caches. If we grow both the number of bins and the image size (so the number of updates per bin and thread, and thus reduction overheads, stay constant), we see an additional performance degradation of 9% in the core-level privatized version when the aggregate size of all privatized histograms overflow the L3 caches, while COUP does not suffer this degradation.

Case Study: Reference Counting 3.6.4

We use two microbenchmarks to compare COUP's performance on reference counting against the software techniques described in Section 3.5.2. The first microbenchmark models immediate-deallocation schemes, and we use it to compare against a conventional atomic-based implementation and SNZI [40]. The second microbenchmark models delayed-deallocation schemes, and we use it to compare against Refcache [28].

Immediate deallocation: In this microbenchmark, each thread performs a fixed number of increment, decrement, and read operations over a fixed number of shared reference counters. We use 1 to 128 threads, 1 million updates per thread, and 1024 shared counters. On each iteration, a thread selects a random counter and performs either an increment or a decrement and read.

SNZI uses binary trees with as many leaves as threads. The performance of SNZI depends on the number of references per object—a higher number of references causes higher surpluses in leaves and intermediate nodes, and less contention on updates. To capture this effect, we run two variants of this benchmark. In the first variant (low count), each thread keeps only 0 or 1 references per object, while in the second mode (high count), each thread keeps up to five references per object.

To achieve this, in low-count mode, when a thread randomly selects an object, it will always increment its counter if it holds no references to that object, and it will always decrement its counter if it holds one reference. In high-count mode, threads will increment with probability 1.0, 0.7, 0.5, 0.5, 0.3, and 0.0 if they hold 0, 1, 2, 3, 4, and 5 local references to that counter, respectively.

For updates, COUP and XADD use commutative-add and atomic fetch-and-add instructions, respectively.

Figure 3.11a and Figure 3.11b show the results for these experiments. In the low-count variant (Figure 3.11a), SNZI incurs high overhead when counts drop to zero, so both COUP and XADD outperform SNZI (by 50% and 17% at 128 cores, respectively). By contrast, in the high-count variant (Figure 3.11b), SNZI enjoys lower contention and outperforms COUP (by 35% at 128 cores). COUP outperforms XADD in both cases.

We conclude that, in high-contention scenarios, COUP provides the highest performance, but in specific scenarios, software optimizations that exploit application-specific knowledge to avoid contention among reads and updates can outperform COUP. We also note that it may be possible to modify SNZI to take advantage of COUP and combine the advantages of both techniques.

Delayed deallocation: In the delayed-deallocation microbenchmark, 128 threads perform increments and decrements (but not reads) on 100,000 counters. We divide the benchmark into epochs, each with a given number of updates per thread. When they finish an epoch, threads check whether counters are zero, simulating delayed-deallocation periods as in Refcache [28].

Our COUP implementation updates counters with commutative-add instructions and maintains a bitmap with a "modified" bit for each counter. The bitmap is updated with commutative-or instructions. Between epochs, cores use ordinary loads to read the value of marked counters and check whether the counters are zero. Refcache uses a per-thread software cache (a hash table) to maintain the deltas to each modified counter. Threads flush this cache when they finish each epoch.

Figure 3.11c shows the performance COUP and Refcache on the delayed deallocation microbenchmark as the number of updates per epoch (x-axis) grows from 1 to 1000 updates per thread and epoch. COUP outperforms refcache across the range, by up to 2.3×.

We conclude that COUP primarily helps delayed-deallocation reference counting by allowing a simpler, lower-overhead implementation to capture the low communication costs of prior software approaches (in this case, using counters and bitmaps instead of hash tables).

3.6.5 Sensitivity to Reduction Unit Throughput

COUP is barely sensitive to reduction unit throughput. We compare the default 256-bit ALU, which has a throughput of one cache line per 2 cycles, with a simpler, unpipelined 64-bit ALU, which has a throughput of one line per 16 cycles. The maximum performance degradation incurred with the slower ALU is 0.88% at 128 cores on `bfs`. Smaller systems incur somewhat lower worst-case degradations (e.g., 0.76% at 64 cores).

3.7 Additional Related Work

Loosely consistent memory (LCM) [64] is a software-controlled coherence protocol built on top of Tempest [90] that allows multiple caches to hold writable copies of the same line. These copies can become incoherent, and software must explicitly reconcile them in a later merge phase. Unlike LCM, COUP preserves cache coherence and transparently merges partial updates, requiring no software intervention.

Moreover, several cache-coherence optimizations reduce the cost of updates, though that is not their primary purpose: self-invalidations, done with either hardware predictors [66] or software protocols [27, 54], remove invalidations from the critical path; adaptive-granularity coherence schemes [61, 117, 121] reduce both false sharing and the amount of dirty data sent on invalidations; and speculation and fast

networks can reduce the cost of atomic operations [41]. These schemes are orthogonal to COUP, which could be used in conjunction with them to improve performance.

While we have focused on shared-memory systems, exploiting commutativity is also common with message passing. The BlueGene/L and BlueGene/Q supercomputers feature specialized collective networks that perform reductions completely in hardware, using ALUs embedded in network routers [8, 24]. In contrast to COUP, their main advantage is minimizing the latency of scalar or short reductions across a very large number of nodes.

Summary 3.8

We have presented COUP, a technique that exploits commutativity to reduce the cost of updates in cache-coherent systems. COUP extends conventional coherence protocols to allow multiple caches to simultaneously hold update-only permission to data. We have introduced an implementation of COUP that uses this support to accelerate single-instruction commutative updates. This implementation requires minor hardware changes and, in return, substantially improves the performance of update-heavy applications.

Beyond this specific implementation, a key contribution of COUP is to recognize that it is possible to allow multiple concurrent updates without sacrificing cache coherence or relaxing the consistency model. Thus, COUP attains performance gains without complicating the parallel programming.

COMMTM

4

As illustrated in Section 2.3, there is a dichotomy between software and hardware conflict detection schemes: software techniques, such as semantic locking, can leverage program semantics to be highly precise, but they incur high runtime overheads; meanwhile, hardware techniques incur small overheads, but are imprecise because they rely on conventional coherence protocols, which can reason in terms of only reads and writes, to detect conflicts.

To solve this dichotomy, 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.

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.

