View metadata, citation and similar papers at core.ac.uk

brought t

provided by Elsevier -

ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 58 (2005) 310–324

Science of Computer Programming

www.elsevier.com/locate/scico

Snapshots and software transactional memory[☆]

Christopher Cole[a,*], Maurice Herlihy[b]

[a] Northrop Grumman Mission Systems, 88 Silva Lane, Middletown, RI 02842, United States
[b] Department of Computer Science, Brown University, Providence, RI 02912, United States

Received 1 November 2004; received in revised form 15 January 2005; accepted 1 March 2005
Available online 13 June 2005

Abstract

One way that software transactional memory implementations attempt to reduce synchronization conflicts among transactions is by supporting different kinds of access modes. One such implementation, Dynamic Software Transactional Memory (DSTM), supports three kinds of memory access: WRITE access, which allows an object to be observed and modified, READ access, which allows an object to be observed but not modified, and RELEASE access, which allows an object to be observed for a limited duration.

In this paper, we examine the relative performance of these modes for simple benchmarks on a small-scale multiprocessor. We find that on this platform and for these benchmarks, the READ and RELEASE access benchmarks do not substantially increase transaction throughput (and sometimes reduce it). We blame the extra bookkeeping inherent in these access modes.

In response, we propose a new SNAP access mode. This mode provides almost the same behavior as RELEASE mode, but admits much more efficient implementations.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Synchronization; Transactions; Transactional memory

1. Introduction

Dynamic Software Transactional Memory (DSTM) [7] is an application programming interface for concurrent computations in which shared data is synchronized without

☆ Supported by NSF grant 0410042, and by grants from Sun Microsystems and Intel Corporation.
* Corresponding author.
E-mail addresses: chris.cole@ngc.com (C. Cole), herlihy@cs.brown.edu (M. Herlihy).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.03.006

using locks. DSTM manages a collection of *transactional objects*, which are accessed by *transactions*. A transaction is a short-lived, single-threaded computation that either *commits* or *aborts*. If the transaction commits, then these changes take effect; otherwise, they are discarded. A *transactional object* is a container for a regular Java object. A transaction can access the contained object by *opening* the transactional object, and then reading or modifying the regular object. Transactions are *linearizable* [8]: they appear to take effect in a one-at-a-time order.

If two transactions open the same object at the same time, a *synchronization conflict* occurs, and one of the conflicting transactions must be aborted. To reduce synchronization conflicts, an object can be opened in one of several *access modes*. An object opened in *WRITE* mode can be read or modified, while an object opened in *READ* mode can only be read. WRITE mode conflicts with both READ and WRITE modes, while READ mode conflicts only with WRITE.

DSTM also provides *RELEASE* mode,[1] a special kind of read-only mode that indicates that the transaction may *release* the object before it commits. Once such an object has been released, concurrent accesses of any kind do not cause synchronization conflicts. It is the programmer's responsibility to ensure that releasing objects does not violate transaction linearizability.

The contribution of this paper is to examine the effectiveness of these access modes on a small-scale multiprocessor. We find that the overhead associated with READ and RELEASE modes mostly outweighs any advantage in reducing synchronization conflict. To address this issue, we introduce a novel *SNAP* (snapshot) mode, an alternative to RELEASE mode with much lower overhead. SNAP mode provides almost the same behavior as RELEASE, but much more efficiently.

2. Related work

Transactional memory was originally proposed as a hardware architecture [6,16], and continues to be the focus of hardware-oriented research [13]. There have also been several proposals for software transactional memory and similar constructs [1,2,9,12,15]. Others [10,14] have studied the performance of read/write locks.

An alternative approach to software transactional memory (STM) is due to Harris and Fraser [5]. Their STM implementation is *word-based*: the unit of synchronization is a single word of memory. An uncontended transaction that modifies N words requires $2N+1$ compare-and-swap calls. Fraser [4] has proposed a FSTM implementation that is *object-based*: the unit of synchronization is an object of arbitrary size. Here, an uncontended transaction that modifies N objects also requires $2N + 1$ compare-and-swap calls. Herlihy et al. [7] have proposed an object-based DSTM implementation, described below, in which an uncontended transaction that modifies N objects requires $N + 1$ compare-and-swap calls, but sometimes requires traversing an additional level of indirection. In both object-oriented STM implementations, objects must be copied before they can be modified. Marathe and Scott [11] give a more detailed comparison of these STM implementations.

