Exploring Data Reusing of Failed Transaction

Shaogang Wang, Dan Wu, Xiaodong Yang, and Zhengbin Pang

School of Computer, National University of Defense Technology
Changsha, Hunan, 410073 China
wshaogang@nudt.edu.cn

Abstract. Transactional Memory (TM) has been the promising parallel programming technique to relieve the tedious work of synchronizing shared object using lock mechanism. Transaction execution required to be atomic and isolated relative to the whole system. The transaction fails if found violated access to the shared object from other transaction, and it will be re-executed till finally commit successfully; currently, most TM systems are required to restore shared memory's state before re-execution, this cleanup cost and the shared object's opening cost greatly hurdle system's performance.

In this paper, we propose a new general transaction iteration's data reusing (TItDR) method which reuses the opened object of failed transaction in the following re-execution. The obvious advantage is that it greatly simplify the opening process if it has been opened in previous failed transaction and most of the cleanup work are no longer needed. TItDR leaves opened object in pseudo-active state and restart the transaction, We talk about conflicts resolution, validation, commit/abort processing problem along with our data reusing method and show that TItDR will not incur more conflicts and more overhead for validation or commit. Both currently proposed software transactional memory (STM) systems and hardware systems (HTM) have much potential data reusing.

Our test result is based on STM implementation, which shows 40% performance improvement on average.

Keywords: transactional memory, data reusing, TItDR.

1 Introduction

Recent research has showed that transactional memory has been the promising parallel programming technique. The proposed TM systems (RSTM[1], UTM[2], logTM[3], TCC[4]) must provide atomicity and isolation for transactions. If concurrent executing transactions find that they can not both successfully commit because of conflicting shared memory access, one transaction must be chosen to re-execute. This abort handling is the import part of TM systems as it greatly affects the overall TM system's performance. Next we summarize the well-known published TM systems, focusing on processing if transaction fails. Analysis shows that there exists potential data reusing of the aborted work for failed transaction execution.

As seen from the currently TM systems(LogTM[3,5,6], TCC[4,7], UTM[2,8], RSTM[1,9]), to abort a transaction, we must do the tedious compensating work to guarantee the transaction's isolation property[10,11,12,1,3]. The burden of memory system doing the restore operation may be greater than the cost of opening the shared object. Because this needs to read and write a large set of dispersed data simultaneously. The retry on failed (i.e. transaction iteration) execution method will reopen the same object with very high chance, the close and then open process is necessary for the conflicting object to ensure consistency. Yet with our experience, other non-conflicting shared object, the reopen process usually does the same thing as reconstruction the metadata, allocating memory space, and updating bookkeeping information etc.

TM system detects conflicts when concurrent transactions visiting the same object and at least one is write[9], reducing transaction conflicts chance can greatly improve the whole system performance. Currently proposed conflicts resolution technique deals with this problem by letting transaction wait for object to be released or back off some time before retry transaction. The principle of reusing data should avoiding introducing more conflicts. Our paper shows that by taking special management, TM system can get this win-win situation.

In this paper, we proposed a common method to reusing transactional object on transaction aborts, which called TItDR. As far as I know, this is the first research on exploring data reusing across transaction iterations. Our contribution includes:

– We proposed a new method called TItDR which exploring the data reusing across transaction iteration.
– We show that current TM system can support our TItDR without redesign from ground.
– We give a hardware framework to support TItDR. Currently proposed transactional protocol can be enhanced to benefit from the data reusing.

2 Basic Idea

In this section, we discuss basic idea of TItDR, and ignore some implementation details which may be different for STM or HTM. In this paper, we call the repeated retry of the transaction until successfully as *transaction iteration*, so one transaction's execution is composed of several iterations with the last iteration is successful; current running iteration is called *active* iteration; the iterations that failed before active iteration is called *obsolete* iteration; the object which has been opened in the active iteration is in *active state*, the object opened in the obsolete iterations but not opened in active iteration is in *obsolete state*. Overall TItDR improved TM system with the following idea:

1. Binding a number called *itnumber* to each transaction iteration, on abort, increase the itnumber and reset itnumber if commits successfully. When shared object is opened, TItDR saves current itnumber together with the object. So by comparing the itnumber with the active iteration's itnumber, we can decide if the object is in obsolete state.