Specifically, we make the following contributions:

- We present a basic version of COMMTM (Section 4.1 to Section 4.5) that achieves the same precision as software *semantic locking* [59, 111].
- We then extend COMMTM with *gather requests* (Section 4.6), which allow software to redistribute reducible data among caches, achieving much higher concurrency in important use cases.
- We evaluate COMMTM with microbenchmarks (Section 4.8) and full TM applications (Section 4.9). Microbenchmarks show that COMMTM scales on a variety of commutative operations, such as set insertions, reference counting, ordered puts, and top-K 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.

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.

4.1 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 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. Figure 1.2 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 4.4. 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 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. 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 [73]).

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);
  }
}
```

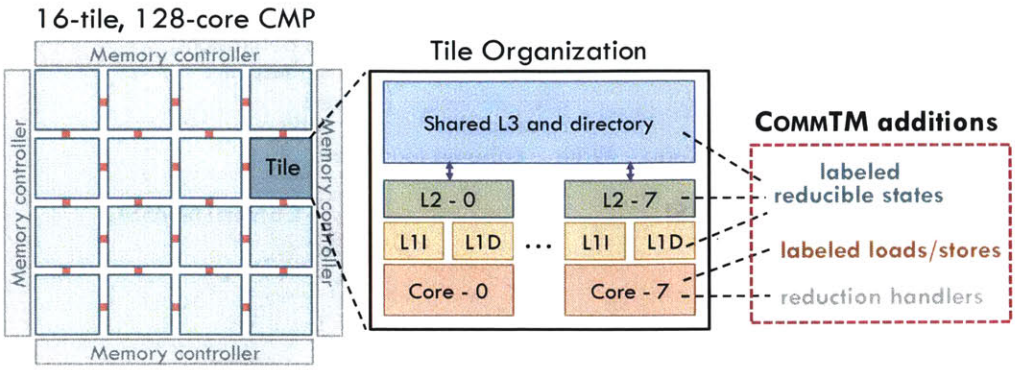


Figure 4.1: Baseline CMP and main CommTM additions.

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).

CommTM Implementation 4.2

Eager-Lazy HTM Baseline 4.2.1

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 lazy (buffer-based) version management, as in LTM [9] and Intel's TSX [114]. We assume a multicore system with per-core private L1s and L2s, and a shared L3, as shown in Figure 4.1. 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 [72]: 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 [16].

Coherence protocol 4.2.2

CommTM extends the coherence protocol with an additional state, *user-defined reducible (U)*. For example, Figure 4.2 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 (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 4.2, 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 stateHowever, 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-

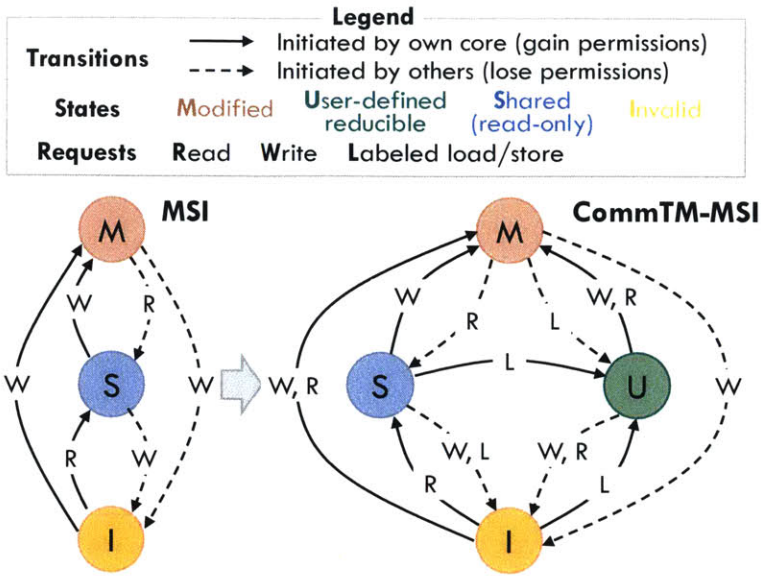


Figure 4.2: State-transition diagrams of MSI and CommTM protocols. For clarity, diagrams omit actions that do not cause a transition (e.g., R requests in S).

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.

4.2.3 Transactional execution

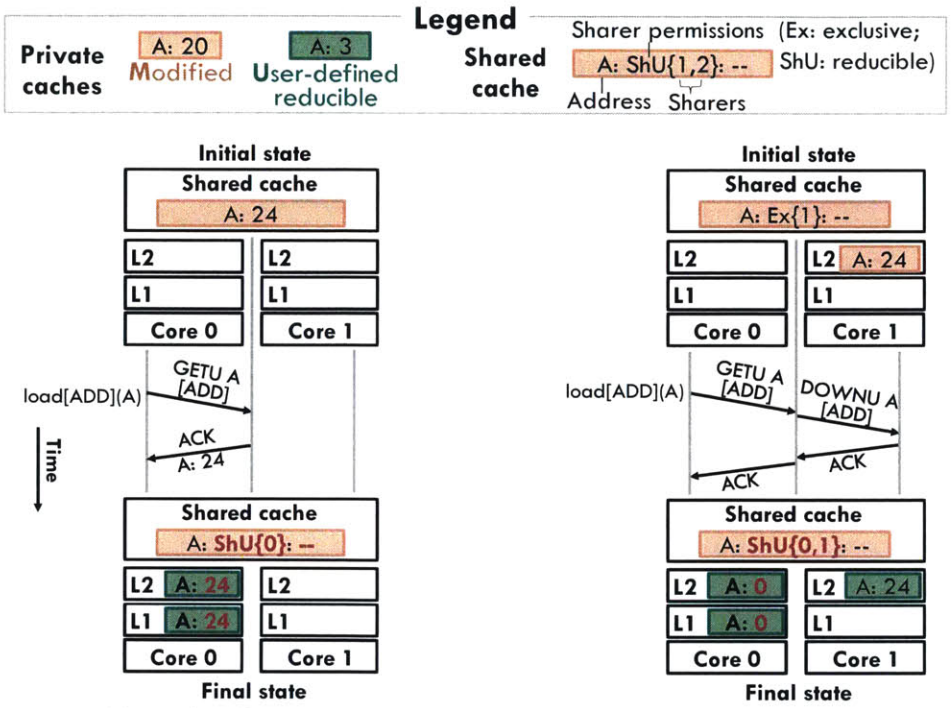
Labeled memory operations within transactions cause lines to enter the U state. We first discuss of permissions change in the absence of transactional conflicts, then explain how conflict detection changes.

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 4.3a.
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 detail in Section 4.2.4.
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 4.3b.

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 4.4 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



(a) No other sharers

(b) Downgrade from M to U

Figure 4.3: 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.

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 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 transaction's *labeled set*. We leverage the existing L1's status bits to track the labeled set, as shown in Figure 4.4.

COMMTM is orthogonal to the conflict resolution protocol. We leverage our baseline's timestamp-based resolution approach: each transaction is assigned an 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 4.5 shows both of these cases in detail for a line in the labeled set.

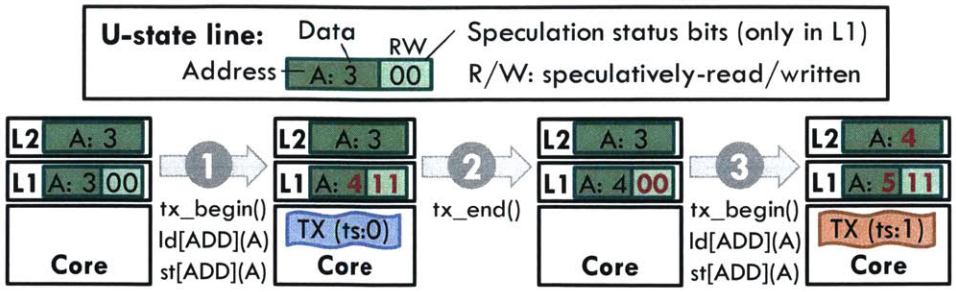