[1] Sometimes called TEMP mode [7].

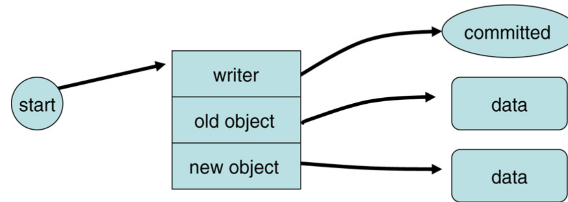

Fig. 1. A transactional object consists of a start pointer and a locator with pointers to the most recent writer, the old version, and the new version.

One important difference between FSTM and DSTM is that the former does not guarantee that an aborted transaction observes a consistent state, a property sometimes called *strict isolation*. In the absence of strict isolation, any transaction that observes an inconsistent state will eventually abort, but before it aborts it may perform illegal actions such as dividing by zero or indexing off the end of an array. DSTM, by contrast, guarantees strict isolation when opening a transactional object.

3. DSTM implementation

Here we summarize the relevant aspects of the DSTM implementation (a more complete description appears elsewhere [7]). In its simplest form, a transactional object has three fields: (1) the `writer` field points to the most recent transaction to open the object in WRITE mode, (2) the `oldVersion` field points to the old version of the object, and (3) the `newVersion` field points to the new version. The object's actual value is determined by the status of the `writer` transaction. If it is committed, then the new version is current, and otherwise the old version is current. If the transaction is active, then the old version is current, and the new version is the `writer` transaction's tentative version, which will become current if and only if that transaction commits.

Ideally, when a transaction opens an object for WRITE, we would like to set the `writer` field to point to that transaction, the `oldVersion` field to point to the current version, and the `newVersion` field to point to a copy of the current version. We cannot make an atomic change to multiple fields, but we can get the same effect by introducing a level of indirection: each transaction object has a single reference field `start`, which points to `locator` structure that contains the `writer`, `oldVersion`, and `newVersion` fields (see Fig. 1). We can change these fields atomically simply by preparing a new locator with the desired field values, and using compare-and-swap to swing the pointer from the old locator to the new one. Figs. 2 and 3 show the process of opening a transactional object in WRITE mode whose most recent writer either aborted or committed.

It is also possible that a transaction attempting to open an object discovers that the most recent writer is still active. The opening transaction may decide either to back off and give the writer a chance to complete, or to proceed, forcing the writer to abort. This policy decision is handled by a separate *Contention Manager* module.

Each time a transaction opens an object, in any mode, the transaction checks whether it has been aborted by a synchronization conflict, a process called *validation*. This check

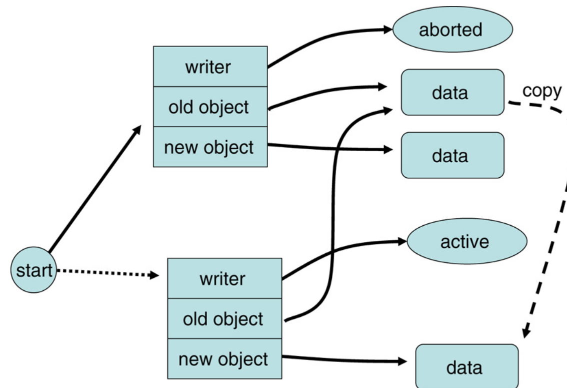

Fig. 2. Opening a transactional object when the most recent writer aborted: the old version is the former old version, and the new version is a copy of the old.

