Anatomy of a Scalable Software Transactional Memory

Yossi Lev
Brown University and
Sun Microsystems Laboratories
yosef.lev@sun.com

Victor Luchangco
Virendra J. Marathe
Mark Moir Dan Nussbaum
Sun Microsystems Laboratories
{victor.luchangco,virendra.marathe,mark.moir,dan.nussbaum}@sun.com

Marek Olszewski
Massachusetts Institute of Technology
and Sun Microsystems Laboratories
mareko@csail.mit.edu

Abstract

Existing software transactional memory (STM) implementations often exhibit poor scalability, usually because of nonscalable mechanisms for read sharing, transactional consistency, and privatization; some STMs also have nonscalable centralized commit mechanisms. We describe novel techniques to eliminate bottlenecks from all of these mechanisms, and present SkySTM, which employs these techniques. SkySTM is the first STM that supports privatization and scales on modern multicore multiprocessors with hundreds of hardware threads on multiple chips.

A central theme in this work is avoiding frequent updates to centralized metadata, especially for multi-chip systems, in which the cost of accessing centralized metadata increases dramatically. A key mechanism we use to do so is a scalable nonzero indicator (SNZI), which was designed for this purpose. A secondary contribution of the paper is a new and simplified SNZI algorithm.

Our scalable privatization mechanism imposes only about 4% overhead in low-contention experiments; when contention is higher, the overhead still reaches only 35% with over 250 threads. In contrast, prior approaches have been reported as imposing over 100% overhead in some cases, even with only 8 threads.

1. Introduction

The advent of multicore chips has made multiprocessors ubiquitous, and the hardware threads per chip and multicore chips per multiprocessor system continue to increase. Scalable applications—that is, applications that can effectively exploit more hardware threads as they become available—are difficult to construct due to tradeoffs between performance, scalability, and complexity. Traditional lock-based programming is too difficult and error-prone to support widespread development of scalable applications.

Transactional memory [13] has gained momentum as an alternative to lock-based programming. With transactional memory, rather than using locks to ensure that critical sections of code appear atomic, a programmer simply indicates that such a section should be executed as a transaction; the system is responsible for guaranteeing that the transaction appears to execute atomically.

Many proposals exist for implementing transactional memory in hardware [13, 2, 22, 31], software [26, 7, 12, 18, 21], and a combination of the two [5, 17, 4, 25, 15, 3]; realizing and improving these implementations remains an active area of research. Unless hardware support for transactional memory becomes as ubiquitous as multiprocessors, and until such a time, software transactional memory (STM) is necessary to support transactional programming.

Recent advances in STM performance [7, 29, 24] have been achieved mostly by using centralized synchronization mechanisms. However, these mechanisms generally do not scale well to large numbers of threads, especially in multi-chip shared memory systems, in which frequent modifications of centralized data structures leads to high latency and excessive memory coherence traffic. These problems are exacerbated when we consider STMs that support *privatization* [8, 14, 27, 20], which is widely considered to be important for providing an intuitive programming model.

In this paper, we present *SkySTM*,[1] the first STM that supports privatization and can scale to systems with hundreds of hardware threads across multiple multicore chips. We are releasing an open-source library containing SkySTM and other STMs discussed in this paper (see [1]).

SkySTM is designed to work in a *hybrid transactional memory* [5] system, allowing transactions to be executed using hardware transactional memory support if available and effective, otherwise running transactions in software. Hybrid transactional memory requires hardware and software transactions to interoperate correctly. SkySTM is designed to minimize overhead for hardware transactions and to avoid aborting them unnecessarily.

Our contributions include detailed descriptions of novel scalable mechanisms for efficiently ensuring transactional consistency and for supporting read sharing and a privatization guarantee, along with performance experiments demonstrating their effectiveness in achieving a scalable STM. Below we provide some background to provide context for these mechanisms.

1.1 Scalability of STMs

Early STM systems either imposed substantial overhead in order to guarantee that user code always executes in a "consistent" state, or allowed inconsistent states to be observed, which can lead to arbitrary behavior in unmanaged languages such as C and C++. Recent breakthroughs in STM performance can be attributed largely to clever mechanisms for guaranteeing consistent transactional execution with much lower overhead than previous approaches. For example, the TL2 STM [7] uses a timestamp mechanism to allow the system to verify in constant time that each new transactional read is consistent with the previous ones.

TL2's timestamp mechanism—and other related techniques for reducing the overhead of maintaining consistency—uses centralized metadata that is modified by every transaction in the worst case. TL2's careful integration of a central timestamp mechanism yields excellent performance compared to other STMs, and reasonable scalability up to a point. However, as we show, in systems with multiple multicore chips, the cost of frequently modifying the centralized timestamp increases dramatically, and the algorithm stops scaling. Other algorithms that frequently update centralized data structures are likely to suffer similarly.

[1] The name SkySTM is a play on the words "Scalable Hybrid STM".

Variants on the basic TL2 algorithm can reduce the frequency of updates to the global counter, thus significantly improving scalability. However, neither the basic form of TL2 nor any of these variants supports privatization. Marathe et al. [20] describe a variant of TL2 due to Detlefs et al. that supports privatization. We have also implemented this variant and, as we show, its scalability on multi-chip multicore systems is even worse than the basic form of TL2.[2] Moreover, as far as we know, the modifications required to make TL2 support privatization are incompatible with the variants to relieve pressure on the global counter. Thus, to date there is no scalable STM that supports privatization.

Supporting privatization is not the only challenge in achieving a scalable STM. Below we explain our general approach to achieving scalability, and discuss how we have applied this approach to eliminate or reduce scalability bottlenecks in several aspects of the STM, finishing with how we provide scalable privatization.

If a transactional application is not scalable due to contention on *application* data, the best STM cannot make it so. Thus, our goal is to avoid using mechanisms in the STM that *introduce* synchronization bottlenecks even when the application is scalable. A primary design goal for achieving this is to avoid contention on the STM's metadata when contention on application data is low. This suggests a *conflict-based* approach to synchronization, in which contention on STM metadata is induced only (or at least primarily) when there is contention on application data.[3] This approach has driven the development of the scalable STM mechanisms we present.

The first use of this approach concerns maintaining consistency during transactional execution. Early STM systems designed by us and others used either *invisible* reads, in which each transaction maintains per-read metadata to be revalidated after each subsequent read; or *visible* reads, in which each reader "registers" for each memory location it reads, so that a transaction wishing to write it can identify and abort such readers. Invisible read schemes imposed substantial overhead on transactions due to the need for repeated validation, and in hybrid TM contexts impose additional overhead on hardware transactions to update metadata to allow software readers to detect conflicts. Visible read schemes were complex, expensive, and nonscalable.

These tradeoffs led us to develop *semivisible* reads (originally semi-transparent reads [16]): a transaction that wishes to write to a memory location can determine whether *any* other transaction is reading the location, but not *which* transaction(s). This enables us to determine when writers conflict with readers and to maintain a counter that is incremented whenever such a conflict occurs. With care, we can arrange for transactions to avoid expensive validation in the vast majority of cases by checking whether *any* writer-reader conflict has occurred during its execution. The counter is updated only when there are writer-reader conflicts, consistent with the conflict-driven synchronization approach described above.

