Reducing Rollbacks of Transactional Memory Using Ordered Shared Locks

Ken Mizuno, Takuya Nakaike, and Toshio Nakatani

Tokyo Research Laboratory, IBM Japan, Ltd.
{kmizuno,nakaike,nakatani}@jp.ibm.com

Abstract. Transactional Memory (TM) is a concurrency control mechanism that aims to simplify concurrent programming with reasonable scalability. Programmers can simply specify the code regions that access the shared data, and then a TM system executes them as transactions. However, programmers often need to modify the application logic to achieve high scalability on TM. If there is any variable that is frequently updated in many transactions, the program does not scale well on TM.

We propose an approach that uses ordered shared locks in TM systems to improve the scalability of such programs. The ordered shared locks allow multiple transactions to update a shared variable concurrently without causing rollbacks or blocking until other transactions finish. Our approach improves the scalability of TM by applying the ordered shared locks to variables that are frequently updated in many transactions, while being accessed only once in each transaction. We implemented our approach on a software TM (STM) system for Java. In our experiments, it improved the performance of an hsqldb benchmark by 157% on 8 threads and by 45% on 16 threads compared to the original STM system.

1 Introduction

Transactional Memory (TM) [1] is a new concurrency control mechanism that aims to simplify concurrent programming with acceptable scalability. In TM, program sections that access shared variables are treated as transactions. If some transactions cause a conflict, as when one transaction updates a variable that another concurrently executing transaction has read, then the TM system rolls back and re-executes one of the transactions.

If there is a variable that is updated in many transactions, it will cause frequent conflicts and degrade the scalability of the program. Figure 1 is a sample program for a hash table. Some transactions are tagged using the keyword `atomic`. In this program, `size` is updated every time an entry is inserted or removed, and this causes frequent conflicts. Figure 2 shows a sample execution sequence of two concurrent transactions in a TM system that uses lazy conflict detection [2,3]. They read the same value from `size` and increment it. The update in each transaction is invisible to the other transaction before it is committed. If Transaction A commits successfully before Transaction B tries to commit, then B should roll back because `size` was updated by the commit of A after the read operation in B. In contrast, if `size` is eliminated, conflicts rarely occur since the accesses to `table` are scattered by the hashes of the keys. A similar situation

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 704–715, 2009.

```
public void put(Object k, Object v) {
  atomic {
    if (contains(k)) {
      updateRecord(table[hash(k)], k, v);
    } else {
      int s = ++size;
      if (s > capacity) expandTable();
      insertRecord(table[hash(k)], k, v);
    }
  }
}
public void remove(Object k) {
  atomic {
    if (contains(k)) {
      size--;
      removeRecord(table[hash(k)], k);
    }
  }
}
pubic Object get(Object k) {
  atomic {
    return getRecord(table[hash(k)], k);
  }
}
```

Fig. 1. Sample program for a hash table with a counter variable

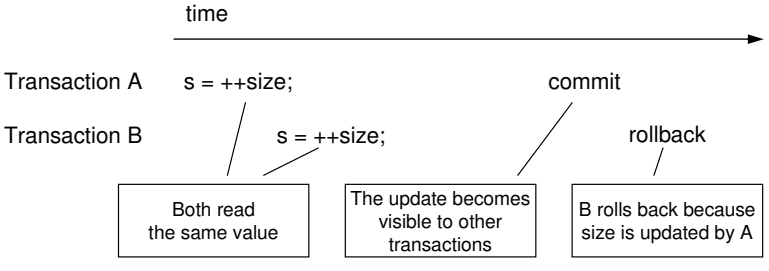

Fig. 2. Sample execution sequence of two transactions accessing hash table

occurs for the variables that hold the next ID numbers in ID number generators, variables that hold the sizes of collection objects, or variables that hold execution statistics. If a program is not tuned for TM, it often contains such variables and they prevent the scalability.

In this paper, we propose an approach for TM systems that uses ordered shared locks [4,5,6] for the variables that are tagged by the programmer. The ordered shared locks allow multiple transactions to update a shared variable concurrently without causing rollbacks or blocking until other transactions finish. If two transactions are executed in the same order as the example of Figure 2 using the ordered shared lock for `size`,

Transaction B reads the result of the increment operation in Transaction A, and both A and B successfully commit.