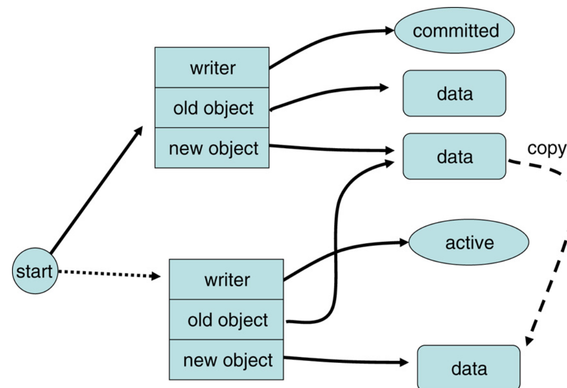

Fig. 3. Opening a transactional object when the most recent writer committed: the old version is the former new version, and the new version is a copy of the new.

prevents an aborted transaction from wasting resources, and also ensures that each transaction has a consistent view of the transactional objects.

Opening an object in WRITE mode requires creating a new version (by copying the old one) and executing a compare-and-swap instruction. When an object is opened in READ mode, however the transaction simply returns a reference to the most recently committed version. The transaction records that reference in a private *read table*. To validate, the transaction checks whether each of its version references is still current. This implementation has the advantage that reading does not require an expensive compare-and-swap instruction. It has two disadvantages: validation takes time linear in the number of objects read, and the contention manager cannot tell whether an object is open in READ mode. For this reason, we call this implementation the *invisible read*.

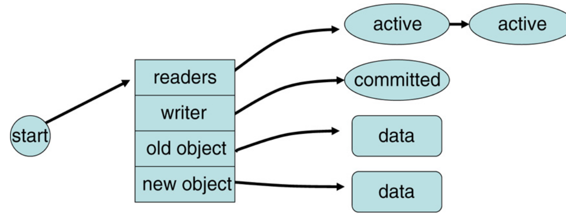

Fig. 4. Visible read: in addition to keeping track of the last writer, we also keep track of a linked list of current readers.

Because of these disadvantages, we devised an alternative READ mode implementation, which we call the *visible read*. This implementation is similar to WRITE mode, except that it does not copy the current version, and the object keeps a list of reading transactions (see Fig. 4). Validating a transaction takes constant time, and reads are visible to the contention manager. Each read does require a compare-and-swap, and opening an object in WRITE mode may require traversing a list of prior readers.

Similarly, RELEASE mode also has both visible and invisible implementations. Releasing an object either causes the version to be discarded (invisible) or the reader removed from the list (visible).

4. Benchmarks

An `IntSet` is an ordered linked list of integers providing `insert()` and `delete()` methods. We created three benchmarks: WRITE, READ, and RELEASE. Each benchmark runs for twenty seconds randomly inserting or deleting values from the list. The WRITE benchmark opens each list element in WRITE mode. The READ benchmark opens each list element in READ mode until it discovers the element to modify, which it reopens in WRITE mode. The RELEASE benchmark opens each element in RELEASE mode, releasing each element after opening its successor (similar to lock coupling). Each benchmark was run using the `Polite` contention manager which uses exponential back-off when conflicts arise. For example, when transaction A is about to open an object already opened by transaction B, the `Polite` contention manager backs off several times, doubling each expected duration, to give B a chance to finish. If B does not finish in that duration, then A aborts B, and proceeds.

The benchmarks were run on a machine with four Intel Xeon processors. Each processor runs at 2.0 GHz (not hyperthreaded) and has 1 GB of RAM. The machine was running Debian Linux and each benchmark was run 100 times for twenty seconds each. The performance data associated with individual method calls was extracted using the Extensible Java Profiler [3]. Each benchmark was run using 1, 4, 16, 32, and 64 threads. The single-thread case is interesting because it provides insight into the amount of overhead the benchmark incurred. In the four-thread benchmarks, the number of threads matches the number of processors, while the benchmarks using 16, 32, and 64 thread show how the transactions behave when they share a processor. To control the list size, the integer values range only from 0 to 255.