An obvious way to implement semivisible reads is to maintain a counter of readers for each memory location. However this simple approach introduces metadata contention between transactions that do not conflict (read sharing of application data causes contended write sharing of metadata). This problem was one of the primary motivations behind Scalable NonZero Indicator (SNZI, pronounced "snazzy") algorithms [9]. We exploit the fact that writers need to know only if there are *any* conflicting readers, not how many there are, to replace the nonscalable counter with a SNZI. The integration of SNZI into SkySTM demonstrates that these objects are not merely of theoretical interest, but are also useful in practice. A secondary contribution of our paper is to present a new and simpler SNZI algorithm that we discovered recently.

[2] Dice experimented with essentially the same algorithm in early 2007 [6], but did not pursue it due to its poor performance.

[3] As explained in Section 2.1, SkySTM hashes memory locations to metadata with the result that nonconflicting memory accesses in the application can sometimes result in "false" conflicts, which can sometimes impede scalability even when the application is scalable. In the remainder of the paper, we include false conflicts when we refer to application conflicts; reducing or eliminating false conflicts is future work.

1.2 Supporting privatization

Making an STM scalable is made more difficult by the so-called *privatization problem* [8, 14, 27, 20]: A thread may use a transaction to "isolate" a shared chunk of memory so that no other thread should be able to access it (e.g., by setting all shared references to that memory chunk to `NULL`), and thereafter (nontransactionally) operate on that memory as though it were private. Allowing nontransactional access to private memory is crucial for both performance and interoperability reasons (see Section 5). However, with many prior STM implementations, an isolated buffer may still be written by another thread *after* the thread isolating the buffer finishes committing. For example, in a deferred-writes STM (i.e., one in which values written by a transaction are kept in a write set and copied back to the memory when the transaction commits, as opposed to one which writes immediately in place and undoes those writes if the transaction aborts), the other thread may have committed before the buffer was isolated but still be copying its write set back to memory. A similar problem occurs with writes by aborted transactions with STMs that perform in-place writes. These writes may interfere with the thread that thinks the memory is private.

In many multithreaded applications, the "lifecycle" of a chunk of memory is complex. For example, it may be used in a shared data structure, then removed from that data structure, then passed to a non-thread-safe legacy library for processing, then placed back into a different shared data structure, and at some point it may be deallocated. Requiring programmers to explicitly annotate code that "privatizes" a chunk of memory before it is accessed nontransactionally or deallocated is undesirable: doing so will be error-prone, especially when incrementally adopting transactional programming into a large lock-based application, and the resulting bugs will manifest nondeterministically and be difficult to reproduce and diagnose.

Previous mechanisms for *implicit* privatization, which address this problem without requiring explicit annotation, rely on expensive centralized mechanisms that severely impact scalability. For example, experiments presented by Yoo et al. [32] show that allowing programmers to *disable* their centralized privatization mechanism can result in more than a 100% performance improvement with just 8 threads in some cases. Allowing programmers to disable privatization where it is *not* needed is undoubtedly better than requiring them to provide annotations where it *is* needed. However, with such severe overhead, programmers will be motivated to disable privatization aggressively. Apart from the additional burden this places on the programmer, it is inevitable that programmers will incorrectly disable privatization, or perhaps subsequent code changes will render a previously correct usage incorrect, again resulting in subtle nondeterministic bugs. We are therefore motivated to find the best implicit privatization mechanisms possible to avoid the need for explicit programmer annotations.

We present a novel privatization mechanism in which a privatizing transaction waits only for *conflicting* transactions, in contrast to the aforementioned mechanisms, which require transactions to wait even for nonconflicting transactions. Thus, if the application has few conflicts, the privatization mechanism does not impede its scalability. Our scalable mechanisms for supporting transactional consistency and read sharing facilitate our privatization mechanism.

Marathe et al. [20] recently introduced a new privatization mechanism that is similarly motivated. In particular, they too observe that a privatizing transaction need not wait for others with which it does not conflict. However, their solution requires a centralized

list of all active transactions, and their experiments confirm our intuition that this list becomes a bottleneck and impedes scalability, even on a single-chip multicore system.

In Section 2, we review the HyTM STM [5], from which SkySTM evolved, and also some other relevant STMs. In Section 3, we describe the mechanism we use to validate reads quickly without introducing too much contention, and show how it improves scalability over the centralized timestamp mechanism used in TL2 and other STMs. Section 4 describes our new SNZI algorithm and shows how SNZI enables scalable read sharing. Section 5 discusses our privatization guarantee, describes a scalable algorithm that provides this guarantee, and demonstrates that it incurs modest overhead and does not impede scalability. We conclude in Section 6.

2. Background

2.1 The HyTM STM

The HyTM STM [5] uses a deferred writes approach, recording transactional writes in a private write set until commit time, when they are copied back to memory. A transaction can own a location either for reading or for writing, where write ownership is exclusive, and read ownership can be shared. Each location is associated with an *ownership record* (*orec*), which keeps the ownership information for the locations[4] with which it is associated.

Both the HyTM STM library and the SkySTM library support various modes, for example invisible vs. semivisible reads, and eager vs. lazy ownership acquisition of locations to be written. In this paper, we focus on semivisible reads and eager write acquisition, and comment on differences for other modes where appropriate.

Write ownership is obtained by storing the id of the (unique) owner in the appropriate orec. This ownership can be revoked only after the current owner is aborted. For read ownership, the HyTM STM uses semivisible reads, whereby a counter in the orec indicates how many (but not which) transactions own it for reading. In contrast to write ownership, when read ownership is revoked, the owners are not notified and aborted; it is the readers' responsibility to validate their read ownerships and abort themselves if any of those ownerships are revoked. To do that, when a transaction acquires an orec for reading (by incrementing the orec's read counter), it keeps a snapshot indicating the id of the last transaction that write-owned the orec. These snapshots are kept in a private read set, and are used to *validate* the transaction, i.e., check whether any of its read ownerships have been revoked. A completed transaction, whether committed or aborted, relinquishes all its ownerships.

Using semivisible reads facilitates efficient interaction with hardware transactions: When first accessing a location, a hardware transaction checks the associated orec to make sure that the access does not conflict with a software transaction. If the hardware transaction writes the location, it ensures that the read count is 0; if a software transaction later changes the counter to a nonzero value, the hardware transaction is aborted. Thus, the hardware transaction need not change any metadata used by the STM, which is important because we want to minimize the overhead for hardware transactions. As we later show, semivisible reads also enables a scalable mechanism for fast read validation.

Finally, a transaction (after validating its read set for the last time) commits by atomically changing its status to "Committed". This locks all the orecs it owns for writing, making the write ownership irrevocable until it is released (which is after the transaction finishes copying the values from its write set to the memory locations it wrote). Once a transaction commits, it becomes blocking (i.e., all concurrent transactions attempting to access the locations that are write owned by the committed transaction must wait for it to release ownership of those locations). A transaction never blocks other transactions while it is executing user code.

[4] Several locations are associated with each orec, introducing false conflicts but not affecting correctness.