2. Thread maintains a local list of opened objects; on opening shared object, add the object together with current iteration's itnumber to the list. Most TM systems maintain the opened list for rolling back, so we may only a mirror modification. If transaction aborts, we keep the opened list through which TM system keeps information about the obsolete object, and do not try to restore the state of transactional opened object. In the next iteration, if transaction opens an obsolete object, the cost can be greatly reduced because we can reuse the obsolete object. For opening for read an obsolete object, we can use directly use the data if validation is successful, and for opening for write, previous iteration's write data maybe incorrect, so the value should be discarded, but we can reuse the metadata to avoid reconstructing.
3. Other transaction's conflicts with opened object now has two types, conflicts with active objects and conflicts with obsolete objects. Our method keeps the obsolete in the "pseudo-active" state that will enlarge the object's open time. That will introduce more conflicts between transactions. So contention manager should take a compromised decision between these two folds. With our experience, always aborting the obsolete object will not bring more conflicts than current TM systems; more, the obsolete object may not be opened in the re-execution if the execution path is different; so aborting the obsolete object will not introduce some unnecessary conflicts;
4. On reopening object in the re-execution, if object is in obsolete state and the object has not been aborted by other transaction, thread can reuse this object without performing the reopening process.

Our method's primary advantage is it greatly reduces the work needed on transaction abort, because we no longer need to restore memory state before transaction restarts. A second advantage is we reduce the cost of reopening process by reusing obsolete object in active transaction iteration. The reusing includes data value reuse, memory space reuse and data structure reuse.

On read operation, value first read to a temporal local place and uses this value in the remainder transaction, transaction commits failed if object's curren has been updated by other transaction during the transaction. TItDR improves the read operation with transaction's itnumber which will increase by 1 on restarted, if current transaction fails, most current TM systems will discard the read list and temporal read value and start a fresh read operation in the next iteration. Yet we can simply keep the read set and value in the next iteration, on opening an object, if this object is in the obsolete read set(object's itnumber will be less than active iteration's itnumber), we can reuse the value if validation successful. On write operation, new value can be directly updated to the temporal location if previous iteration has opened for write. Our basic idea is simple, we believe that the failed transaction is not having nothing to gain.

As we study from current proposed TM systems, we think that it is feasible to incorporate with our method to have data reusing benefit. In the next section, we give one implementation example based on RSTM and show some problem that is brought with data reusing.

3 Example Software Implementation

STM system usually construct through software library or language extension which uses complex metadata organization[1], in this section, we give the detailed optimization of RSTM to implement our TItDR methods. RSTM is a non-blocking STM system implemented as C++ class library. RSTM support visible/invisible transactional object read, eager/lazy transactional object write. In RSTM, every transactional object is accessed through ObjectHeader, which points to the current version of the object. The ObjectHeader contains the visiting information from transactions; The Transaction Descriptor referenced through an object's header determines the transaction's state. If the transaction commits, then NewDataObject is the current version of the object. If the transaction aborts, then OldDataObject restored to the current version. If the transaction is active, no other transaction can read or write the object without aborting the transaction. We will ignore detailed information in this paper and only gives the optimization of RSTM, which is referred by RSTM_datareuse.

We redefine transaction's ABORTED state, which means the time between current iteration is aborted and next iteration starts. RSTM_datareuse adds itnumber to transaction descriptor to holds transaction's iteration information; ObjectHeader uses the second low-bit of NewData as obsolete flag for eager write, for every entry in the explicit list of visible reader, adds one bit obsolete flag for visible reader. The obsolete flag indicates that object is in obsolete state, which is opened in previous iteration, but has not opened in current iteration.

RSTM uses bookkeeping lists (invsibleReadList, visibleReadList, eagerWriteList, lazyWriteList) to hold currently opened object, In addition RSTM_datareuse adds the list entry with itnumber field which hold current transaction iteration's itnumber.

When transaction initially starts, the bookkeeping lists are empty and the itnumber reset to 1. On read operation, RSTM_datareuse adds the object to thread local invisibleReadList or visibleReadList based on reading type. In case of visible read, mark the corresponding read flag in the ObjectHeader. The read

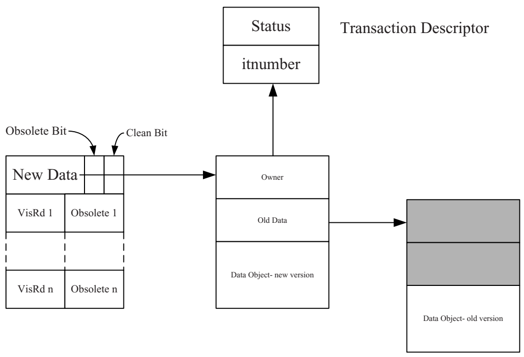

Fig. 1. metadata used to implement TItDR

flag together with the obsolete flag cleared indicating that the read object is in active state. For write operation, RSTM allocate a cloned object to hold new value. The ObjectHeader's owner state with obsolete cleared indicating the object is in write active state. If the transaction is aborted, for visible read and eager write, iterates the list and marks the active object's corresponding obsolete flag. So other transaction may only need to check the obsolete bit to see whether the object is in active state or obsolete state. RSTM_datareuse does not drop the bookkeeping list or free new allocated memory; this has performance benefit as avoids rebuilding the metadata on transaction's re-execution. As comparing with RSTM, RSTM_datareuse's abort processing is really simple.