Table 1
Single-thread throughput: the single-processor throughput (transactions committed per millisecond) for both the invisible and visible implementations

	Invisible	Visible
WRITE	(36.6)	(22.3)
READ	13.5%	107.3%
RELEASE	54.5%	95.2%

5. Benchmark results

We found that throughput measurements are affected by the cost of maintaining the read-only table in the invisible implementation and the readers list in the visible implementation. We start by examining single-thread results to highlight the relative overheads of the two approaches.

5.1. Single-thread results

Table 1 shows the single-processor throughput (transactions committed per millisecond) for both the invisible and visible implementations. In the single-thread benchmarks, there is no concurrency, and hence no synchronization conflicts, so the throughput numbers reflect the modes' inherent overheads.

To ease comparisons, in this table and in later tables, we give the global transactions-per-millisecond (TPM) throughput only for the WRITE benchmark. We give the other benchmarks as relative percentages of the corresponding WRITE benchmark. For example, in Table 3, invisible WRITE has throughput 36.6 TPM, and invisible READ is shown as 13.5%, implying a raw throughput of 4.9 TPM.

The invisible WRITE benchmark had better throughput than the visible WRITE benchmark because the invisible WRITE incurs no overhead synchronizing with readers. The visible WRITE, by contrast, checks whether any transaction has the object open in READ mode. Even though there are no such transactions (in a single-threaded benchmark), the check takes time.

The invisible RELEASE benchmark has higher throughput than the READ benchmark because each object released reduces the number of transactions that must by validated at each API call. The invisible READ benchmark also suffers because the contention manager cannot detect when a read is in progress, so any writer will immediately abort any concurrent readers without giving them the chance to finish.

Both the visible and invisible READ and RELEASE benchmarks perform poorly compared to the corresponding WRITE benchmarks because of the overhead of maintaining either the read-only table or the readers list.

We now outline the costs as observed by profiling the benchmarks. (These numbers are summarized in Table 2.) As the invisible READ benchmark traverses the list, it takes approximately 280 ns to open each object in read mode. When it finds an object it

Table 2
Common method call timings (nanoseconds)

	Invisible	Visible
WRITE	180	730
READ & RELEASE	280	135
UPGRADE	250	160
RELEASE	90	40

Table 3
Invisible implementation: the transactions-per-millisecond throughput of the invisible implementation for varying numbers of threads

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(36.6)	(35.7)	(32.9)	(29.6)	(24.7)
READ	13.5%	4.7%	1.7%	1.8%	2.2%
RELEASE	54.5%	23.7%	12.6%	12.8%	15.1%

needs to modify, it takes approximately 250 ns to upgrade to write access. Similarly, RELEASE takes approximately 370 ns to open each object (280 ns to open the object and 90 ns to release it). Compare these numbers with the WRITE benchmark, which takes approximately 180 ns to open each object, and requires no additional work to modify an object. As the visible READ benchmark traverses the list, it takes approximately 565 ns to open each object in read mode. When it finds an object it needs to modify it takes at least 1000 ns to upgrade to write access. The principal cost of upgrading is traversing the ever-growing readers list to determine which of the readers is still active. This cost grows as readers accumulate. Similarly, the RELEASE benchmark takes approximately 680 ns to open each object (500 ns to open the object and 180 ns to release it). When RELEASE upgrades an object for writing, it too incurs the cost of detecting active transactions in the readers list. Compare this with the visible WRITE benchmark, which takes approximately 730 ns to open each list element. Although WRITE takes longer to traverse each list element, no further work is required to modify an object.

When comparing the visible and invisible implementations, bear in mind that the visible implementation does much of the work necessary to allow concurrent access to objects in the open methods, while the invisible implementation requires every API call to verify the transaction's state and therefore its entire overhead is not reflected in the open methods alone.

We now turn our attention from single-thread executions, where overhead dominates, to multithread executions, where we might expect to see gains for the READ or RELEASE benchmarks due to reduced synchronization conflicts.

5.2. Multithreading results

Table 3 shows the transactions-per-millisecond throughput of the invisible implementation for varying numbers of threads, and Table 4 does the same for the visible implementation.