2.2 TL2 and its variants

The need to validate reads makes it difficult to make transactional memory systems efficient and scalable. Whenever a transaction reads some memory location, it must ensure that the value it gets back is consistent with the values it has previously read. This is particularly important in unmanaged languages like `C++`, because using inconsistent reads can lead to arbitrary program behavior. A naïve validation algorithm that iterates over the entire read set on every validation incurs overhead that is quadratic in the number of reads over the whole transaction. This overhead dominates the cost of the transaction as its size increases.

Like SkySTM, TL2 [7] associates an orec with each memory location. TL2 uses a global shared counter to generate timestamps for committed transactions. Transactions are serialized in the order of their (unique) timestamps. When a transaction commits, it stores its timestamp in the orec of each location that it writes. Thus, a transaction can quickly check that a location has not changed since it began by recording the global counter when it begins, and comparing that value to the timestamp in the orec.

The open-source TL2 library [30] contains several variants on the basic algorithm described in [7], which differ in how they manage the global counter. As originally described, TL2 increments the counter every time a transaction that writes a location commits, and uses the resulting value as its timestamp. Thus, every such transaction has a unique timestamp. This induces contention on the global counter, even when transactions do not conflict with each other. In its default mode, called GV4, TL2 mitigates this problem by using a *pass-on-failure* strategy: if a transaction fails to increment the counter, it does not retry the increment, but instead uses the new value of the counter as its timestamp. This is safe because the counter is always incremented after the transaction has locked all the orecs associated with its writes, and validated its reads; hence transactions that contend on incrementing the counter do not conflict with each other, and can therefore commit "at the same time", using the same timestamp. This optimization significantly reduces the cost of the contention as long as the cost of shared memory access is low, because each writer-transaction tries to increment the counter exactly once, whether it succeeds or fails. However, as we later show, as the cost of shared memory access increases, this mechanism's scalability breaks down.

An alternate scheme, called GV5, further reduces contention for the global counter by not requiring every transaction even *attempt* to increment it. Instead, when a transaction commits, it reads the global counter and increments it locally, using the resulting value as its timestamp but not writing the incremented value back to the global counter. Thus, unlike in GV4, orecs may have timestamps that are greater than the value in the global counter. If a transaction accesses such an orec, then it aborts and ensures that the global counter's value is at least as large as the timestamp in the orec before retrying. Although GV5 reduces contention on the global counter, it introduces unnecessary aborts. In particular, if a single thread repeatedly accesses the same location in successive transactions, every second transaction will fail even if there are no other threads running.

GV6, the variant that seems to perform the best, combines the two schemes described above: A transaction uses the GV4 scheme with probability 1/32, and the GV5 scheme the rest of the time. This approach aims to avoid unnecessary aborts by advancing the counter, but avoids a bottleneck on the counter by incrementing it less frequently than GV4.

2.3 Other recent STMs with centralized mechanisms

RSTM [19] uses a global *committing counter* that is incremented every time a writer transaction commits, and transactions do full read set validation only when the global counter changes [28].

RingSTM [29] reduces the cost of validation by keeping an ordered ring with a record for each transaction that has committed, but may not have finished copying the values from its write set to shared memory. Again, the ring is a centralized structure that is modified by every transaction, and thus is likely to prevent RingSTM from scaling up to systems with many threads.

Other timestamp-based STMs have focused on reducing contention on the centralized timestamp using hardware support for distributed synchronized hardware clocks [23] or techniques to reduce transaction aborts due to false conflicts [10]. These STMs have only been demonstrated to scale to 16 or fewer threads, and they do not guarantee correct privatization.

2.4 Adding privatization to TL2

As mentioned above, TL2 does not provide privatization. The *Ord* algorithm due to Detlefs et al. (described in [20]) does provide privatization in a TL2-like STM.

Marathe et al. [27, 20] identify two kinds of undesirable behavior exhibited by many early STMs with respect to privatization: "delayed cleanup" and "doomed transactions". *Delayed cleanup* occurs when a transaction that is finishing its cleanup (either copying back values to memory during commit in a deferred-writes STM, or rolling back changes upon abort in an in-place STM) interferes with a thread that has already privatized the affected memory. The *doomed transaction* problem occurs when a transaction reads from memory that has already been privatized and is being updated non-transactionally; the STM fails to detect the conflict with the non-transactional accesses, and therefore delivers inconsistent data to the application, which can result in arbitrary incorrect behavior.

The delayed cleanup problem can manifest in TL2 if the serialization and completion (returning from the commit operation) order of transactions are inverted. To avoid this inversion, Ord uses a ticket-lock like algorithm to enforce consistent serialization and completion orders. Essentially, each transaction grabs a ticket at the beginning of its commit operation (after acquiring write ownership of locations in its write set), completes its validation and cleanup, and then waits for its "turn" to return from the commit operation.

The doomed transaction problem can also arise in TL2 because when a transaction reads a location, it checks only that this location has not been written transactionally since the transaction began; it does not ensure that other locations previously read still contain the values read. Thus, the transaction may read a buffer that another transaction privatized by writing such a location, and not detect the conflict until it attempts to commit. If the transaction is a read-only transaction, it might even successfully commit with the inconsistent view of the memory. To avoid the problem, Ord requires every transaction to revalidate its entire read set every time any transaction commits. Information about new commits is relayed via the global counter in TL2. Since the counter in GV5 (and hence in GV6) does not reliably indicate whether a transaction committed recently, Ord cannot exploit the benefits of these algorithms.

3. Scalable and Efficient Transactional Reads

3.1 Fast read validation

SkySTM uses a new variant of the `RWConflictCounter` algorithm [16, 5] to provide fast read validation. This algorithm exploits the fact that we have semivisible reads by incrementing a global counter `RWConflictCounter` only when a writing transaction *that has conflicted with a reader* commits, rather than every time *any* writing transaction commits. We need not perform a full read-set validation if `RWConflictCounter` has not changed since the last time the read set was known to be valid. Instead, a transaction reads `RWConflictCounter` when it begins, and uses the value read for fast read validation after every read operation. If the transaction detects that `RWConflictCounter` has changed, it does a full read validation, replacing its local version with the new `RWConflictCounter` value if the full validation succeeds. This is safe because the read set is known to be valid as of a point in time *after* the new value was obtained.

To facilitate incrementing `RWConflictCounter` appropriately, we add a one-bit field `rwc` to each orec. This bit is set whenever a transaction acquires write ownership of an orec that was owned by at least one reader (i.e., the transaction "steals" the orec from the readers). If the transaction later commits, the `rwc` bit tells it that it must increment `RWConflictCounter`, as explained below. This information is stored in the orec—and not just recorded privately by the stealing transaction—so that if the orec is stolen again, the need to increment `RWConflictCounter` is not lost.

At commit time, if a transaction owns for writing an orec that has its `rwc` bit set, it increments `RWConflictCounter`. This increment must be performed at or before the final read validation of a successful transaction (which precedes any copying of values from the write set back to shared memory). We first try to validate the read set and increment `RWConflictCounter` together by calling `FetchAndInc` on `RWConflictCounter` and comparing the value returned to the local version mentioned above. If `RWConflictCounter` has changed, we run a full read validation, knowing that if it succeeds, `RWConflictCounter` has already been incremented. A transaction that increments `RWConflictCounter` clears the `rwc` bit of an orec it owns for writing when it releases the write ownership of that orecs.