In this paper, we make the following contributions.

- We present an approach that uses ordered shared locks in TM.
- We explain an implementation of the approach on a software-based TM (STM) system for Java.
- We evaluate the performance of the approach using some benchmark programs.

We give a brief overview of the original ordered shared lock in Section 2. We explain our approach and an implementation of it as an STM system for Java in Section 3. We evaluated the performance of the implementation of our approach in Section 4. We review related work in Section 5 and conclude in Section 6.

2 Ordered Shared Lock

The ordered shared lock [4,5,6] is an extension of the shared/exclusive lock that allows multiple transactions to share locks even when one of the locks is a lock for write access.[1] Locks between multiple reads are shared as in the traditional shared/exclusive lock.

Figure 3 is a sample execution sequence of transactions that use ordered shared locks. Transaction B can acquire a lock on X even though Transaction A has acquired a lock on X. In this case, Transaction B observes the result of the write operation in A even though A has not committed yet. The lock on X is said to be *on hold* for Transaction B since it was acquired after Transaction A acquired the lock on X and before Transaction A releases the lock. The lock for B remains on hold until Transaction A releases the lock. When more than two transactions acquire locks for the same object concurrently, the lock for Transaction B is on hold until all of the transactions that acquired the lock before Transaction B acquired the lock release their locks.

Transactions that use ordered shared locks should satisfy the following constraints.

Acquisition Rule. If Transaction A acquires a lock on object X before Transaction B acquires a lock on object X, the operation on X in Transaction A must precede the operation on X in Transaction B.

Relinquishing Rule. A transaction may not release any lock as long as any of its locks are on hold.

The example in Figure 3 satisfies the constraints since Transaction B reads X after Transaction A wrote to X, and the release operation for the lock on X of Transaction B is deferred until A releases the lock on X.

The ordered shared lock may cause cascading rollbacks. If Transaction A is rolled back in the example of Figure 3, then Transaction B should also be rolled back because it has read the result of an operation from Transaction A. Therefore, the ordered shared lock is not suitable for transactions that cause rollbacks frequently because it causes frequent cascading rollbacks.

[1] We describe only the most permissive protocol of the ordered shared locks that allows ordered sharing both between read and write, and between write and write.

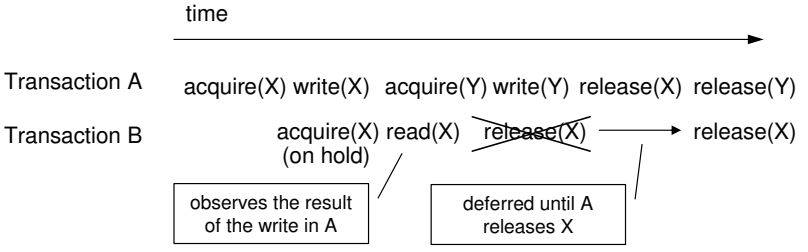

Fig. 3. Sample execution sequence of two transactions with ordered shared locks

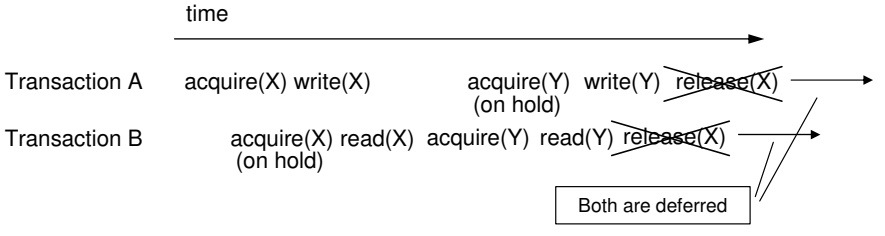

Fig. 4. Sample execution sequence that causes deadlock

The ordered shared lock approach may also cause a deadlock. However, deadlock can be detected and recovered from by using a traditional deadlock detection and recovery technique. A simple way to recover from a deadlock is to roll back one of the deadlocked transactions. Figure 4 is a sample execution sequence that causes a deadlock. In this example, both Transaction A and B are unable to commit before the other transaction releases its locks.

3 Our Approach

In this section we introduce our approach that uses the ordered shared locks for TM. We first explain how to integrate the ordered shared locks in the TM. After that, we describe the design of our STM system for Java, and then we explain the implementation of our approach.