If transaction is aborted and restarted, now the bookkeeping lists holds opened object in previous iterations. The reopening process makes some difference, first checks if the object is in the bookkeeping list, if found and the itnumber is less than current transaction itnumber, the opened object needs validation to see whether the object is still valid, if validate successfully, clear ObjectHeader's obsolete flag, and now the object is in current transaction's active state. For write operation, because RSTM_datareuse does not free the cloned object, active transaction's cloned object will reuse this memory space. RSTM_datareuse incurs a bit of lookup and validation for open cost, this cost is neglectable because the system needs periodically validating to ensure opened object is still in valid[13]. A pseudo-code for open_RW operation is as follows:

On transaction commit successfully, RSTM_datareuse only update memory of the active objects (i.e. bookkeeping list entry's itnumber equals active iteration's itnumber). There may be obsolete objects in bookkeeping list when transaction commit successfully, this is due to execution path is different between committed iteration and previous iteration. For these objects, we need to restore their previous value before transaction.

The object's obsolete state divides conflicts into two types: transaction conflicts with active object or obsolete object, RSTM_datareuse always abort the obsolete object for several reasons. First, aborting obsolete object's cost is small for it does not need to abort current active transaction execution. Second, releasing obsolete object makes object's open time shorter which allows more transaction parallelism. Third, transaction's open set may be different between iterations if the transaction has branches and the condition is based on the shared object's return value. So always aborting the obsolete object will avoid some false conflicts. The contention management policies can be used with no modification for only conflict with active objects is resolved by CM (Contention Manager) and the introduced obsolete state can be ignored by CM.

Modern STM systems incrementally validate opened object to test whether the execution is valid. RSTM opened set is maintained in transaction's bookkeeping lists, RSTM_datareuse only needs to validate the active object in the list and ignores the obsolete object, because if the obsolete object will be reopened, the opening process includes the validation operation. So although RSTM_datareuse does not drop obsolete objects, its validation cost will not be greater than RSTM.

4 The Hardware Approach

Hardware transactional memory is a hardware system that supports implementing nondurable ACI properties for threads manipulating shared data. A very natural way of implementing HTM is enhancing cache coherence protocol to support transactional processing. LogTM supports eviction of transactional accessed cache lines during a transaction by retaining ownership of the cache line. The cache coherence protocol's directory state is in sticky state when an active transaction's opened object is written back while the ownership is still reserved by transaction. In this way transaction that conflicts with the sticky object can be detected by forwarding the request to the owner if the owner is in transaction mode. A second feature is using software log to restore memory state on abortion. To implement our data reusing idea, we should enhance logTM's protocol with some extensions similar to our software approach.

Transactional object has two copies (*active* copy and *shadow* copy) in cache, the active copy stores current value and shadow copy store backup data. In this way, we no longer need the undo_log maintained by processor, for the backuped value is stored in shadow copy. The space requirement is the mainly cost, yet we think it is acceptable as we can enhance the multi-level cache to support our requirement. Another method to reduce the cost is to use the similar method used in operating system when mapping virtual memory to cache entry mapping. If memory first opened in transaction, update the active and shadow copy with object's current data. The following update to the object will be written to the active copy, if transaction commits successfully, update the memory system with the active copy. On failure, mark active copy as obsolete, if obsolete object is visited, use the shadow copy value.

Transaction's iteration number needs to be hold in processor. Cache and directory maintain iteration number information for every opened object. On transaction fails, processor increase its iteration number and mark opened shared objects as obsolete. On reopening object, the validation processor is really simple, it only needs to see whether the cache block is still valid in cache.

Undo_log: only active transaction's evicted cache object is written into the undo log. If transaction failed and the undo_log is not empty, we should replay the undo_log to restore memory state. The undo_log is managed by software, which dealing the case hardware cache overflows.

Conflicts detection is through the cache coherence protocol. When processor get intervention message which visit the obsolete object in his cache, the processor should forwarding the old data and need not abort current transaction. Another tricky is when another processor conflicts with out-of-cache active objects (i.e. objects in the undo_log), in this situation, simply send NACK message to abort it.