We can improve the scalability of this approach by using a *pass-on-failure* strategy like the one used in TL2: a thread fails to increment `RWConflictCounter` only if some other thread did so successfully at the same time, and our strategy is correct as long as `RWConflictCounter` has been incremented at least once since the transaction last stole any orec and before it commits. Therefore, the contention on the central `RWConflictCounter` will never exceed the contention on the TL2 central counter, and will match it only if all writing transactions conflict with other transactions, in which case the application is unlikely to scale well regardless of the STM implementation.

Fast read validation with `RWConflictCounter` also guarantees that all read values are consistent at some point *during* the validation operation, in contrast to TL2's mechanism, which only guarantees that they were current at the beginning of the transaction. This is why SkySTM is immune to the doomed transaction problem while TL2 is not (see Sections 2.4 and 5.5).

Integration with Hardware Transactions: As described so far, aborted transactions and transactions that do not own for writing any orec whose `rwc` bit is set neither update `RWConflictCounter` nor clear the `rwc` bit of orecs whose ownership they release. However, in a system that allows software and hardware transactions to coexist, an unowned orec whose `rwc` bit is set is problematic: a hardware transaction that writes to a location mapping to such an orec must either increment `RWConflictCounter` or abort itself. Both these options are unacceptable: the former would add significant overhead to hardware execution and the latter would reduce the number of transactions that can succeed in hardware.

We instead address the issue in two ways:

1. An aborted transaction behaves like a committing transaction in that it updates `RWConflictCounter` if any member of its write set has its `rwc` bit set and clears the `rwc` bit of any orec for which it successfully releases write ownership.

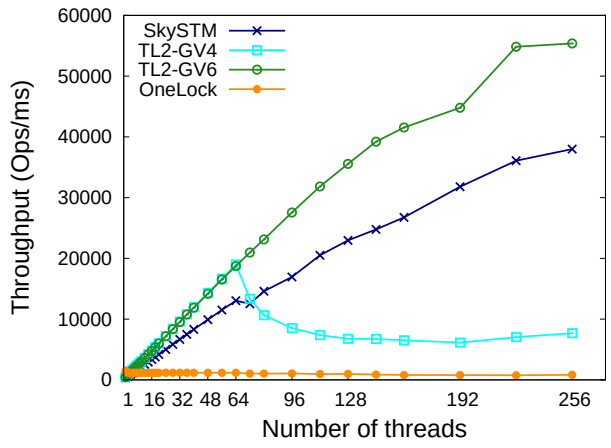

Figure 1. HashTable with *keyrange* = 128,000, 50% insertions, 50% deletions.

2. The last transaction to release *read* ownership of an orec whose `rwc` bit is set also clears the bit after incrementing `RWConflictCounter`.

These measures cause `RWConflictCounter` to change more often, inducing more full validations, than necessary for correctness. But that slight additional cost increases the number of transactions that can run in hardware without introducing unnecessary overhead for hardware execution.

3.2 Evaluation

All experiments presented in this paper were run on a 1.4 GHz Sun SPARC Enterprise® T5440 server [11], which combines chip-level multithreading (*CMT*) and symmetric multiprocessing (*SMP*) design aspects. The T5440 contains four UltraSPARC® T2 Plus processors connected via four UltraSPARC T2 Plus XBR coherency hubs. Each T2 Plus processor contains a single 4 MB L2 cache shared by eight cores, each of which supports 8 hardware threads, for a total of 64 hardware threads per chip. Thus, while the machine supports up to 256 hardware threads, inter-thread communication overhead increases significantly when running more than 64 threads, as then not all threads share the same L2 cache.

In this section, we show that a centralized synchronization mechanism, such as the timestamp variable used by the TL2 STM, may severely impede scalability, even in the absence of application contention. We use a simple HashTable benchmark, which stores keys in 131,072 (2^{17}) buckets, with chaining. Insert and delete operations do not modify the table if the target key is found or not found respectively. Each thread executes insert and remove operations with equal probability (50%) with keys chosen uniformly at random from the range of 0 to 127,999. The table is initialized with 64,000 random keys in this range, so that the probability for an insert or remove operation to succeed remains roughly the same (50%) throughout the run (the table always contains approximately half of the keys in the range). The benchmark provides a high degree of parallelism, and thus should scale well.

Figure 1 shows the throughput of the HashTable benchmark when running with: TL2-GV4, TL2-GV6, SkySTM using semivisible reads and the centralized `RWConflictCounter` mechanism, and a single lock. SkySTM scales almost linearly up to 256 threads, while TL2-GV4 scales only while all threads are on the same chip (i.e., up to 64 threads); beyond this point, throughput decreases. Thus, although SkySTM runs 1.39 times slower than TL2-GV4 on a single-thread run, its throughput is 4.94 times higher with 256 threads. On the other hand, the GV6 variant of TL2 outperforms all other solutions, as it reduces the contention on the centralized variable without the overhead of semivisible reads. Unfortunately, we are not aware of any way to provide privatization with the GV6 variant of TL2, or avoid the additional overhead for concurrent hardware transactions that is imposed by TL2's invisible reads.

Note that despite TL2's centralized timestamp variable, TL2-GV4 scales very well when its communication remains on-chip. The combination of a shared L2 cache and TL2's pass-on-failure strategy reduces contention on the timestamp sufficiently to allow good scalability. However, with more than 64 threads, the added cost of off-chip L2 cache misses, caused by the timestamp variable bouncing between the four chips, produces significant performance degradation.

When we modify the benchmark to increase contention, the performance of the TL2-GV4 variants doesn't change much, but SkySTM performance suffers considerable degradation, to the point where SkySTM's performance only beats that of TL2-GV4 by 13% at 256 threads. This degradation is caused at least in part by SkySTM's poor contention management. Addressing that issue is ongoing work.

4. Scalable Read-Sharing

In applications with high levels of read sharing, semivisible reads can introduce significant contention even when there is no conflict inherent to the application. For example, the root of a binary tree is read by every operation on the tree, but typically is rarely written. In this case, the root should not be a significant source of contention. However, with the naive implementation of semivisible reads described in Section 2.1, which uses a simple read counter in each orec, every operation modifies the read counter twice. Thus, the read counter is highly contended. For SkySTM to scale for such applications, we must reduce the contention on the read counter.

The key observation for reducing contention on the read counter is that the exact value of the counter is not important: it matters only whether the count is zero or nonzero. Therefore, two transactions that want to acquire read ownership of an orec need not contend if the orec is already owned for reading by some other transaction; the read counter is already nonzero and additional read acquisitions will not make it zero. Similarly, a transaction releasing read ownership will not make the read counter zero unless it is the only reader. This observation led to the development of a *scalable nonzero indicator* (SNZI) [9], which supports *arrive* and *depart* operations that respectively increment and decrement an abstract counter, and a *query* operation that determines whether the counter is nonzero.