3.1 Integrating Ordered Shared Locks in TM

In our approach, programmers specify some variables as *uncooperative variables*, and then the TM system uses ordered shared locks for them. This prevents the rollbacks due to conflicts on the uncooperative variables and improves the scalability.

Programmers should specify the variables with these properties as the uncooperative variables:

- They are frequently updated in many transactions.
- They are accessed at most once in each transaction.
- Each transaction accesses at most one uncooperative variable.

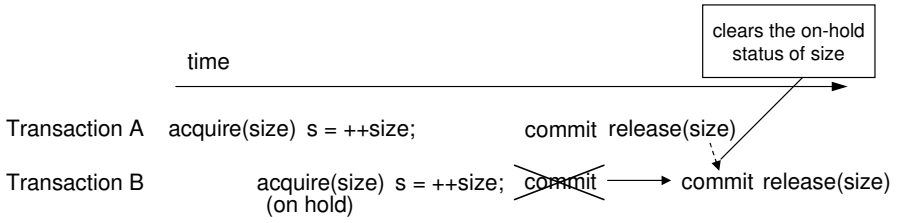

Fig. 5. Sample execution sequence for hash table using our approach

The variables that satisfy the first property cause frequent conflicts in the standard TM systems. The ordered shared locks are suitable for such variables because they prevent the rollbacks due to the conflicts. The other two properties avoid rollbacks. If one of these two properties does not hold, rollbacks can occur due to situations such as illustrated in Figure 4. We think these two properties are acceptable because our motivation is to remove the rollbacks caused by a few uncooperative variables.

In order to integrate the ordered shared locks into the TM, we divided the relinquishing rule described in Section 2 into two rules. Now the constraints of the ordered shared locks can be interpreted as:

Acquisition Rule. If Transaction A acquires a lock on object X before Transaction B acquires a lock on object X, then the operation on X in Transaction A must precede the operation on X in Transaction B.

Committing Rule. A transaction may not commit as long as any of its locks is on hold.

Relinquishing Rule. A transaction may not release any lock before it commits or aborts.

We say a transaction is *ready to commit* when it reaches the end of the transaction and none of its locks is on hold. The conflict detection for the variables other than the uncooperative variables is included in the commit operation when we are using the lazy conflict detection. It should be executed after the transaction becomes ready to commit, although it can be executed speculatively before the transaction becomes ready to commit to detect conflicts and abort more quickly. Figure 5 shows a sample execution sequence for the example in Figure 1 using our approach. Now the two transactions successfully commit and `size` is incremented twice.

A transaction with ordered shared locks will roll back in these situations:

- When a conflict occurs in variables that do not use ordered shared locks.
- When a deadlock is detected in the transaction.
- When a transaction that this transaction is waiting for is rolled back. (A cascading rollback.)

The first situation is the same as the original TM. The main benefit of the ordered shared lock is that the uncooperative variables do not cause any rollback in this situation. The second and the third situations are caused by the ordered shared lock.

Our approach is strongly serializable [7], which means that the transactions are serializable in the order of the commit operations, as long as it is applied to a TM system that is also strongly serializable. To prove this, we need to check this property: The

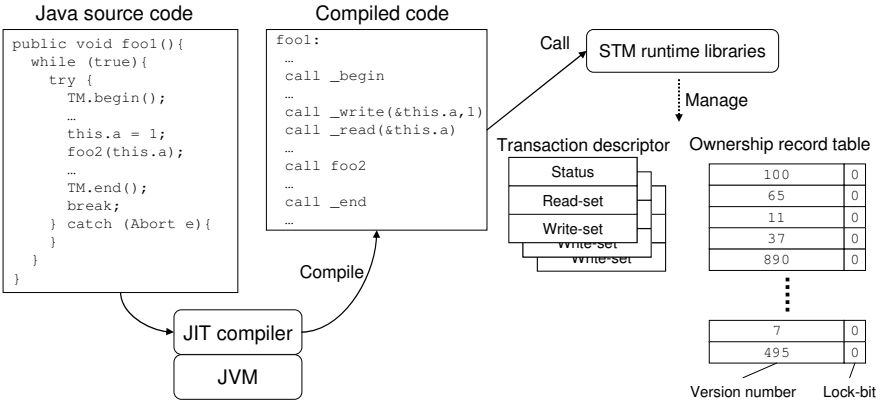

Fig. 6. Overview of Our STM System

commit operation of Transaction A must finish before the beginning of the commit operation of Transaction B if Transaction B reads the value of the variable that Transaction A wrote. If the variable is an uncooperative variable, this property is guaranteed by the three rules of this section. If the variable is not an uncooperative variable, this property is satisfied as long as the TM system is strongly serializable. Most TM systems that use lazy conflict detection guarantees this property by executing the validation and commit atomically.