Figure 4.4: Value management for U-state lines is similar to M. L1 tag bits record whether the line is speculatively read or written (using the state and label to infer whether from labeled or unlabeled instructions). Upon commit, spec-R/W bits are reset to zero. Before being written by another transaction, dirty U-state lines are written back to the L2.

4.2.4 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.

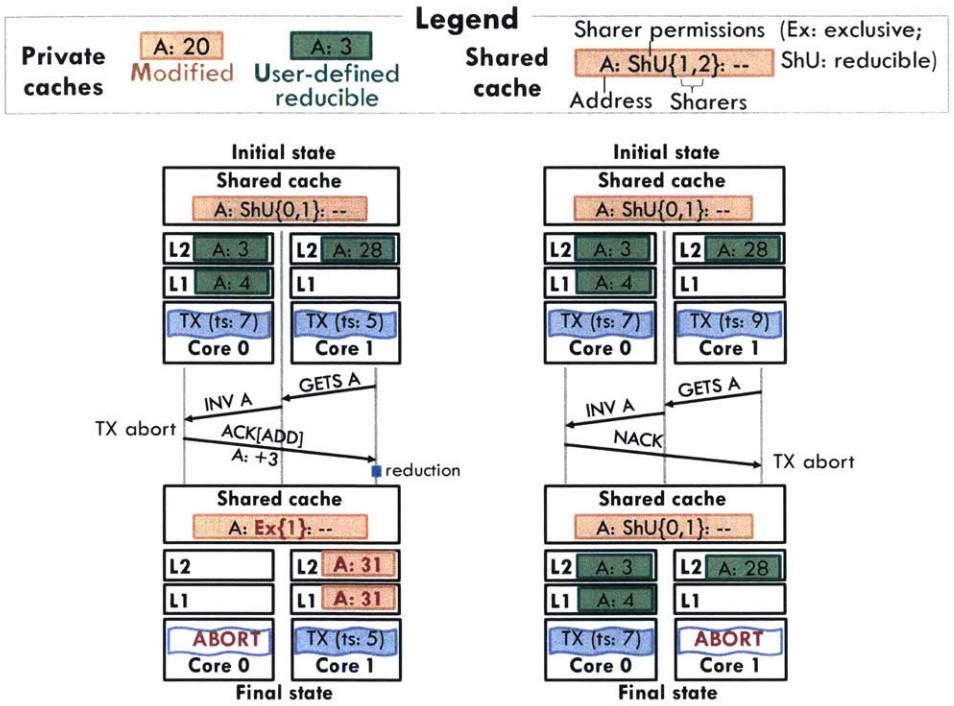
We choose to perform reductions at the core that issues the reduction-triggering request. Specifically, each core features a *shadow hardware thread* dedicated to perform reductions. Figure 4.6 shows the steps of a reduction in detail: ① When the directory receives a reduction-triggering request, it 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 [69, 96, 113], or use a low-performance helper core [10, 25].

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 4.5b. 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 4.8 and Section 4.9).

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 4.1, 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



(a) INV from lower-timestamp transaction

(b) INV from higher-timestamp transaction

Figure 4.5: 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.

dedicate an extra virtual network for forwarded U-state data. This adds moderate buffering requirement to on-chip network routers [80], which must already support 3-6 virtual networks in conventional protocols [14, 75, 102]. 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 [4, 69, 96, 108] (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

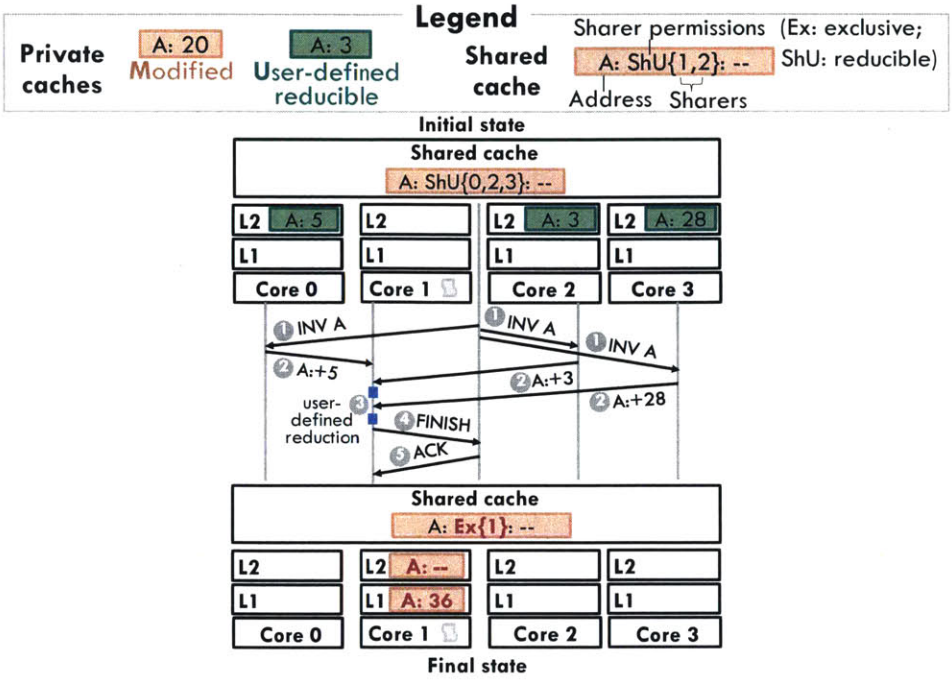


Figure 4.6: Core 0 issues unlabeled or differently-labeled request, causing a full reduction of A's U-state data, held in several private caches.

non-speculative reductions), we do not observe this behavior in any of our use cases.

4.2.5 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.

4.3 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)
- One extra virtual network for forwarded U data, which adds few KBs of router buffers across the system [34].

- 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% [48]). 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.

Generalizing COMMTM 4.4

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 stateIn 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 [22, 44] or Bulk [21, 82]. 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.

COMMTM vs Semantic Locking 4.5

Just as eager conflict detection is the hardware counterpart to two-phase locking [11, 47], COMMTM as described so far is the hardware counterpart to semantic locking (Section 2.3). 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 [59] (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 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.

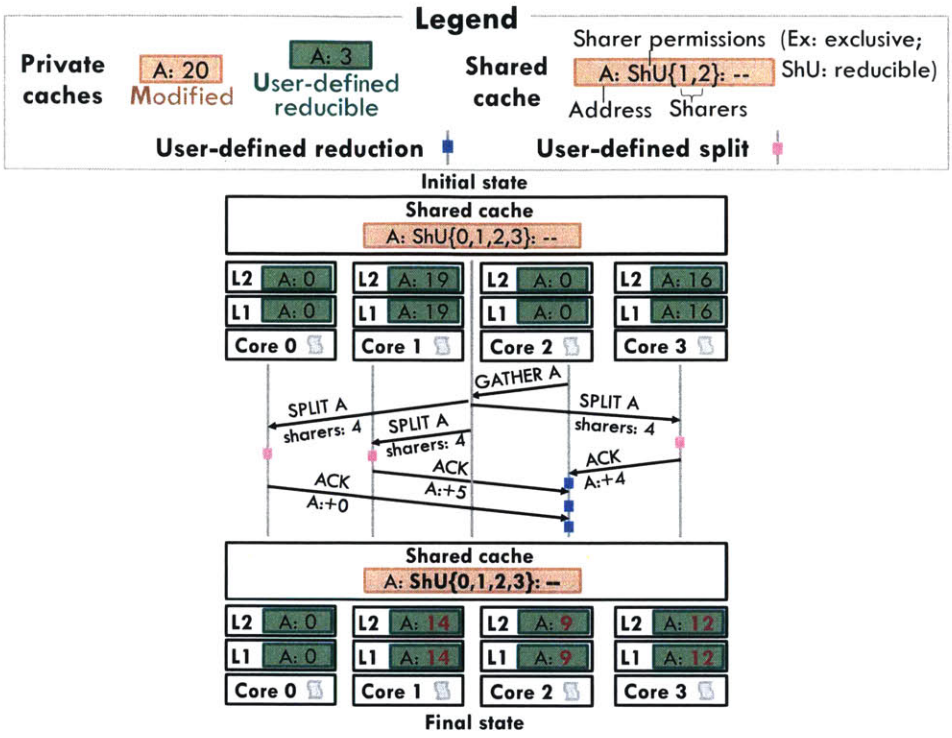


Figure 4.7: Gather requests collect and reduce U-state data from other caches. In this example, core 2 initiates a gather to satisfy a local decrement. User-defined splitters at other cores donate part of their local deltas to core 2. For instance, core 3 splits its initial value, 16, into 12, which it retains, and 4, which it donates.

4.6 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 [59]. 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` 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.

Gather requests: Figure 4.7 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)
      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:

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 4.1: Configuration of the simulated system.