To implement SNZI, we use a rooted tree of SNZI objects in which a child is implemented using its parent (i.e., an operation on a child may invoke operations on its parent). A reader may invoke an arrive operation at any node; it invokes the corresponding depart operation on the same node. An arrive operation on a node other than the root invokes an arrive operation on its parent if it may change the child's abstract counter from 0 to 1, and a depart operation on the child invokes a depart operation on the parent if it changes the child's abstract counter from 1 to 0. Query operations occur directly on the SNZI root node, whose abstract counter is nonzero if and only if the abstract counter of any node in the SNZI tree is nonzero. Thus, a child acts as a filter for its parent, and the tree structure greatly reduces contention for the root node. Although the root is frequently accessed, it is rarely written, so we can use a nonscalable algorithm (e.g., a simple counter) at the root without jeopardizing the scalability of the solution as a whole.

The basic SNZI object does not suffice for read indicators: a transaction wanting to write a location may "steal" the associated orec from transactions that own it for reading. In that case, the read counter is reset to 0. This motivated the development of the SNZI-R ("snazzier") variant [9], which supports a *reset* operation.

Although subsequent readers that arrive and depart (after the writing transaction commits or aborts) increment and decrement the abstract counter normally, readers whose ownership was revoked when the orec was stolen must *not* decrement the abstract counter: their contribution to the read counter was already removed by the reset operation. Often, a reader does not attempt to depart because it recognizes that its ownership has been revoked. However, the attempt cannot always be avoided: the reader might be just about to release ownership of the orec (i.e., depart) when the orec is stolen. Therefore, we must distinguish departs by readers whose ownership has been revoked from those that still own the orec for reading.

To achieve this, a SNZI-R object maintains an *epoch*, which is returned by the query operation (together with whether the abstract counter is nonzero). To arrive, a reader uses the query operation to determine the current epoch, and tries to arrive in that epoch. To depart, it uses the epoch to which it previously arrived. A reset operation replaces the current epoch with a new one (epochs must not be reused). The abstract counter is modified only by arrive and depart operations of the current epoch. Thus, if the current epoch is changed (by a reset operation) between the time a reader arrives and the time it departs, the reader's depart operation will have no effect on the abstract counter. In SkySTM, a reset operation coincides with a write acquisition of an orec, and the epoch is the id of the transaction that last acquired it for writing.

Since the publication of [9], we developed a new version of the hierarchical SNZI algorithm that is significantly simpler and slightly more efficient. Pseudocode for the new SNZI-R arrive and depart operations[5] is presented in Figure 2. The key difference between this new algorithm and the original one is that in the new algorithm, when the counter is zero, we *first* arrive at the parent and only then try to increment the child's counter from zero to one. Changing the order in which we arrive at the parent and modify the child's counter eliminates the need for the intermediate $\frac{1}{2}$ value and the version number used in the original SNZI algorithm [9], and significantly simplifies the correctness proof.

4.1 SNZI in SkySTM

SkySTM uses the new SNZI-R algorithm for implementing the read indicator in an orec. A small counter, denoted as the *SNZI counter*, is used in the orec to implement a root SNZI object; an arrive or depart operation on that object simply increments or decrements the orec's SNZI counter, conditional on the epoch of the arrive or depart operation being the current epoch (recall that the current epoch is the id of the transaction that most recently owned the orec for writing, which is also maintained in the orec).

The SNZI-R tree structure has three layers: the orec, a single child node of the orec and several "grandchild" leaf nodes. Each thread is assigned a leaf; it always arrives and departs at this leaf. Threads are scattered randomly among the leaves so that each leaf is assigned to at most eight threads. We are exploring various tree structures and assignment schemes to better adapt to different performance characteristics over a range of machine architectures.

It can be shown that because the orec's SNZI counter is incremented only by a single child node, and only when this node's abstract counter may change from 0 to 1, the orec's SNZI counter is at most 1 when no arrive operation is executing. Thus, we found that allocating a few bits in the orec for this counter is sufficient to avoid overflow; in the unusual case that the counter does get saturated, we use a back off and retry strategy. (In [9], we introduced a special algorithm for the root node that used only a single bit in the orec for the indicator, but we did not find it necessary in practice for SkySTM.)

[5] As in [9], for simplicity of presentation, the pseudocode shows an arrive operation that requires epochs to be totally ordered. In SkySTM, we use a slightly modified variant that works with unordered epochs.

```
X = (ctr, epoch): (ℕ, ℕ), initially (0, 0)
parent: SNZI-R

Arrive(e)
    pArr ← false
    repeat
        x ← Read(X)
        if x.epoch > e then return
        if x.ctr = 0 ∨ x.epoch < e then
            x' ← (1, e)
            if ¬pArr then
                parent.Arrive(e)
                pArr ← true
        else
            x' ← (x.ctr + 1, e)
    until CAS(X, x, x')
    if pArr ∧ x'.ctr ≠ 1 then
        parent.Depart(e)
    return

Depart(e)
    repeat
        x ← Read(X)
        if x.epoch ≠ e then return
    until CAS(X, x, (x.ctr − 1, e))
    if x.ctr = 1 then
        parent.Depart(e)
```

Figure 2. Code for new SNZI-R Arrive and Depart operations

Although the SNZI-R algorithm greatly improves the scalability of read indicators, it entails nontrivial overhead compared to the simple counter. Therefore, SkySTM uses the SuperSNZI-R strategy [9], which incorporates a small additional *read counter* field in the orec, to be used when the contention is low. To arrive at the read indicator of an orec, a transaction checks whether the orec's SNZI counter is zero, and if so, tries to arrive by simply incrementing the read counter. After several failed attempts to do so, the transaction resorts to arriving using the SNZI-R tree associated with the orec. Thus, the more costly SNZI-R algorithm is used only when the SNZI counter is already nonzero (indicating that the orec's read indicator is already contended), or when contention on the read counter is encountered. In addition to significantly improving performance, using SuperSNZI-R allows SkySTM to allocate SNZI-R trees only for contended orecs, thus significantly reducing the space overhead of the solution.

Note that with the arrive operation presented in Figure 2, all threads that read a node's counter as 0 arrive at the node's parent, and then all but one of them (the one that successfully changes the node's counter from 0 to 1) cancel their arrival at the parent by departing from it before returning from their arrive operation. To reduce contention on the parent due to such superfluous arrivals, SkySTM uses a simple extension to the SNZI algorithm in which a thread announces, by writing to an Announce flag using a regular write operation, that it is about to arrive at the parent; other threads that observe this flag as set delay their arrival at the parent, avoiding it completely if the node's counter is modified while they are delaying. While this simple extension does not *guarantee* that it will reduce the number of superfluous arrivals at the parent, it works well in practice, significantly reducing the number of arrive and depart operations invoked on the parent.