3.2 Design of a Java STM

Figure 6 shows an overview of our Java STM system. It consists of an STM runtime library that is based on an IBM STM for C/C++ [8], and a JIT compiler that generates code that calls the runtime library.

The JIT compiler generates two versions of JIT-compiled code for each method invoked in a transaction: a non-transactional version and a transactional version [9,10]. In the transactional version, the `stm_read` or `stm_write` function of the runtime library is called when the code accesses the heap area. Transactions are specified using special methods named `TM.begin()` and `TM.end()`.

We used lazy conflict detection and lazy versioning [9,11]. With lazy versioning, the results of the write operations in a transaction are buffered to a transaction-local data structure. The results are reflected to the shared variables when the transaction successfully commits, but they are discarded if the transaction rolls back. The alternative is eager versioning [10,12], which stores the result of a write operation directly in the shared variable, and copies the old values to a transaction-local data structure in order to write them back to the shared variable if the transaction rolls back.

To simplify the memory management of the STM runtime, we do not require the STM operations to be non-blocking, but instead we use locking to access the metadata [12]. We devised a simple mechanism to avoid starvation in which a long-running transaction is repeatedly aborted by short running transactions. When the number of

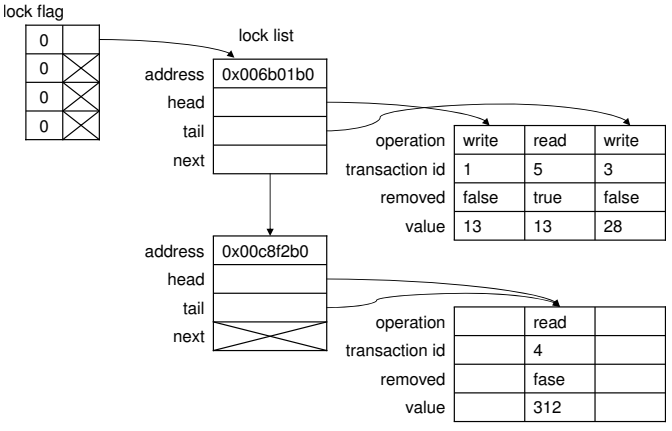

Fig. 7. Data Structure to Control Ordered Shared Lock

rollbacks of a transaction exceeds a threshold, the transaction acquires a lock to prevent the other transactions from proceeding to the commit process.

Our Java STM system provides weak atomicity, and thus it does not guarantee the safety of publication or privatization [11,13].

When garbage collection (GC) occurs, we abort all of the transactions. This is because GC may move objects and invalidate the metadata of the transactions. We use periodic validations to avoid infinite loops due to inconsistent reads. When an inconsistent read causes an exception such as a "divide by zero" exception, the TM system catches the exception and executes the usual cleanup operations in `TM.end()`. If the exception is caused by an inconsistent read, then the validation fails and the transaction rolls back.

3.3 Implementation of Our Approach

In this subsection we describe our implementation of the ordered shared lock in our Java STM system. This implementation is based on the description in [5,6].

Figure 7 shows the data structure used to control the ordered shared locks. The system provides a *lock list* for each variable that uses the ordered shared locks. The lock list is implemented using an array and the array contains transactions which hold the ordered shared lock in the order of lock acquisition. The list also contains the types of the operations (read or write), flags that indicate whether the entry was removed due to a rollback, and the values that the transactions read or wrote. A write operation stores the value in the lock entry and it is reflected into the shared variable in the commit operation. A read operation reads the value from the preceding lock entry if it exists. The lock lists are stored in a hash table using the addresses of the variables as keys. Chaining is used when a hash collision occurs.