Table 4
Visible implementation: the transactions-per-millisecond throughput of the visible implementation for varying numbers of threads

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(22.3)	(23.1)	(21.4)	(20.0)	(17.6)
READ	52.6%	0.3%	0.2%	0.2%	0.1%
RELEASE	52.6%	0.1%	0.3%	0.4%	0.3%

Table 5
Invisible read transactions/milliseconds with work

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(19.5)	(19.1)	(17.7)	(16.4)	(13.6)
READ	16.9%	8.0%	3.5%	3.1%	2.9%
RELEASE	57.0%	28.5%	19.1%	18.8%	21.3%

Table 6
Visible read transactions/milliseconds with work

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(2.6)	(2.6)	(2.5)	(2.4)	(2.3)
READ	93.1%	1.0%	0.8%	0.8%	0.5%
RELEASE	91.8%	0.2%	0.3%	0.7%	0.9%

Surprisingly, perhaps, the concurrency allowed in READ and RELEASE did not overcome the overhead in either implementation (with one minor exception). In the invisible implementation, a transaction takes an excessive amount of time to traverse the list because it must validate its read-only table with each DSTM API call. A transaction attempting to insert a large integer may never find the integer's position in the list before being aborted. In the visible implementation, the single-threaded benchmark has a slight advantage because it does not need to copy the version being opened. In the multithreaded benchmarks, however, the visible implementation incurs additional overhead because it must traverse and prune a non-trivial list of readers.

We went on to investigate how the benchmarks perform when transactions do some "work" while holding the objects. We had each transaction count to a random number between 0 and 500,000 before trying to commit the transaction. As illustrated in Tables 5 and 6, adding work has a larger negative impact on the visible implementation. Nevertheless, the throughput of the READ and RELEASE benchmarks continue to trail the throughput of the WRITE benchmark.

5.3. *Optimized visible read*

The visible read implementation considered so far allows the list of readers to grow until the object is opened by a writer. We now investigate an alternative implementation in

Table 7
Optimized visible read transactions/milliseconds

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(22.3)	(23.1)	(21.4)	(20.0)	(17.6)
READ	107.3%	0.3%	0.9%	1.1%	1.2%
RELEASE	95.2%	0.1%	0.7%	1.0%	1.6%

which readers “prune” inactive readers from the list. This reduces the number of readers in the list at the cost of increasing the cost of traversing those readers.

As shown in Table 7, the optimized implementation removes much of the overhead; the single-thread READ and RELEASE benchmarks produces results similar to the WRITE benchmark. The optimized READ and RELEASE both improved under contention, but the throughput is still significantly less than under the WRITE benchmark. The principal barrier to better throughput is the cost of trimming the readers list. This cost is minimal in the single-thread case because the list has at most one reader, but once contention is introduced trimming the list can take anywhere from 100 to 500 ns depending on how many transactions are in the list. If this cost is added to the 565 and 680 ns the READ and RELEASE benchmarks take to open objects, it is easy to see why these implementations have less throughput than the WRITE benchmark which takes only 730 ns to open an object.

The cost of maintaining the read-only and readers lists means that we were unable to get significant increases in throughput using any of the READ or RELEASE benchmarks. This observation prompted an investigation into other methods of reducing the overhead.

6. Snapshot mode

In an attempt to find a low-overhead alternative, we devised a new *snapshot* mode for opening an object.

```
TMObject<T> tmObject;
...
T version = tmObject.openSnap();
```

In this code fragment, the call to `openSnap()` returns a reference to the version that would have been returned by a call to `openRead()`. It does not actually open the object for reading, and the DSTM does not keep any record of the snapshot. All methods throw `DeniedException` if the current transaction has been aborted.