The hardware approach does not need to replay the undo_log to restore memory state if transaction fails. This cost is much greater than the RSTM, because logTM write the previous back to memory, while RSTM simply modify the object's header to point back original data as it keeps both active and backup data in memory. On reopening objects it reuses the shadow copy so saves memory visiting cost. Currently we are working on the test environment to give our reusing detailed test result.

5 Test Results and Analysis

In this section, we give our test result of data reusing in transactional memory, we implement TItDR based on RSTM2 as shown in section 3. We test benchmarks on a 2-processor blade server with Intel Xeon 2.3GHz, 4core processor. We compiled both our implementation and RSTM with gcc3.4.4 with O3 optimization level. We tested for a period of 10 seconds for each benchmark, varying the thread number from 1 to 8. Results were averaged over a set of 3 test runs and all experiments use the Polka contention manager.

We use the same benchmark with RSTM2[1] which includes: shared counter; linked list; hash table; LFUCache and random graph.

5.1 Total Transaction Throughput

Throughput comparison was given by the total finished transactions during 10 seconds; we give both eager and lazy write type benchmark results.

For the shared counter benchmark, we get the best speedup compared with RSTM2, this is due to that all threads want to increase the same shared variable, which can not be accessed parallel. All the thread must line up to access the counter, so with threads number increase, both our method and RSTM does not increase the total transactions throughput. Yet for RSTM's eager write type, with thread number increase, we got decrease total committed transactions, for it needs more work to contention management, restoring. For the same reason, Eager write type with data reusing will not suffer this problems. Another reason for we get the best speedup is that the counter benchmark is dominated by write, with no read operation. For software approach TM, the reusing for write will save more work than reusing for read. This is due to that we relieve the memory burden of reclaim and reallocate memory space for speculative writes. For cache coherence based HTM systems, it is another case; hardware can easily get his obsolete read object by checking cache status, and return current value.

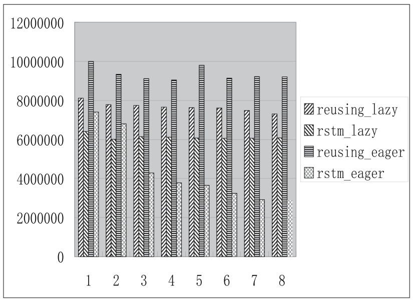

Fig. 2. Throughput of shared counter

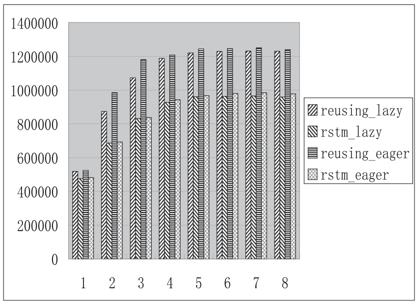

Fig. 3. Throughput of linked list

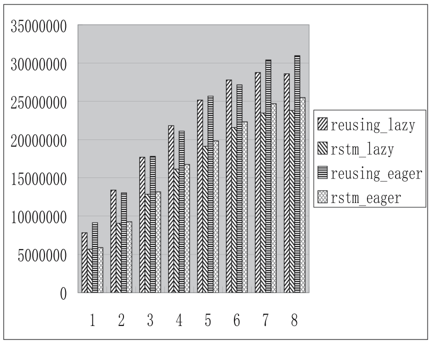

Fig. 4. Throughput of hash table

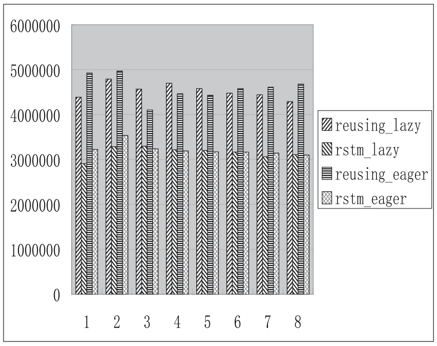

Fig. 5. Throughput of LFUCache

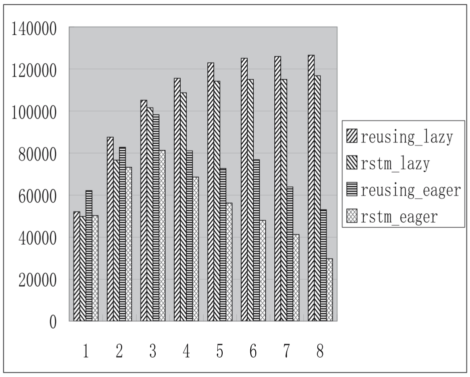

Fig. 6. Throuput of random graph tests

For hash table and random graph benchmarks, we get continued increased throughput as threads number increase, while RSTM's RandomGraph get decreased throughput, it is because that as thread continues add vertex to the graph, the graph will get large, so every time we want to locate a vertex, it must traverse a large number of vertexes before getting to the vertex, this will increase transaction's read and write set and the transaction will getting large.