Each record of the hash map has a *lock flag* field. In order to satisfy the acquisition rule, the lock acquisition and a read or write operation are executed atomically using the lock flag: the transaction sets the lock flag, appends an entry to the lock list, executes

the operation, and then clears the lock flag. We also implemented a method that adds a specified value to a specified variable atomically for use in increment or += operations. If such operations are executed as a combination of read and write operations, another transaction may access the variable in the middle of the operations. This may cause a deadlock.

In order to implement the commit rule, each transaction has a variable named `wait_count`, which indicates the number of locks that are on hold. The variable is incremented when the transaction acquires a lock that will be on hold. When a transaction releases a lock in a commit or rollback operation, it decrements the `wait_count` for the transaction whose lock released the on-hold status. When a transaction reaches its end, it waits until `wait_count` becomes zero before executing the validation. To detect a deadlock, the transaction will be rolled back if it times out before `wait_count` becomes zero. This approach has little overhead for the deadlock detection, but it causes a significant slowdown when a deadlock occurs. We chose this approach because the properties of the uncooperative variables described in Section 3.1 avoid deadlock and the deadlock detection mechanism is rarely triggered.

4 Experiments

This section describes the benchmark experiments that we watched to analyze the performance of our approach.

4.1 Benchmark Programs

We tested three programs: a small program intensively accessing HashMap, the SPECjbb2005 [14] benchmark, and the hsqldb benchmark [15] in the DaCapo benchmarks [16]. We executed them in the synchronized mode, the TM mode without ordered shared locks, and the TM mode with ordered shared locks. In the synchronized mode, each critical section is guarded using Java's synchronized block. In the TM modes, each critical section is executed as a transaction. We used 10-millisecond intervals for the periodic read-set validations in the TM modes. We executed the benchmarks on Red Hat Enterprise Linux Server release 5.1 on a 16-way POWER6 machine. Our STM system for Java is based on the IBM 64-bit JVM version 1.5.0. We used 8 GB of Java heap to minimize the GC events in our experiments.

HashMap. We implemented a benchmark program for java.util.HashMap. In our benchmark, a HashMap object is first filled with 1,000 elements and then each thread calls one of the `get`, `put`, or `remove` methods in a critical section. The critical section is guarded by synchronizing the HashMap object, but it is executed as a transaction in the TM modes. When a thread calls the `put` method, it always puts a new entry rather than updating an existing entry. The transactions are 95% calls to `get`, 2.5% calls to `put`, and 2.5% calls to `remove`.

For the TM mode with ordered shared locks, we specified as an uncooperative variable the variable named `elementCount`, which holds the number of elements and which is updated in the `put` and `remove` methods. In the `put` method, this variable is also used to check if it exceeds the threshold to resize the table. We set the initial size of the HashMap large enough to prevent any resize operations during an experimental run.

SPECjbb2005. We used a modified version of SPECjbb2005. The original SPECjbb-2005 prepares a thread-local data structure called a *warehouse* to avoid lock contentions. We modified it to share a single warehouse for all of the threads, similar to the implementation of Chung et al. [17]. Each SPECjbb2005 transaction (such as *new order* or *order status*) is executed in a critical section. However, the `JBBDataStorage`[2] transaction that traverses multiple tables is divided into multiple critical sections, each of which traverses one table. The critical sections are guarded by synchronizing the warehouse object in the synchronized mode.

For the TM mode with ordered shared locks, we specified as an uncooperative variable the variable named `historyCount`, which is always updated in a SPECjbb2005 transaction named `DeliveryTransaction`.

Hsqldb. We used the hsqldb benchmark in the DaCapo benchmarks. In this benchmark, the database tables are placed in memory, and an SQL statement is executed in a critical section. Hsqldb is not designed to execute SQL statements concurrently, and there is some shared data that does not actually need to be shared. We modified the code to avoid rollbacks caused by the shared data:

- There is a counter variable that tracks the number of created objects. It is used to trigger garbage collection at a specified threshold. We removed the counter since it provides little benefit in a multi-threaded environment.
- Some hash tables are used to store data that is essentially thread-local. This is implemented as shared data, which we modified to use thread-local hash tables.
- Hsqldb uses object pools to increase the single-thread performance. We disabled them because they decrease the multi-thread performance.