The `version` argument to the next three methods is a version reference returned by a prior call to `openSnap()`.

```
try {
  tmObject.snapValidate();
  } catch (SnapshotException e) {
    ...
  }
```

The call returns normally if a call to `openSnap()` (or `openRead()`) would have returned the same version reference, and otherwise it throws `SnapshotException`. Throwing this exception does not abort the current transaction, allowing the transaction to retry another snapshot.

```
tmObject.snapUpgradeRead(version);
```

If the version argument is still current, this method opens the object for reading, and otherwise throws (`SnapshotException`).

```
T newVersion = tmObject.snapUpgradeWrite(version)
```

If the version argument is still current, this method returns a version of the object open for writing, and otherwise throws (`SnapshotException`).

Objects opened in RELEASE mode are typically used in one of the following three ways. Most commonly, an object is opened in RELEASE mode and later released. The transaction will be aborted if the object is modified in the interval between when it is opened and when it is released, but the transaction will be unaffected by modifications that occur after the release.

```
TMObject<Entry> tmObject;
...
Entry entry = tmObject.openRelease();
...
tmObject.release();
```

The same effect is achieved by the following code fragment:

```
Entry entry = tmObject.openSnap();
...
tmObject.snapValidate(entry);
```

The first call returns a reference to the object version that would have been returned by `openRelease()` (or `openRead()`), and the second call checks that the version is still valid. There is no need for an explicit release because the transaction will be unaffected if that version is changed (assuming it does not validate again).

Sometimes an object is opened in RELEASE mode and never released (which is equivalent to opening the object in READ mode). To get the same effect in SNAP mode, the transaction must apply `snapUpgradeRead()` to the object, atomically validating the snapshot and acquiring READ access.

Finally, an object may be opened in RELEASE mode and later upgraded to WRITE mode. The `snapUpgradeWrite()` method provides the same effect.

To illustrate how one might use SNAP mode, Fig. 5 shows the code for a `insert()` method based on SNAP mode. It is not necessary to understand this code in detail, but there are three lines that merit attention. As the transaction traverses the list, `prevObject` is a reference to the last transactional object accessed, and `lastObject` is a reference to that object's predecessor in the list. In the line marked A, the method validates for the last time that `lastObject` is still current, effectively releasing it. If the method discovers that

Table 8
SNAP with invisible

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(36.6)	(35.7)	(32.9)	(29.6)	(24.7)
READ	13.5%	4.7%	1.7%	1.8%	2.2%
RELEASE	54.5%	23.7%	12.6%	12.8%	15.1%
SNAP	170.7%	46.8%	33.2%	34.6%	39.0%

Table 9
SNAP with visible

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(22.3)	(23.1)	(21.4)	(20.0)	(17.6)
READ	107.3%	0.3%	0.9%	1.1%	1.2%
RELEASE	95.2%	0.1%	0.7%	1.0%	1.6%
SNAP	469.9%	269.6%	263.2%	210.7%	214.9%

the value to be inserted is already present, then in the line marked B, it upgrades access to the predecessor entry to READ, ensuring that no other transaction deletes that value. Similarly, if the method discovers that the value to be inserted is not present, it upgrades access to the predecessor entry to WRITE, so it can insert the new entry.

The principal benefit of SNAP mode is that it can be implemented very efficiently. This mode is "stateless", in the sense that the DSTM runtime does not need to keep track of versions opened in SNAP mode (unlike READ mode). The `snapValidate()`, `snapUpgradeRead()` and `snapUpgradeWrite()` calls simply compare their arguments to the object's current version. Moreover, SNAP mode adds no overhead to transaction validation.

7. SNAP benchmarks

The results of running the same benchmark in SNAP mode instead of RELEASE mode are shown in Table 8 (invisible) and Table 9 (visible). For both visible and invisible implementations, SNAP mode has substantially higher throughput than both READ and RELEASE mode. Opening an object in SNAP mode takes about 100 ns, including validation. It takes about 125 ns to upgrade an object opened in SNAP mode to WRITE mode.

