Snapshots and Software Transactional Memory

Christopher Cole
Northrop Grumman Corporation
chris.cole@ngc.com

Maurice Herlihy
Brown University
Computer Science Department
herlihy@cs.brown.edu.

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 mode, which allows an object to be observed and modified, READ mode, which allows an object to be observed but not modified, and TEMP mode, 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 TEMP mode implementations do not substantially increase transaction throughput (and sometimes reduce it). We blame the extra bookkeeping inherent in these modes.

In response, we propose a new SNAP access mode. This mode provides almost the same behavior as TEMP mode, but admits much more efficient implementations.

1. INTRODUCTION

Dynamic Software Transactional Memory (DSTM) [7] is an application programming interface for concurrent computations in which shared data is synchronized without 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. Changes to objects opened by a transaction are not visible to other transactions until the transaction commits. (Changes are discarded if the transaction aborts.) Transactions are *linearizable* [8]: they appear to take effect in a one-at-a-time order.

PODC Workshop on Concurrency and Synchronization in Java Programs, July 25-26, 2004, St. John's, Newfoundland, Canada.

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 *TEMP* mode, 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 TEMP modes mostly outweighs any advantage in reducing synchronization conflict. To address this issue, we introduce a novel *SNAP* (snapshot) mode, an alternative to TEMP mode with much lower overhead. SNAP mode provides almost the same behavior as TEMP, 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 [2, 1, 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.

3. DSTM IMPLEMENTATION

Here we summarize the relevant aspects of the DSTM implementation (a more complete description appears elsewhere [7]).

When transaction A attempts to open an object, it may discover that the object has already been opened by a transaction B. A can decide either to back off and give B a chance to complete, or it can proceed, forcing B to abort. The policy decision is handled by a separate *Contention Manager* module.

Each time a transaction opens an object, it checks whether it has been aborted by a synchronization conflict, a process called *validation*. This check prevents an aborted transaction from wasting resources, and also ensures that each transaction has a consistent view of the transactional objects.

When a transaction opens a transactional object, it acquires a reference to a *version* of that object. If the object is opened in WRITE mode, the transaction may modify that version, and otherwise it may only observe that version.

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, the transaction simply returns a reference to the most recent 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*.

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. 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, TEMP 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 TEMP mode, releasing each element after opening its successor (similar to lock coupling). Each experiment 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 and has 1 GB of RAM. The machine was running Debian Linux and each experiment 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 experiment 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.

	Invisible		Visible	
WRITE	36.6		22.3	
READ	4.9	(13.5%)	23.9	(107.3%)
RELEASE	19.9	(54.5%)	21.2	(95.2%)

Table 1: Single-Thread Throughput

5. BENCHMARK 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, the READ and RELEASE throughput numbers are labeled with their percentages of the comparable WRITE benchmark. (For example, the invisible READ's throughput of 4.9 is 13.5% of the invisible WRITE's throughput.)

The invisible WRITE had better throughput than the visible WRITE, because when the visible WRITE opens an object, it 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 READ performed poorly because it validates each object previously open for READ each time a new object is opened. The visible READ performed slightly better than WRITE because it does not need to copy the object version. The invisible RELEASE performed better than the invisible READ because it releases objects, and once an object is released, it no longer needs to be validated.

Table 2 shows the time (in nanoseconds) for common method calls. The WRITE and "READ & TEMP" rows show the time needed to open an object in those modes, the UPGRADE row shows the time needed to upgrade from READ or TEMP mode to WRITE mode, and the RELEASE line shows the time needed to release an object opened in TEMP mode. These timings to not always mirror the benchmark throughput numbers because the visible implementation incurs all its overhead in calls to the `open()` method,

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

Table 2: Common Method Call Timings (nanoseconds)

while the invisible implementation incurs costs each time the current transaction is validated as a side-effect of other DSTM calls.

We now turn our attention from single-thread executions, where overhead dominates, to multi-thread executions, where we hope to see gains in READ or RELEASE mode due to reduced synchronization conflicts.

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.

Surprisingly, perhaps, the concurrency allowed in READ and TEMP 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 overhead because it must traverse and prune a non-trivial list of readers.