For the TM mode with ordered shared locks, we specified as an uncooperative variable the variable named `currValue` in the `NumberSequence` class. This variable is used to generate the ID numbers of the database records.

4.2 Result

Figure 8 shows the relative throughput of the benchmarks compared to the single-thread performance of the synchronized mode. Figure 9 shows the abort ratio, the ratio of aborted transactions to all of the transactions. The label "OS" denotes the ordered shared lock in the figures.

In all of the benchmarks, the results of the TM modes are worse than that of the synchronized mode when the number of threads is small. This is because of the overhead of the software-based TM.

In the HashMap benchmark, our approach shows a low abort ratio because we eliminated the conflicts on the uncooperative variable. However, the throughput was degraded because the path length of the commit operation is relatively long in the kernel loop of this benchmark. In our approach, the commit operations in transactions that access the same uncooperative variable are serialized because no transaction can start a commit operation until the preceding transactions release the ordered shared lock. As

[2] Implemented using java.util.TreeMap.

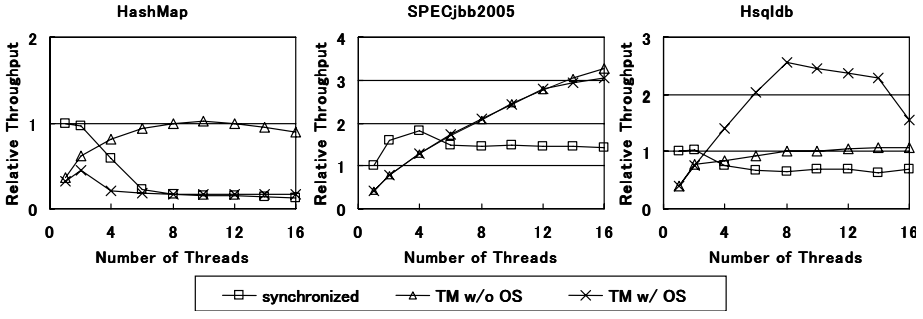

Fig. 8. Relative throughput compared to the single-thread performance of the synchronized mode (higher is better)

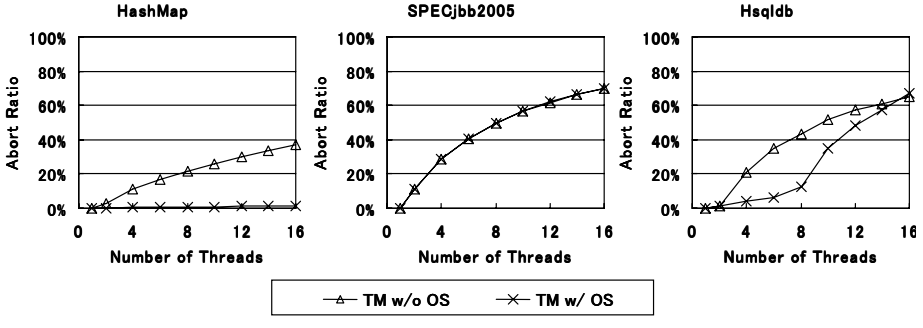

Fig. 9. Abort ratio (lower is better)

the number of rollbacks is reduced, it will be more beneficial when HashMap is used in larger transactions.

Our approach did not improve the performance of the modified SPECjbb2005 benchmark. In this benchmark, reducing the conflicts on one uncooperative variable does not reduce the abort ratio. There are other variables that cause conflicts but which are not suitable for the ordered shared lock.

Finally, the results for the hsqldb benchmark were greatly improved. The throughput increased by 157% on 8 threads and by 45% on 16 threads compared to the TM mode without ordered shared locks. The uncooperative variable was the primary cause of the conflicts, thus the ordered shared lock had provided a great benefit. On the down side, the abort ratio was increased when there were more than 8 threads. This seems to be caused by cascading rollbacks. When the abort ratio is very high, our approach causes further rollbacks due to the cascading rollbacks, which reduces the benefit of our approach.