Even though invisible SNAP mode outperforms invisible READ and RELEASE, it still has lower throughput than invisible WRITE. We believe this disparity reflects inherent inefficiencies in the invisible READ implementation. The invisible SNAP implementation must upgrade to invisible READ mode whenever it observes that a value is absent (to ensure it is not inserted), but transactions that open objects in invisible READ mode are often aborted, precisely because they are invisible to the contention manager.

```
  public boolean insert(int v) {
    List newList = new List(v);
    TMObject<List> newNode = new TMObject<List>(newList);
    TMThread thread = TMThread.currentThread();
    while (thread.shouldBegin()) {
      thread.beginTransaction();
      boolean result = true;
      try {
        TMObject<List> lastNode = null;
        List lastList = null;
        TMObject<List> prevNode = this.first;
        List prevList = prevNode.openSnap();
        TMObject<List> currNode = prevList.next;
        List currList = currNode.openSnap();
        while  (currList.value < v) {
          if (lastNode != null)
/*A*/       lastNode.snapValid(lastList);
          lastNode = prevNode;
          lastList = prevList;
          prevNode = currNode;
          prevList = currList;
          currNode = currList.next;
          currList = currNode.openSnap();
        }
        if (currList.value == v) {
/*B*/      prevNode.snapUpgradeRead(prevList);
          result = false;
        } else {
          result = true;
/*C*/     prevList = prevNode.snapUpgradeWrite(prevList);
          newList.next = prevList.next;
          prevList.next = newNode;
        }
        // final validations
        if (lastNode != null)
          lastNode.snapValid(lastList);
        currNode.snapValid(currList);
      } catch (SnapshotException s) {
        thread.getTransaction().abort();
      } catch (DeniedException d) {
      }
      if (thread.commitTransaction()) {
        return result;
      }
    }
    return false;
  }
```

Fig. 5. SNAP-mode insert method.

Table 10
Invisible read transactions/milliseconds with work

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(19.5)	(19.1)	(17.7)	(16.4)	(13.6)
READ	16.9%	8.0%	3.5%	3.1%	2.9%
RELEASE	57.0%	28.5%	19.1%	18.8%	21.3%
SNAP	162.7%	51.0%	35.2%	36.5%	41.1%

Table 11
Visible read transactions/milliseconds with work

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(2.6)	(2.6)	(2.5)	(2.4)	(2.4)
READ	93.1%	1.0%	0.8%	0.8%	0.5%
RELEASE	91.8%	0.2%	0.3%	0.7%	0.9%
SNAP	133.9%	116.0%	66.4%	68.6%	71.3%

Table 12
Visible with 50% modification

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(22.1)	(23.0)	(21.4)	(19.6)	(17.2)
READ	105.6%	0.8%	2.3%	4.8%	6.6%
RELEASE	94.2%	0.4%	2.2%	4.3%	6.3%
SNAP	491.5%	352.1%	337.7%	302.9%	184.0%

Table 13
Visible with 10% modification

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(22.3)	(22.9)	(20.9)	(19.4)	(17.2)
READ	106.0%	2.3%	13.4%	20.1%	18.2%
RELEASE	94.7%	1.7%	11.7%	16.7%	17.6%
SNAP	491.6%	379.4%	420.7%	300.7%	115.1%

While the result of combining invisible READ and SNAP modes is disappointing, the result of combining visible READ and SNAP modes is dramatic: here is the first alternative mode that outperforms WRITE mode across the board. Tables 10 and 11 show introducing work does not change the relative performance of these implementations.

To investigate further, we implemented some benchmarks that mixed "modifying" method calls with "observer" (read-only) method calls. We introduced a `contains()` method that searches the list for a value. We tested benchmarks in which the percentages of modifying calls (`insert()` and `delete()`) varied were 50% (Table 12), 10% (Table 13),

Table 14
Visible with 1% modification

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(20.9)	(20.9)	(20.3)	(18.0)	(15.9)
READ	105.9%	24.4%	56.8%	43.3%	11.8%
RELEASE	99.4%	25.7%	16.9%	16.3%	16.7%
SNAP	501.2%	466.0%	483.4%	348.9%	154.7%

Table 15
Visible with 0% modification

	1 Thread	4 Threads	16 Threads	32 Threads	64 Threads
WRITE	(11.5)	(11.8)	(11.0)	(10.2)	(9.0)
READ	107.7%	76.1%	42.3%	23.6%	6.0%
RELEASE	95.6%	27.5%	21.6%	21.5%	22.4%
SNAP	531.0%	472.4%	555.9%	662.5%	790.0%

1% (Table 14), and 0% (Table 15). Each of the SNAP mode benchmarks had higher throughput than its WRITE counterpart, and was the only benchmark to do so.