We ran a number of other experiments, including longer lists and adding additional delays ("work") to transactions. The results, omitted here for brevity, are essentially unchanged: overall, READ and TEMP modes do not enhance throughput.

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, to address the problem of increasing throughput on our four-processor platform, we devised a new SNAP mode described in the next section.

6. SNAPSHOT MODE

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

```
public TMCloneable open(SNAP)

  throws DeniedException
```

This method returns a reference to the version that would have been returned by a call to `open(READ)`. 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 `open(SNAP)`.

```
public void snapValidate(TMCloneable version)
  throws DeniedException, SnapshotException
```

The call returns normally if a call to `open(SNAP)` (or `open(READ)`) would have returned the same version reference. Otherwise, the call throws a `SnapshotException`. Throwing this exception does not abort the current transaction, allowing the transaction to retry another snapshot.

```
public TMCloneable
  snapUpgradeRead(TMCloneable version)
  throws SnapshotException, DeniedException
```

If the version argument is still current, this method opens the object in READ mode, and otherwise throws an exception (`SnapshotException`).

```
public TMCloneable snapUpgradeWrite(TMCloneable)
  throws SnapshotException, DeniedException
```

If the version argument is still current, this method opens the object in WRITE mode, and otherwise throws an exception (`SnapshotException`).

Objects opened in TEMP mode are typically used in one of the following three ways. Most commonly, an object is opened in TEMP 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.

```
Entry entry = (Entry)tmObject.open(TEMP);
...
entry.release();
```

The same effect is achieved by the following code fragment:

```
Entry entry = (Entry)tmObject.open(SNAP);
...
tmObject.snapValidate(entry);
```

The first call returns a reference to the object version that would have been returned by `open(TEMP)` (or `open(READ)`), 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 TEMP 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 TEMP mode and later upgraded to WRITE mode. The snapUpgradeWrite() method provides the same effect.

To illustrate how one might use SNAP mode, figure 1 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 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

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	36.6		35.7		32.9		29.6		24.7	
READ	4.9	(13.5%)	1.7	(4.7%)	0.6	(1.7%)	0.5	(1.8%)	0.5	(2.2%)
RELEASE	19.9	(54.5%)	8.5	(23.7%)	4.2	(12.6%)	3.8	(12.8%)	3.7	(15.1%)

Table 3: Invisible Implementation

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	22.3		23.1		21.4		20.0		17.6	
READ	23.9	(107.3%)	0.1	(0.3%)	0.2	(0.9%)	0.2	(1.1%)	0.2	(1.2%)
RELEASE	21.2	(95.2%)	0.03	(0.1%)	0.1	(0.7%)	0.2	(1.0%)	0.3	(1.6%)

Table 4: Visible Implementation

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 run-time 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 TEMP mode are shown in Tables 5 (invisible) and 6 (visible). For both visible and invisible implementations, SNAP mode has substantially higher throughput than both READ and TEMP mode. Opening an object in SNAP mode takes about 100ns, including validation. It takes about 125ns to upgrade an object opened in SNAP mode to to WRITE mode.

Even though invisible SNAP mode outperforms invisible READ and TEMP, 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.

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.

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 7), 10% (Table 8), 1% (Table 9), and 0% (Table 10). Each of the SNAP mode benchmarks had higher throughput than its WRITE counterpart, and was the only benchmark to do so.

8. CONCLUSIONS

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 TEMP modes performed on our small-scale benchmarks. While our SNAP mode implementation substantially outperforms both READ and TEMP 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.

Notice that the DSTM guarantees that every transaction, even ones that are 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. 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 two approaches remains an open topic for further research.

```
  public boolean insert(int v) {
    List newList = new List(v);
    TMObject newNode = new TMObject(newList);
    TMThread thread = (TMThread)Thread.currentThread();
    while (thread.shouldBegin()) {
      thread.beginTransaction();
      boolean result = true;
      try {
        TMObject lastNode = null;
        List lastList = null;
        TMObject prevNode = this.first;
        List prevList = (List)prevNode.openSnap();
        TMObject currNode = prevList.next;
        List currList = (List)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 = (List)currNode.openSnap();
        }
        if (currList.value == v) {
/*B*/      prevNode.snapUpgradeRead(prevList);
          result = false;
        } else {
          result = true;
/*C*/     prevList = (List)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;
  }
```