```
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 4.7 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 `decrements` 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.

4.7 Experimental Methodology

As in Chapter 3, we perform microarchitectural, execution-driven simulation using zsim. We evaluate a 16-tile CMP with 128 simple cores and a three-level memory hierarchy, shown in Figure 4.1, with parameters given in Table 4.1. 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 [114] 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 (introduced in Section 4.8) and full-blown TM applications (discussed in Section 4.9). 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 [7], and perform enough runs to achieve 95% confidence intervals $\leq 1\%$ on all results.

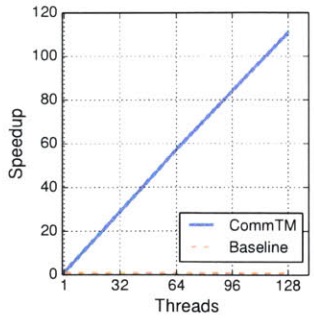


Figure 4.8: Speedup of counter microbenchmarks.

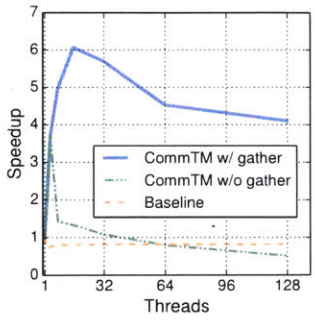
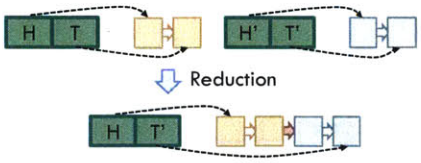
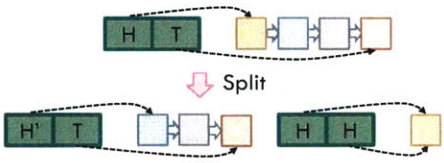


Figure 4.9: Speedup of reference-counting microbenchmark.



(a) Reducing a list descriptor



(b) Splitting a list descriptor

Figure 4.10: A linked-list descriptor contains its head and tail pointers, and can be shared in U states by multiple caches. Each U-state copy represents a partial linked list. A reduction merges all partial lists and generate the resulting descriptor, and a split divides the partial list into two: one only containing the previous head element and the other containing the rest.