5 Related Work

In order to reduce the costs of rollbacks, various kinds of nested transactions have been introduced. However, none of them are beneficial for the variables whose values affect

the execution of other parts of the transaction. For example, with closed nesting [18], the conflicts in an inner transaction are detected at the end of the inner transaction, so only the inner transaction is rolled back. Unfortunately, this provides no benefit when the conflicts occur between the end of the inner transaction and the end of the outer transaction. Open nesting [18] solves this problem, but the results of the inner transaction must not affect the other parts of the outer transaction. It is not applicable to transactions such as `put` in Figure 1, which use the results of increment operations. With abstract nesting [19], only an inner transaction is rolled back when conflicts caused by the inner transaction are detected at the end of the outer transaction. However, this provides no benefits if the result of the re-execution of the inner transaction affects other parts of the outer transaction.

The correctness of TM was discussed in [20]. This work introduces a correctness criterion called *opacity* that captures the inconsistency of reads. We did not use this approach since we can safely detect the exceptions caused by inconsistent reads in Java.

The ordered shared lock was proposed by Agrawal et al. [4,5,6]. The analysis of [5] showed that the ordered shared lock improves the performance for database workloads with large numbers of data contentions. The ordered shared lock has also been applied to real-time databases in [6] to eliminate blocking.

6 Conclusion

We proposed an approach that uses the ordered shared lock in TM systems. In this approach, frequently conflicting variables that are accessed only once in each transaction are controlled by ordered shared locks and the other variables are controlled by a normal conflict detection mechanism of the TM. This approach improves the scalability because we can reduce the conflicts caused by the uncooperative variables. In our experiments, we improved the performance of an hsqldb benchmark by 157% on 8 threads and by 45% on 16 threads compared to the original STM system.

References

1. Herlihy, M., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: 20th Annual International Symposium on Computer Architecture, pp. 289–300. ACM Press, New York (1993)
2. Hammond, L., Wong, V., Chen, M.K., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional Memory Coherence and Consistency. In: 31st Annual International Symposium on Computer Architecture, pp. 102–113. ACM Press, New York (2004)
3. Fraser, K., Harris, T.: Concurrent Programming Without Locks. ACM Trans. Comput. Syst. 25(2), 1–61 (2007)
4. Agrawal, D., Abbadi, A.E.: Locks with Constrained Sharing. In: 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 85–93. ACM Press, New York (1990)
5. Agrawal, D., Abbadi, A.E., Lang, A.E.: The Performance of Protocols Based on Locks with Ordered Sharing. IEEE Trans. Knowl. Data Eng. 6(5), 805–818 (1994)

6. Agrawal, D., Abbadi, A.E., Jeffers, R., Lin, L.: Ordered Shared Locks for Real-Time Databases. VLDB J. 4(1), 87–126 (1995)
7. Breitbart, Y., Garcia-Molina, H., Silberschatz, A.: Overview of Multidatabase Transaction Management. VLDB J. 1(2), 181–239 (1992)
8. XL C/C++ for Transactional Memory for AIX, `http://www.alphaworks.ibm.com/tech/xlcstm/`
9. Harris, T., Fraser, K.: Language Support for Lightweight Transactions. In: 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 388–402. ACM Press, New York (2003)
10. Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and Runtime Support for Efficient Software Transactional Memory. In: ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, pp. 26–37. ACM Press, New York (2006)
11. Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing Isolation and Ordering in STM. In: ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pp. 78–88. ACM Press, New York (2007)
12. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In: 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 187–197. ACM Press, New York (2006)
13. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R.L., Saha, B., Welc, A.: Practical Weak-Atomicity Semantics for Java STM. In: 20th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 314–325. ACM Press, New York (2008)
14. SPECjbb2005, `http://www.spec.org/jbb2005/`
15. hsqldb, `http://hsqldb.org/`
16. The DaCapo benchmark suite, `http://dacapobench.org/`
17. Chung, J., Minh, C.C., Carlstrom, B.D., Kozyrakis, C.: Parallelizing SPECjbb2000 with Transactional Memory. In: Workshop on Transactional Memory Workloads (2006)
18. Moss, J.E.B., Hosking, A.L.: Nested Transactional Memory: Model and Architecture Sketches. Sci. Comput. Program. 63(2), 186–201 (2006)
19. Harris, T., Stipic, S.: Abstract Nested Transactions. In: The 2nd ACM SIGPLAN Workshop on Transactional Computing, TRANSACT 2007 (2007)
20. Guerraoui, R., Kapalka, M.: On the Correctness of Transactional Memory. In: 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 175–184. ACM, New York (2008)