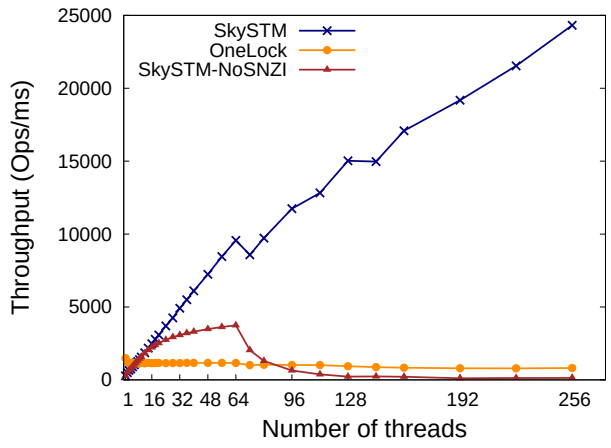

Figure 3. HashTable with a high level of read sharing: *keyrange* = 128,000, 50% insertions, 50% deletions.

4.2 Evaluation

In this section, we demonstrate how SNZI improves performance under heavy read sharing when using semivisible reads. To this end, we introduce a level of indirection to the HashTable benchmark: in this variant, the table of buckets is accessed via a shared pointer that has to be read (by all threads) at the beginning of each operation (such as might be used by a resizable hash table). We compare SkySTM with a version that uses simple counters for its read indicators (that is, with SNZI disabled). Figure 3 presents the results.

SkySTM scales reasonably well up to 256 threads despite heavy read sharing on the shared HashTable pointer. In contrast, the version of SkySTM with SNZI disabled starts scaling worse than SkySTM with as few as 16 threads, and when the number of threads exceeds 64, so that multiple chips are in use, throughput degrades with additional threads. By 96 threads, SkySTM with SNZI disabled performs worse than the single global lock.

While SNZI greatly improves the scalability, it does not come for free: with more than 8 threads, the throughput of the HashTable variant without the indirection is 36% higher than that of the variant with the indirection. The level of indirection introduces some additional cost, but in similar experiments with TL2, it caused only a 12% degradation, suggesting that SkySTM's degradation is primarily due to SNZI overhead, not the extra level of indirection.

5. Scalable Privatization

Like most STMs, SkySTM does not support *strong atomicity*, and therefore forbids concurrent accesses by nontransactional and transactional accesses to the same data. If this rule is violated, the atomicity guarantee of transactional memory may be compromised, for example by allowing a transaction to see inconsistent data or other threads to see internal states of a transaction.

However, it is desirable to allow nontransactional access to data that might have been *previously* accessed transactionally for at least two reasons. First, processing the data may require the use of library functions that have not been instrumented for use within transactions. It is not always possible to recompile such functions—the source code may not be available, for example. Second, the instrumentation necessary to run within transactions imposes significant overhead, so isolating data (using a transaction) and processing it nontransactionally can significantly improve overall performance.

As discussed in Section 1, we prefer to address the privatization problem *implicitly*, without requiring explicit annotations by the programmer. Previous implicit privatization mechanisms [24, 20] depend on centralized metadata that is modified frequently, and hence severely degrade scalability.

In Section 2.4 we mentioned two privatization related problems, one due to delayed cleanup, and one due to doomed transactions. As explained in Section 5.5, SkySTM (unlike TL2) is immune to the doomed transaction problem. Below we explain how to address the delayed cleanup problem in a scalable way. For clarity, our description focuses on the following *privatization guarantee*:

> If a transaction "isolates" a chunk of memory so that no other thread should be able to access it (e.g., by setting all shared references to that memory chunk to `NULL`), then once the isolating transaction completes its commit phase, no further writes to that memory chunk are executed by other threads.

This guarantee ensures that standard privatization idioms work as expected. In particular, a thread can "isolate" a memory buffer by using a transaction to remove it from the shared (transactional) space, and then subsequently access such privatized memory buffers safely without transactions. As discussed in Section 5.5, the privatization algorithms considered in this paper actually provide a stronger guarantee that enables *proxy privatization*, in which one thread privatizes a chunk of memory on behalf of another.

Unlike previous privatization mechanisms, our new scalable one is *conflict-based*: nonconflicting transactions do not wait for each other. It does not use any centralized metadata, and hence does not affect the scalability of conflict-free programs.

Although we describe our mechanism for avoiding the delayed cleanup problem in the context of SkySTM, which uses deferred writes, it can be applied to other STMs including TL2 and STMs that use in-place writes, and it works in a hybrid environment with hardware and software transactions running concurrently.

5.1 The basic solution

Our implicit privatization algorithm relies on the following simple observation:

> A transaction that isolates a buffer must conflict with any other transaction that concurrently accesses that buffer.

In particular, a privatizing transaction must write some shared data to indicate that the buffer is no longer shared—data that must be read by any transaction that may access the buffer. For example, a buffer that is accessible only through a single global pointer can be isolated by setting that pointer to `NULL`. In this case, the global pointer is the shared data involved in the conflict—it is written by the privatizing transaction and is read by all transactions accessing the buffer. Thus, a privatizing transaction need wait only for transactions that have read some location that it wrote.

With an STM that uses deferred writes, no values are copied back into shared memory by a writing transaction until after it has executed its final read validation. Thus, a conflicting transaction that has not yet begun its final read validation cannot interfere with a privatizing transaction that has already committed because it will fail that validation and abort. Thus, a privatizing transaction need wait only for conflicting transactions that have begun their final read validations but not yet finished copying their write sets back to memory.

To enable waiting for these transactions, we add a *committing indicator* (CI) field to each orec. This field indicates whether any transaction that earlier had read ownership of that orec has already executed its final read validation of that orec, but has not yet completed its copyback. Like the read indicator, the committing indicator can be implemented by a simple counter—such a counter would be incremented together with the final read validation of the orec, and decremented after the commit operation has copied back

all values from its write set to shared memory. In this case, the CI of an orec associated with a memory location read by a transaction is nonzero from the last time the transaction read-validates that orec until the transaction is done copying back the values from its write set to shared memory. Therefore, a privatizing transaction can provide the privatization guarantee by simply delaying the return from its commit operation until the CI of each orec it owns for writing is zero. Then any transaction that owned any of these orecs for reading has either finished its copyback phase, or will still read-validate that orec before committing (and the read validation will fail, causing the transaction to abort). In either case, once a transaction sees the CI of each orec in its write set as zero, no further writes to a buffer it has isolated can occur.

The following summarizes the steps taken by the new commit operation that provides implicit privatization:

1. (If in lazy acquire mode) acquire write ownership of all orecs in the write set in the usual manner (ignoring the CI fields).
2. For each orec in the read set that is not in the write set, atomically read-validate it, release read ownership (if necessary), and increment its CI. If validation fails, undo any prior increments of CIs, and abort (releasing the orecs owned for writing)
3. Change status from Active to Committed; if this fails, decrement CIs incremented in Step 2 and release each orec owned for writing.
4. Copy values from the write set back to shared memory.
5. Decrement the CIs incremented in Step 2.
6. For each orec owned for writing, wait until its CI is zero.
7. For each orec owned for writing, release write ownership.