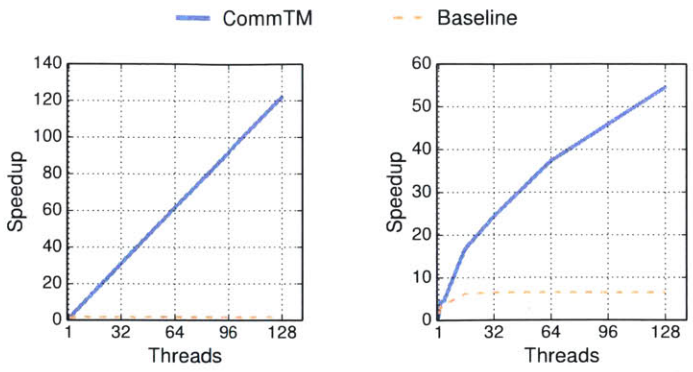
4.8 CommTM on Microbenchmarks

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 Section 4.2. Figure 4.8 shows that COMMTM achieves linear scalability, while the baseline HTM serializes all transactions. While counters are our simplest case, prior work reports that counter updates are a major cause of aborts in real applications [30, 93].

Reference counting: We implement a reference counter using the non-negative bounded counter described in Section 4.6, with and without gather requests. 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 holds 0, 1, 2, 3, 4, and 5 local references, respectively, and decrements it otherwise. Figure 4.9 shows that the baseline HTM achieves no speedup, and COMMTM without gather requests provides some speedup with 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 3.7× at 128 threads. The sub-linear scalability is due to more frequent gather requests and splits at high thread counts.

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. Figure 4.10a shows how COMMTM



(a) 100% enqueues

(b) 50% enqueues, 50% dequeues

Figure 4.11: Speedup of linked list microbenchmark under baseline HTM and COMMTM.

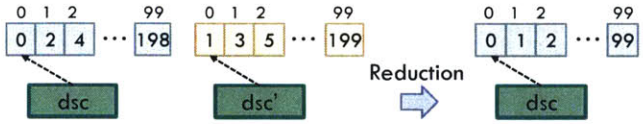


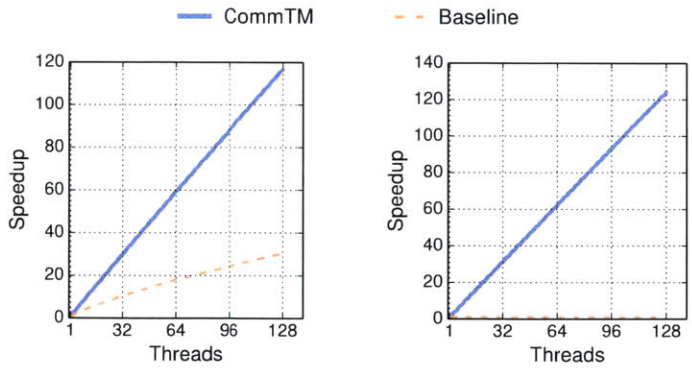
Figure 4.12: A top-K set descriptor with $K = 100$.

makes these operations concurrrent. 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, each reducible, local descriptor has its own tail pointer, and threads can enqueue/dequeue elements locally. Figure 4.10a 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 Figure 4.10b.

Figure 4.11 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 Figure 4.11a, or 50% enqueues and 50% dequeues (randomly interleaved) in Figure 4.11b. 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 [77] and are key in challenging parallel algorithms [103]. 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). Figure 4.13a 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).

Top-K: A *top-K set*, common in databases, contains the K highest elements of a set [77]. We implement insertions to a top-K set similarly to the linked-list: a descriptor contains a pointer to the top-K data (stored as a heap), and only the descriptor uses reducible states. Threads build up local top-K heaps, and reads trigger a reduction that merges all local heaps, as shown in Figure 4.12.



(a) Ordered put (b) Top K

Figure 4.13: Speedups of (a) an ordered put benchmark, and (b) a top-K insertion benchmark.

	Input set	Uses gather?	Commutative operations
boruvka	usroads [35]	✗	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 [71]	✗	Updating cluster centers(32b `ADD`, 32b `FP ADD`)
ssca2	-s16 -i1.0 -u1.0 -l9 -p9 [71]	✗	Modifying global information for a graph (32b `ADD`)
genome	-g4096 -s64 -n640000 [71]	✓	Remaining-space counter of a resizable hash table (bounded 64b `ADD`)
vacation	-n4 -q60 -u90 -r32768 -t8192 [71]	✓	Remaining-space counter of a resizable hash table (bounded 64b `ADD`)

Table 4.2: Benchmark characteristics.

Figure 4.13b 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 124× speedup at 128 threads.

CommTM on Full Applications 4.9

We evaluate COMMTM on several TM benchmarks: `boruvka` [59], and `genome`, `kmeans`, `ssca2`, and `vacation` from STAMP [71]. Table 4.2 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. We compile `genome` and `vacation` with resizable hash tables (similar to Blundell el al.[15]), which use conditionally-commutative updates to a bounded counter to determine when to resize.

Figure 4.14 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 1–128 threads (x-axis). As before, all speedups are relative to the performance of a sequential execution in the baseline HTM. Figure 4.14 shows 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 4.15 gives more insight into these results by showing the breakdown of total 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 to the baseline's at 8 threads. Lower bars are better.

Figure 4.15 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.

The breakdown of total cycles explains why COMMTM has little impact on performance of ssca2: contention is rare and therefore only a small fraction of cycles are spent on aborted transactions.

Figure 4.16 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. Figure 4.17 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 45% on kmeans. This also explains why non-transactional cycles are lower in Figure 4.15 (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.

4.10 Additional Related Work

Prior work in hardware speculation, especially 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 [86], SONTM [11], Wait-n-GoTM [50], and OmniOrder [83], 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 [68] relaxes serializability and implements snapshot isolation, which only flags write-write dependences as conflicts. SI-TM, like other schemes that weaken serializability [3, 104], 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

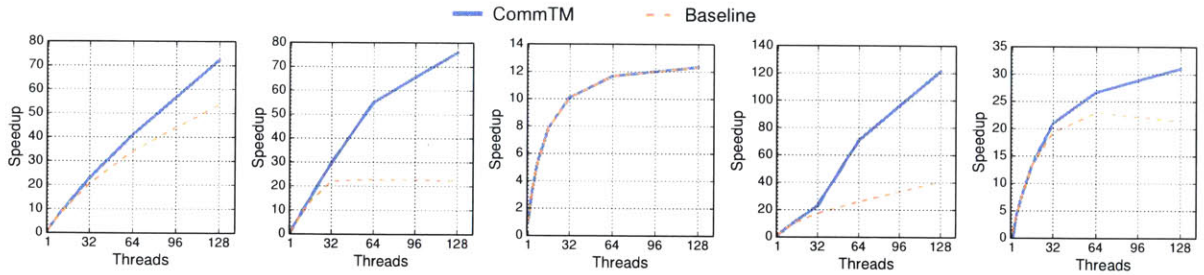


Figure 4.14: Per-application speedups of COMMTM and baseline HTM on 1–128 threads (higher is better).