Figure 1: SNAP-mode insert method

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	36.6		35.7		32.9		29.6		24.7	
READ	4.9	(13.5%)	1.7	(4.7%)	0.6	(1.7%)	0.5	(1.8%)	0.5	(2.2%)
RELEASE	19.9	(54.5%)	8.5	(23.7%)	4.2	(12.6%)	3.8	(12.8%)	3.7	(15.1%)
SNAP	62.4	(170.7%)	16.7	(46.8%)	10.9	(33.2%)	10.2	(34.6%)	9.6	(39.0%)

Table 5: SNAP with Invisible

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	22.3		23.1		21.4		20.0		17.6	
READ	23.9	(107.3%)	0.1	(0.3%)	0.2	(0.9%)	0.2	(1.1%)	0.2	(1.2%)
RELEASE	21.2	(95.2%)	0.03	(0.1%)	0.1	(0.7%)	0.2	(1.0%)	0.3	(1.6%)
SNAP	104.8	(469.9%)	62.3	(269.6%)	56.4	(263.2%)	42.2	(210.7%)	37.8	(214.9%)

Table 6: SNAP with Visible

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	22.1		23.0		21.4		19.6		17.2	
READ	23.3	(105.6%)	0.2	(0.8%)	0.5	(2.3%)	0.9	(4.8%)	1.1	(6.6%)
RELEASE	20.8	(94.2%)	0.1	(0.4%)	0.5	(2.2%)	0.8	(4.3%)	1.1	(6.3%)
SNAP	108.4	(491.5%)	81.0	(352.1%)	72.4	(337.7%)	59.5	(302.9%)	31.7	(184.0%)

Table 7: Visible with 50% Modification

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	22.3		22.9		20.9		19.4		17.2	
READ	23.6	(106.0%)	0.5	(2.3%)	2.8	(13.4%)	3.9	(20.1%)	3.1	(18.2%)
RELEASE	21.1	(94.7%)	0.4	(1.7%)	2.4	(11.7%)	3.2	(16.7%)	3.0	(17.6%)
SNAP	109.7	(491.6%)	86.9	(379.4%)	88.1	(420.7%)	58.3	(300.7%)	19.8	(115.1%)

Table 8: Visible with 10% Modification

9. REFERENCES

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 538–547. ACM Press, 1995.

[2] Anderson and Moir. Universal constructions for large objects. In *WDAG: International Workshop on Distributed Algorithms*. 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 and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[6] M. Herlihy and J. E. 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, and 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*, pages 92–101. ACM Press, 2003.

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

[9] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 151–160. ACM Press, 1994.

[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*, pages 106–114. ACM Press, 1990.

[11] Virendra J. Marathe and 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

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	20.9		20.9		20.3		18.0		15.9	
READ	22.2	(105.9%)	5.1	(24.4%)	11.5	(56.8%)	7.8	(43.3%)	1.9	(11.8%)
RELEASE	20.8	(99.4%)	5.4	(25.7%)	3.4	(16.9%)	2.9	(16.3%)	2.7	(16.7%)
SNAP	105.0	(501.2%)	97.4	(466.0%)	98.1	(483.4%)	62.8	(348.9%)	24.6	(154.7%)

Table 9: Visible with 1% Modification

	1 Thread		4 Threads		16 Threads		32 Threads		64 Threads	
WRITE	11.5		11.8		11.0		10.2		9.0	
READ	12.4	(107.7%)	9.0	(76.1%)	4.6	(42.3%)	2.4	(23.6%)	0.5	(6.0%)
RELEASE	11.0	(95.6%)	3.2	(27.5%)	2.4	(21.6%)	2.2	(21.5%)	2.0	(22.4%)
SNAP	61.3	(531.0%)	55.6	(472.4%)	61.0	(555.9%)	67.3	(662.5%)	71.3	(790.0%)

Table 10: Visible with 0% Modification

transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

[13] Ravi Rajwar and James R Goodman. Transactional lock-free execution of lock-based programs.

[14] Martin Reiman and 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*, pages 168–177. ACM Press, 1991.

[15] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

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