The substeps of Steps 4 through 7 need not be executed in the order shown. However, some constraints are necessary: First, each substep of Step 4 must occur while the transaction has write ownership of the appropriate orec (i.e., between the corresponding substeps of Steps 1 and 7). Second, all substeps of Step 4 must occur before any substep of Step 5. Otherwise, the transaction might allow a privatizing transaction to complete before this transaction completes its copybacks. Third, each substep of Step 6 must occur after the corresponding orec has been acquired for writing in Step 1 (or earlier, in an eager-acquire STM). Otherwise, another transaction may increment the CI and proceed to its copyback phase after this transaction found the CI to be zero. Checking the CI after write ownership has been acquired prevents this because the other transaction will fail to read-validate the orec when it attempts to increment CI. Fourth, each substep of Step 6 must occur before the corresponding substep of Step 7 for the same orec. Otherwise, this transaction may be starved by a series of transactions that read the value after this transaction wrote it and keep the CI nonzero as they increment and decrement it. Finally, to achieve the proxy privatization property discussed in Section 5.5, Step 6 must *entirely* precede Step 7. This prevents another transaction from reading a location written by a privatizing transaction before the privatizing transaction has done all waiting needed to ensure the buffer is privatized.

Despite these constraints, there are many options for ordering the substeps in this algorithm. Several considerations apply. First, the main point of the CIs is to reduce the amount of time a privatizing transaction must wait for a conflicting reader. This suggests decrementing CIs as soon as it is safe to do so, as shown above. Similarly, delaying waiting steps allows more time for the waited-for condition to become true, while other useful work continues to be performed. This suggests waiting for CIs to become zero as late as possible.

On the other hand, some loops may be combined, which may improve single-threaded performance at the cost of slightly *increased* CI hold times. In our implementation, we combine the substeps of Steps 4 and 6 into a single loop, which iterates over orecs in the write set, for each one performing copybacks for locations covered by the orec, and then waiting for the CI to become zero. We also perform Step 5 after Step 7, because the sooner a transaction releases an orec, the sooner another can acquire it. We have not yet experimented extensively with alternative orderings.

5.2 CI optimizations

Various enhancements to the basic algorithm can reduce the cost of updating the CIs.

No overhead for read-only transactions: Since a transaction must wait only for conflicting readers that might write to a buffer it has isolated, it need not wait for read-only transactions. Therefore, a read-only transaction can completely avoid the overhead of updating CIs. Also, a read-only transaction never waits for any CI to become zero since it does not have write ownership of any orec. Hence, our privatization scheme imposes no overhead on read-only transactions.

Scalable committing indicators: Like read indicators, scalable and efficient committing indicators can be implemented using SuperSNZI [9] (see Section 4). When using such CIs, however, it may not always be possible to atomically validate the orec and update the corresponding CI, as described in Step 2 of the commit operation.[6] If so, it is important to perform the update of the corresponding CI *before* the orec is read-validated. Otherwise, there would be a window between the time the orec is read-validated and the time the CI is updated, in which an isolating transaction may acquire write ownership of the orec and commit successfully, without waiting for the reader to finish its updates to the isolated buffer, and without causing that reader to abort (as the reader has already validated the orec). Updating the CI before validating the orec guarantees that a writer will always wait for all conflicting readers that might still be copying values back to memory.

5.3 Interaction with hardware transactions

Our privatization algorithm is designed for the SkySTM runtime, which enables hardware and software transactions to run concurrently. SkySTM guarantees that hardware and software transactions do not violate one another's atomicity by having a hardware transaction read STM metadata to detect any conflicts with software transactions. If a conflict is detected, the hardware transaction aborts itself. Otherwise, because the metadata is checked as part of the hardware transaction, the transaction can commit only if the metadata does not change before the hardware transaction commits. Thus, when a hardware transaction does commit successfully, it is guaranteed that there is no conflict with any software transaction.

Hardware transactions cannot violate the privatization guarantee by writing to a buffer isolated by a *software* transaction, because the software transaction causes any hardware transaction that may access the isolated buffer to abort when the buffer is isolated. Therefore, to provide the privatization guarantee when hardware transactions are involved, the only additional guarantee we need is that if a *hardware* transaction isolates a buffer, that buffer will not later be written by a software transaction. Unfortunately, we cannot use the same solution that is used by software transactions (waiting for CIs to be zero), because once a hardware transaction observes a nonzero CI, it will be aborted when the CI changes. Therefore, for hardware transactions, we simply abort the hardware transaction if it needs to write to a location whose corresponding CI is nonzero (this additional test adds no overhead because the orec for each location to which the transaction writes is checked in any case).

[6] For example, for an STM with invisible reads, performance is likely to dictate that the CI should not be colocated with the rest of the orec.

5.4 Using our solution with in-place writes

With an STM that uses in-place writes, it is not true that transactions can only write to an isolated buffer after they have committed successfully. It is still true, however, that such a transaction must have read something written by the isolating transaction, and hence that the writing transaction is no longer valid once the isolating transaction has committed successfully. Therefore, we can provide the privatization guarantee as follows:

- A transaction increments the appropriate CI immediately after the read operation, and decrements it after it has executed all writes to shared data (including writes whose purpose is to undo the writes done by an aborted transaction).
- Similar to the solution for an STM with deferred writes, once it commits successfully, a transaction waits until the CIs corresponding to its write operations become zero.

Doing this guarantees that a successfully committed isolating transaction waits until all transactions that have read memory that the isolating transaction has subsequently written (and are thus doomed) find out that they are aborted and finish undoing all their write operations, hence providing the privatization guarantee.

The optimization for read-only transactions can also be applied to this variant, by deferring increments to the CIs until the first write operation by the transaction is executed.

This variant is unlikely to scale as well as the deferred-writes variant, as a transaction must wait until all transactions with which it has a write-read conflict validate their read sets, find out that they are aborted, and undo their writes. In particular, the solution for in-place writes may entail waiting for arbitrary user code to finish executing before the next read-validation is executed. On the other hand, this solution will probably still scale better than a solution based on quiescence [14], which waits for *all* running transactions to read-validate, and requires reading their snapshots of a global counter, entailing significant coherence traffic [32].

5.5 Stronger privatization guarantees

By using the privatization mechanism as presented so far, and by read-validating after every transactional read, SkySTM avoids both kinds of privatization problems identified in [20], namely delayed cleanup and doomed transactions. Nonetheless, in some cases, it may be desirable to provide a stronger privatization guarantee so that transactions do not perform even *reads* from an isolated buffer after the isolating transaction commits. This guarantee is not provided by our solution: it does not wait for all conflicting aborted transactions to finish, and hence these transactions may read the buffer after they are aborted. Such a transaction is guaranteed to abort before returning to user code, so there is no risk of returning inconsistent data. However, once a thread has isolated a buffer, it may deallocate its memory. Subsequently, the operating system may deallocate the page containing that memory, so accessing it, even just by reading it, could cause the program to crash.

One approach to providing this stronger guarantee is to use a transaction-aware memory management system [14], which can then employ explicit privatization barriers [27]. Such a solution is acceptable if allocation/deallocation is relatively infrequent. A simpler approach for some environments is to use nonfaulting loads to guarantee that reading from a buffer that was freed will not cause a runtime error.