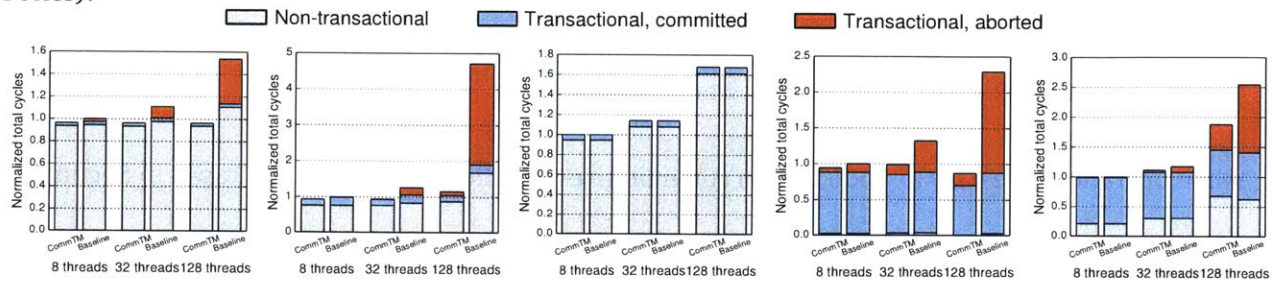


Figure 4.15: Breakdown of total cycles for COMMTM and baseline HTM for 8, 32, and 128 threads (lower is better).

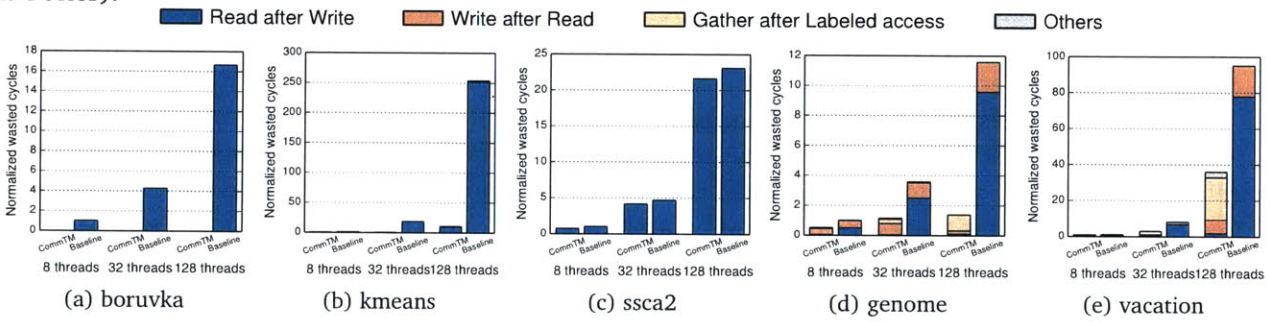
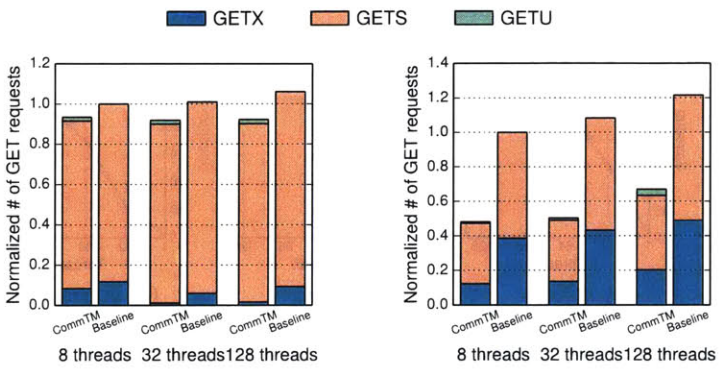


Figure 4.16: Breakdown of wasted cycles for COMMTM and baseline HTM for 8, 32, and 128 threads (lower is better).

(e.g., unlike COMMTM, SI-TM bottlenecks on kmeans [68]).

Other techniques focus on reducing the cost of mispeculation. ReSlice [97] reexecutes only the conflicting load and its dependent instructions, and RetCon [15] 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 [73, 74] 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



(a) boruvka (b) kmeans

Figure 4.17: 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).

when operations are easy to undo, which commutative operations may lack (e.g., top-K in Section 4.8).

4.11 Summary

We have presented COMMTM, an HTM that exploits semantic commutativity to allow multiple transactions updating shared data concurrently and without conflicts. COMMTM extends the coherence protocol and the conflict detection scheme and preserves transactional guarantees. Moreover, COMMTM's basic scheme allows as much concurrency as semantic locking. Gather requests allow COMMTM to reduce conflicts even further.

COMMTM bridges the precision-overhead dichotomy of hardware vs software conflict detection: As a result, 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. At 128 cores, COMMTM outperforms an eager-lazy HTM by up to 3.4× and reduces or even eliminates aborts.

Conclusion

5

This thesis has presented novel techniques to exploit commutativity in shared-memory systems without the overheads of software techniques. In particular, we have presented the following contributions:

- COUP is a general technique that extends coherence protocols to allow local and concurrent single-instruction commutative updates. Specifically, COUP decouples read and write permissions, and introduces commutative-update primitive operations, in addition to reads and writes. With COUP, multiple caches can acquire a line with update-only permission, and satisfy commutative-update requests locally, buffering and coalescing updates. On a read request, the coherence protocol gathers all the local updates and reduces them to produce the correct value before granting read permission.

  COUP integrates seamlessly into existing coherence protocols, requires inexpensive hardware, preserves coherence and does not affect the memory consistency model. Simulation results on a 128-core system show that COUP accelerates update-heavy applications by up to 2.4×. Meanwhile, COUP lowers traffic by up to 20× and reduces memory access latency by up to 12×.

- COMMTM is a hardware transactional memory 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 an unlimited number of user-defined multi-instruction 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. We have shown that COMMTM's basic scheme allows as much concurrency as semantic locking, and gather requests allow COMMTM to reduce even more conflicts.

  COMMTM bridges the precision-overhead dichotomy of hardware vs software conflict detection. In return, COMMTM scales many operations that serialize in conventional HTMs, 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.