8. Conclusions

Naturally, these results are valid only for the specific implementation and platform tested here. It may be that platforms with more processors, or a different contention manager, or different internals would behave differently. Nevertheless, for the time being, most multiprocessors will be small-scale multiprocessors and our results apply directly to such platforms.

More research is needed to determine the most effective methods for opening objects concurrently in software transactional memory. We were surprised by how poorly READ and RELEASE modes performed on our small-scale benchmarks. While our SNAP mode implementation substantially outperforms both READ and RELEASE modes, it is probably appropriate only for advanced programmers. It would be worthwhile investigating whether or not a contention management scheme could increase the throughput of read transactions, or if there are more efficient designs for tracking objects open for reading.

As noted above, DSTM guarantees strict isolation, in the sense that every transaction, even one doomed to abort, sees a consistent set of objects. For the invisible READ, this guarantee is expensive, because each object read must be revalidated every time a new object is opened. The visible READ supports strict isolation much more efficiently. An alternative approach, used in Fraser's FSTM [4], does not guarantee that transactions see consistent states, but uses periodic checks and handlers to protect against memory faults and unbounded looping due to inconsistencies. The relative merits of these approaches remains open to further research.

References

[1] Yehuda Afek, Dalia Dauber, Dan Touitou, Wait-free made fast, in: Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, ACM Press, 1995, pp. 538–547.

[2] James H. Anderson, Mark Moir, Universal constructions for large objects, in: WDAG: International Workshop on Distributed Algorithms, in: LNCS, Springer-Verlag, 1995.

[3] http://ejp.sourceforge.net/.

[4] Keir Fraser, Practical lock-freedom, Technical Report UCAM-CL-TR-579, University of Cambridge Computer Laboratory, February 2004.

[5] Tim Harris, Keir Fraser, Language support for lightweight transactions, in: Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, ACM Press, 2003, pp. 388–402.

[6] Maurice Herlihy, J. Eliot B. Moss, Transactional memory: Architectural support for lock-free data structures, in: Proceedings of the Twentieth Annual International Symposium on Computer Architecture, 1993.

[7] Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, Software transactional memory for dynamic-sized data structures, in: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, ACM Press, 2003, pp. 92–101.

[8] Maurice P. Herlihy, Jeannette M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492.

[9] Amos Israeli, Lihu Rappoport, Disjoint-access-parallel implementations of strong shared memory primitives, in: Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, ACM Press, 1994, pp. 151–160.

[10] Theodore Johnson, Approximate analysis of reader and writer access to a shared resource, in: Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, ACM Press, 1990, pp. 106–114.

[11] Virendra J. Marathe, Michael L. Scott, A qualitative survey of modern software transactional memory systems, Technical Report 839, Department of Computer Science, University of Rochester, June 2004.

[12] Mark Moir, Transparent support for wait-free transactions, in: Proceedings of the 11th International Workshop on Distributed Algorithms, 1997, pp. 305–319.

[13] Ravi Rajwar, James R. Goodman, Transactional lock-free execution of lock-based programs, in: ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, ACM Press, New York, ISBN: 1-58113-574-2, 2002, pp. 5–17.

[14] Martin Reiman, Paul E. Wright, Performance analysis of concurrent-read exclusive-write, in: Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, ACM Press, 1991, pp. 168–177.

[15] Nir Shavit, Dan Touitou, Software transactional memory, in: Symposium on Principles of Distributed Computing, 1995, pp. 204–213.

[16] Janice M. Stone, Harold S. Stone, Philip Heidelberger, John Turek, Multiple reservations and the Oklahoma update, IEEE Parallel Distrib. Technol. 1 (4) (1993) 58–71.