Avoiding transactional reads of privatized buffers also avoids the doomed transaction problem in TL2, which is susceptible to this problem because fast read validation in TL2 guarantees only that the location being read-validated has not been written by another transaction since the reading transaction began. Thus, if one transaction reads a location to determine that a particular memory buffer is not private, and another transaction subsequently privatizes that buffer, writing the location read by the first transaction, and then writes that buffer nontransactionally, then the first transaction (now doomed) might read the newly privatized buffer without detecting the conflict. The Ord variant [20] avoids such inconsistent reads of privatized buffers by requiring a transaction to revalidate its entire read set whenever the global version number changes. SkySTM does not have this problem because its read validation guarantees that all values read are valid and mutually consistent at some point *during the read validation*. As reads from a privatized buffer are performed only by doomed transactions, the read validation operation that follows the first read from the isolated buffer will detect that the transaction is doomed, and abort it.

Proxy privatization For ease of exposition, the descriptions above focus on the case in which a thread uses a transaction to privatize a memory buffer and then accesses the buffer nontransactionally after the transaction completes. However, privatization need not be so direct—one thread might use a transaction to privatize a buffer for another. We call this scenario *proxy privatization*. In this case, a stronger guarantee is needed: latent writes to the privatized buffer must be prevented even before the privatizing transaction completes its commit, because the other thread may access the buffer before then. Our algorithm and (our implementation of) Ord both guarantee that this idiom works correctly.

It remains to be seen exactly how ongoing efforts to precisely define language semantics for use with transactions will address the privatization problem. However, it seems clear that useful language semantics for contexts in which nontransactional memory accesses to previously transactional data are desirable or unavoidable must outlaw the undesirable behaviors that comprise the privatization problem. We therefore believe that the scalable privatization guarantee made by SkySTM—and the mechanisms we have introduced to achieve it—are likely to be useful in implementing whatever language semantics are eventually specified.

5.6 Evaluation

In this section we examine the cost of supporting privatization by comparing the performance of SkySTM to a variant with privatization disabled, as well as comparing TL2 to the privatization-enabled Ord variant [20]. Figure 4 presents the throughput of the HashTable benchmark (regular version, no indirection).[7] For SkySTM, both versions scale similarly, with the privatization solution paying a relative overhead of around 4% across the board. This demonstrates one of the most important advantages of our algorithm: when the application scales, so does our privatization algorithm. In contrast, the Ord algorithm pays dearly for privatization, scaling substantially worse than TL2-GV4. As explained earlier, the modifications to TL2-GV4 to achieve the privatization-safe Ord algorithm cannot be applied to TL2-GV6.

We do not claim that our privatization algorithm is almost free, only that its cost is small when contention is low. As contention rises, our privatization solution will likely require more transactions to wait for each other, and hence its overhead is likely to increase.

To evaluate our solution with different levels of contention, we measured the privatization overhead with a red-black tree benchmark, using different key ranges (and tree sizes) to control contention levels. In all experiments, the tree is initialized with half the keys of the tested range, and in each iteration a thread chooses whether to insert, delete, or lookup a random key with distribution

[7] This benchmark has little contention and no memory is actually privatized. We are looking at the cost of enabling an application to privatize memory. Privatizing memory has no cost itself other than the cost of the conflicts it may introduce.

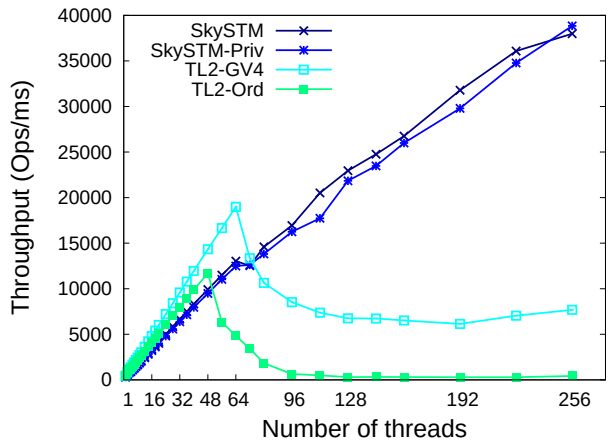

Figure 4. Performance cost of enabling privatization. HashTable, *keyrange* = 128,000, 50% insertions, 50% deletions.

2:2:6. The results of these experiments (that we omit due to lack of space) show that the privatization *overhead* of our solution strongly depends on the level of contention. With all tree sizes, the run-time overhead of our privatization solution in the single-thread run was approximately 3.6%. With 256 threads, this overhead increases to 11.8% with a 128,000 key range, and to 35% with a smaller 1,600 key range. On the other hand, the overhead of the Ord algorithm (comparing to the TL2-GV4 variant) is over 1000% in the 256-thread run with *both* the 1,600 and 128,000 key ranges.

6. Concluding remarks

We have presented SkySTM, the first STM system that can scale to hundreds of threads across multiple multicore chips. The key to achieving scalability is a focus on conflict-based synchronization so that, given a scalable application with few conflicts, the STM will not introduce contention on metadata that will impede scalability. The key mechanisms we used to achieve this are:

- semivisible reads for containing the overhead of ensuring consistent execution;
- scalable non-zero indicators (SNZIs) to make read sharing with semivisible reads scalable; and
- a novel conflict-based implicit privatization algorithm that is integrated with the mechanisms above to avoid synchronization between transactions that do not conflict at the application level.

SkySTM, which we are making available as an open-source library [1], is part of a hybrid transactional memory system that can exploit best-effort hardware transactional memory to boost performance. While we are pleased with the scalability of SkySTM, many opportunities to improve it remain. In particular, further reducing overhead while maintaining scalability is an important goal.

Acknowledgments: The authors wish to thank Dave Dice and Nir Shavit for useful discussions. They also identified the problem that came to be known as the privatization problem [8].

References

[1] http://research.sun.com/scalable/.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, Feb. 2005.

[3] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA*, 2008.

[4] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*, Jun 2007.

[5] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, 2006.

[6] D. Dice, November 2008. Personal communication.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006.

[8] D. Dice and N. Shavit. What really makes transactions faster? In *Transact*, 2006.

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

[10] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

[11] G. Grohoski. Niagara-2: A highly threaded server-on-a-chip. Hot Chips 18, 2006.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[13] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[14] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: A scalable transactional memory allocator. In *ISMM*, 2006.

[15] Y. Lev and J.-W. Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *PPoPP*, 2008.

[16] Y. Lev and M. Moir. Fast read sharing mechanism for software transactional memory. In *PODC*, 2004.

[17] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Transact*, 2007.

[18] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. In *DISC*, 2005.

[19] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Transact*, 2006.

[20] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *ICPP*, 2008.

[21] M. Olszewski, J. Cutler, and J. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *PACT*, Sept. 2007.

[22] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA*, 2005.

[23] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA*, 2007.

[24] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP*, 2006.

[25] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO*, 2006.

[26] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 1995.

[27] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. Technical Report 915, Univ. of Rochester, 2007.

[28] M. F. Spear, V. J. Marathe, W. N. S. III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC*, 2006.

[29] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable transactions with a single atomic instruction. In *SPAA*, 2008.

[30] The TL2 library. http://research.sun.com/scalable.

[31] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA*, 2007.

[32] R. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *SPAA*, 2008.