These contributions enable shared-memory systems that understand and exploit both strict and semantic commutativity. Beyond these specific implementations, a key contribution of this thesis is to recognize that it is possible to make the memory interface more expressive to exploit high-level properties of operations with reasonable extra complexity. In return, the improved expressivity enables parallel systems to approach the minimal communication and synchronization truly required by the algorithms they run.

COUP and COMMTM open exciting avenues for future research. Prior work has developed a rich set of conflict detectors that go beyond COMMTM's current capabilities. Since COMMTM demonstrates

that hardware speculation can also benefit from conflict-detection techniques that have traditionally been considered software-only. It would be interesting to see how many of these techniques can also be easily adapted by hardware. Moreover, while this thesis focuses on commutativity, exploiting other high-level properties is promising. This may include topics such as utilizing idempotence or eliminating dynamically dead updates. We leave these explorations to future work.

Bibliography

[1] GReat Images in NASA (GRiN), http://grin.hq.nasa.gov.

[2] HSA Platform System Architecture Specification. Technical report, HSA Foundation, 2015.

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

[4] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. In *Proc. of the 22nd annual Intl. Symp. on Computer Architecture (ISCA-22)*, 1995.

[5] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader. Scalable graph exploration on multicore processors. In *Proc. of the ACM/IEEE conf. on Supercomputing (SC10)*, 2010.

[6] J. H. Ahn, M. Erez, and W. Dally. Scatter-add in data parallel architectures. In *Proc. of the 11th IEEE intl. symp. on High Performance Computer Architecture (HPCA-11)*, 2005.

[7] A. Alameldeen and D. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4), 2006.

[8] G. Almási, P. Heidelberger, C. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *Proc. of the Intl. Conf. on Supercomputing (ICS'05)*, 2005.

[9] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. of the 11th IEEE intl. symp. on High Performance Computer Architecture (HPCA-11)*, 2005.

[10] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-32)*, 1999.

[11] U. Aydonat and T. S. Abdelrahman. Hardware support for relaxed concurrency control in transactional memory. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-43)*, 2010.

[12] P. Bailis, A. Fekete, M. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *VLDB*, 8(3), 2014.

[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. of the 17th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-17)*, 2008.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.

[15] C. Blundell, A. Raghavan, and M. M. Martin. RETCON: transactional repair without replay. In *Proc. of the 37th annual Intl. Symp. on Computer Architecture (ISCA-37)*, 2010.

[16] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *Proc. of the 34th annual Intl. Symp. on Computer Architecture (ISCA-34)*, 2007.

[17] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly, 2008.

[18] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *PODC*. 2013.

[19] I. Calciu, J. Gottschlich, and M. Herlihy. Using elimination and delegation to implement a scalable NUMA-friendly stack. In *HotPar*, 2013.

[20] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5), 2008.

[21] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of the 33rd annual Intl. Symp. on Computer Architecture (ISCA-33)*, 2006.

[22] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Proc. of the 13th IEEE intl. symp. on High Performance Computer Architecture (HPCA-13)*, 2007.

[23] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proc. of the 4th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, 1991.

[24] D. Chen, N. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, J. J. Parker, et al. The IBM Blue Gene/Q interconnection network and message unit. In *SC*, 2011.

[25] S. Chen, P. B. Gibbons, M. Kozuch, and T. C. Mowry. Log-based architectures: using multicore to help software behave correctly. *ACM SIGOPS Operating Systems Review*, 45(1), 2011.

[26] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *Proc. of the 26th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*, 2012.

[27] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proc. of the 20th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-20)*, 2011.

[28] A. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multi-threaded applications. In *EuroSys*, 2013.

[29] 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 *Proc. of the 24th Symp. on Operating System Principles (SOSP-24)*, 2013.

[30] C. Click. AzulâĂŹs experiences with hardware transactional memory. In *Transactional Memory Workshop*, 2009.

[31] F. J. Corbato. A Paging Experiment with the Multics System. In *MIT Project MAC Report MAC-M-384*, 1968.

[32] D. Culler, J. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann, 1999.

[33] W. Dally. GPU computing: To exascale and beyond. Invited talk. *Supercomputing, New Orleans*, 2010.

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

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

[36] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th USENIX symp. on Operating Systems Design and Implementation (OSDI-6)*, 2004.

[37] J. Demmel and H. D. Nguyen. Fast reproducible floating-point summation. In *ARITH*, 2013.

[38] D. Dill, A. Drexler, A. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proc. of the 10th Intl. Conf. on Computer Design (ICCD)*, 1992.

[39] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *IWOMP-4*, 2008.

[40] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable nonzero indicators. In *PODC*, 2007.

[41] S. Franey and M. Lipasti. Accelerating atomic operations on GPGPUs. In *NOCS-7*, 2013.

[42] M. Frigo, P. Halpern, C. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proc. of the 21st ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

[43] A. Gottlieb, R. Grishman, C. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultra-computer: Designing a MIMD Shared Memory Parallel Computer. *IEEE Trans. Comput.*, 100(2), 1983.

[44] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st annual Intl. Symp. on Computer Architecture (ISCA-31)*, 2004.

[45] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[46] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture (ISCA-20)*, 1993.

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

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

[49] H. Hoffmann, D. Wentzlaff, and A. Agarwal. Remote store programming. In *Proc. of the 5th intl. conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2010.

[50] S. A. R. Jafri, G. Voskuilen, and T. Vijaykumar. Wait-n-GoTM: improving HTM performance by serializing cyclic dependencies. In *Proc. of the 18th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*, 2013.

[51] N. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August. Speculative separation for privatization and reductions. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2012.

[52] W. Jung, J. Park, and J. Lee. Versatile and scalable parallel histogram construction. In *Proc. of the 23rd Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-23)*, 2014.

[53] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. In *Proc. of the 44th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-44)*, 2011.

[54] J. Kelm, D. Johnson, M. Johnson, N. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *Proc. of the 36th annual Intl. Symp. on Computer Architecture (ISCA-36)*, 2009.

[55] R. Kessler and J. Schwarzmeier. CRAY T3D: A new dimension for Cray Research. In *COMPCON*, 1993.

[56] F. Kjolstad and M. Snir. Ghost cell pattern. In *Workshop on Parallel Programming Patterns*, 2010.

[57] K. Knowlton. A fast storage allocator. *CACM*, (8), 1965.

[58] C. Koelbel. *HPF handbook*. MIT Press, 1994.

[59] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2011.

[60] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2007.

[61] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy. In *Proc. of the 45th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-45)*, 2012.

[62] G. Kurian. *Locality-aware Cache Hierarchy Management for Multicore Processors*. PhD thesis, MIT, 2014.

[63] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proc. of the 10th USENIX symp. on Operating Systems Design and Implementation (OSDI-10)*, 2012.

[64] J. Larus, B. Richards, and G. Viswanathan. LCM: Memory system support for parallel language implementation. In *Proc. of the 6th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994.

[65] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proc. of the 24th annual Intl. Symp. on Computer Architecture (ISCA-24)*, 1997.

[66] A. Lebeck and D. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proc. of the 22nd annual Intl. Symp. on Computer Architecture (ISCA-22)*, 1995.

[67] C. Leiserson and T. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[68] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson. SI-TM: reducing transactional memory abort rates through snapshot isolation. In *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014.

[69] K. Mackenzie, J. Kubiatowicz, M. Frank, W.-J. Lee, W. Lee, A. Agarwal, and M. F. Kaashoek. Exploiting two-case delivery for fast protected messaging. In *Proc. of the 4th IEEE intl. symp. on High Performance Computer Architecture (HPCA-4)*, 1998.

[70] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE TPDS*, 15(6), 2004.

[71] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of the IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2008.

[72] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, D. A. Wood, et al. LogTM: log-based transactional memory. In *Proc. of the 12th IEEE intl. symp. on High Performance Computer Architecture (HPCA-12)*, 2006.

[73] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proc. of the 12th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.

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

[75] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 network architecture. In *Hot Interconnects 9, 2001.*, 2001.

[76] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (ISCA-42)*, 2015.

[77] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proc. of the 11th USENIX symp. on Operating Systems Design and Implementation (OSDI-11)*, 2014.

[78] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[79] M. Papamarcos and J. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proc. of the 11th annual Intl. Symp. on Computer Architecture (ISCA-11)*, 1984.

[80] L.-S. Peh and W. J. Dally. A delay model and speculative architecture for pipelined routers. In *Proc. of the 7th IEEE intl. symp. on High Performance Computer Architecture (HPCA-7)*, 2001.

[81] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2011.

[82] X. Qian, W. Ahn, and J. Torrellas. Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-43)*, 2010.

[83] X. Qian, B. Sahelices, and J. Torrellas. OmniOrder: Directory-based conflict serialization of transactions. In *Proc. of the 41st annual Intl. Symp. on Computer Architecture (ISCA-41)*, 2014.

[84] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.

[85] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. of the 32nd annual Intl. Symp. on Computer Architecture (ISCA-32)*, 2005.

[86] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *Proc. of the 41st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-41)*, 2008.

[87] L. Rauchwerger and D. Padua. The privatizing doall test: A run-time technique for doall loop identification and array privatization. In *Proc. of the Intl. Conf. on Supercomputing (ICS'94)*, 1994.

[88] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE TPDS*, 10(2), 1999.

[89] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[90] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. of the 21st annual Intl. Symp. on Computer Architecture (ISCA-21)*, 1994.

[91] R. F. Resende, D. Agrawal, and A. El Abbadi. Semantic locking in object-oriented database systems. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1994.

[92] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 1996.

[93] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing legacy code: An experience report using GCC and memcached. In *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014.

[94] D. Sanchez and C. Kozyrakis. SCD: A scalable coherence directory with flexible sharer set encoding. In *Proc. of the 18th IEEE intl. symp. on High Performance Computer Architecture (HPCA-18)*, 2012.

[95] D. Sanchez and C. Kozyrakis. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proc. of the 40th annual Intl. Symp. on Computer Architecture (ISCA-40)*, 2013.

[96] D. Sanchez, R. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proc. of the 15th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV)*, 2010.

[97] S. R. Sarangi, W. L. Torrellas, Y. Zhou, et al. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proc. of the 38th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-38)*, 2005.

[98] N. Satish, N. Sundaram, M. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD*, 2014.

[99] S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proc. of the 7th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.

[100] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7), 2010.

[101] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2010.

[102] 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 *Proc. of the 31st Intl. Conf. on Computer Design (ICCD)*, 2013.

[103] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *Proc. of the 25th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[104] T. Skare and C. Kozyrakis. Early release: Friend or foe. In *Workshop on Transactional Memory Workloads*, 2006.

[105] D. Sorin, M. Hill, and D. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3), 2011.

[106] O. Villa, D. Chavarría-Miranda, V. Gurumoorthi, A. Márquez, and S. Krishnamoorthy. Effects of floating-point non-associativity on numerical computations on massively multithreaded systems. *Cray User Group*, 2009.

[107] T. Von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th annual Intl. Symp. on Computer Architecture (ISCA-19)*, 1992.

[108] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th annual Intl. Symp. on Computer Architecture (ISCA-19)*, 1992.

[109] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-21)*, 2012.

[110] J. Warnock, B. Curran, J. Badar, G. Fredeman, D. Plass, Y. Chan, S. Carey, G. Salem, F. Schroeder, F. Malgioglio, et al. 22nm next-generation IBM System z microprocessor. In *Proc. of the IEEE Intl. Solid-State Circuits Conf. (ISSCC)*, 2015.

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

[112] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU architecture. *IEEE Micro*, 31(2), 2011.

[113] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang. Pangaea: a tightly-coupled IA32 heterogeneous chip multiprocessor. In *Proc. of the 17th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-17)*, 2008.

[114] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *Proc. of the ACM/IEEE conf. on Supercomputing (SC13)*, 2013.

[115] H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *Proc. of the Intl. Conf. on Supercomputing (ICS'00)*, 2000.

[116] J. Zebchuk, M. Qureshi, V. Srinivasan, and A. Moshovos. A tagless coherence directory. In *Proc. of the 42nd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-42)*, 2009.

[117] J. Zebchuk, E. Safi, and A. Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *Proc. of the 40th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-40)*, 2007.

[118] L. Zhang, Z. Fang, and J. Carter. Highly efficient synchronization based on active memory operations. In *Proc. of the 18th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*, 2004.

[119] M. Zhang, J. Bingham, J. Erickson, and D. Sorin. PVCoherence: Designing flat coherence protocols for scalable verification. In *Proc. of the 20th IEEE intl. symp. on High Performance Computer Architecture (HPCA-20)*, 2014.

[120] M. Zhang, A. Lebeck, and D. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-43)*, 2010.

[121] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas. Protozoa: Adaptive granularity cache coherence. In *Proc. of the 40th annual Intl. Symp. on Computer Architecture (ISCA-40)*, 2013.