Table 1. the result is got from running benchmark with 8 threads and uses invisible read, lazy write acquire rule. The number is given on thread average. Validation success on write means that the object has not been updated by other thread. Validation success on read means that our read value is still valid.

Benchmark	W_times	V_success	V_failed	R_times	V_success	V_Failed
Counter	1031806	74566	162	0	0	0
LinkedList	44119	50	1683	12854531	4287378	160397
HashTable	1151057	761	2454	5755649	3920	16814
LFUCache	1218634	134212	3795	115610	351	10390
RandomGraph	123708	183	9406	9314085	2210817	17373

This large transaction's aborting cost is the primary reason for the decreasing performance. With our method, test results show good scalability. Our method's cost of aborting will not change with different transaction size.

TItDR has better performance speedup for eager write type, e.g. the counter benchmark gets 35% to 2.2 times performance increase for eager, and for lazy, we got 25% enhancement. Totally, we got the average performance speedup of 41.4% for all benchmarks.

5.2 Potential of Data Reusing

To see the potential data reusing there exists in transactional memory systems, we count the times of open operation, and the times we can find data in failed transaction's open set, with the times that the data is valid.

For counter and LFUCache benchmark, the benchmark is dominated by write operation, so write conflicts may occur very common, this servers two folds effect, first, it will regularly make other transaction's read not valid, so to reuse read data, there is more chances that validation is failed. Second, the write may have more chances that validation is success. The LinkedList, HashTable and RandomGraph benchmarks are another case; the read operation has more chances to find that the aborted transaction's read value is still valid.

6 Conclusion and Future Work

We believe that TItDR is very attractive for TM systems have great potential data reusing. The TItDR method greatly reduced the work to abort a transaction and will accelerate the reopening process. From my experience, hardware based data reusing is more attractive than software, for it can easy get the real data reusing which can get value from cache. We have worked mainly on the software approach. Yet the software approach should be optimized to efficiently support currently proposed TM systems or rebuild TM system from ground with data reusing in mind. We have not explored how to efficiently reuse data in the nested transaction environment. Further studying cache coherence based transactional memory with data reusing support includes protocol verification, implementation and performance test.

References

1. Marathe, V.J., Spear, M.F., Heriot, C., Acharya, A., Eisenstat, D.: Scherer III, W.N., Scott, M.L.: Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester (2006)
2. Chuang, W., Narayanasamy, S., Venkatesh, G., Sampson, J., Van Biesbrouck, M., Pokam, G., Calder, B., Colavin, O.: Unbounded page-based transactional memory. In: ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, San Jose, California, USA, pp. 347–358. ACM Press, New York, NY, USA (2006)

3. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: Logtm: Log-based transactional memory. In: Proceedings of the 12th International Symposium on High-Performance Computer Architecture, pp. 254–265 (2006)
4. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. SIGARCH Comput. Archit. News 32(2), 102 (2004)
5. Liblit, B.: An operational semantics for LogTM. Technical Report, University of Wisconsin–Madison (2006) Version 1.0 (1571)
6. Moore, K.E.: Thread-level transactional memory. In: Wisconsin Industrial Affiliates Meeting (2004)
7. Hammond, L., Carlstrom, B.D., Wong, V., Hertzberg, B., Chen, M., Kozyrakis, C., Olukotun, K.: Programming with transactional coherence and consistency (tcc). In: ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, pp. 1–13. ACM Press, New York, NY, USA (2004)
8. Lie, S.: Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology (2004)
9. William, N., Scherer, I., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC 2005. Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, pp. 240–248. ACM Press, New York, NY, USA (2005)
10. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: ISCA 1993. Proceedings of the 20th annual international symposium on Computer architecture, pp. 289–300. ACM Press, New York, NY, USA (1993)
11. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. In: OOPSLA 2006. Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 253–262. ACM Press, New York, NY, USA (2006)
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: PPoPP 2006. Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 187–197. ACM Press, New York, NY, USA (2006)
13. Spear, M.F., Marathe, V.J., Scherer III, W.N., Scott, M.L.: Conflict detection and validation strategies for software transactional memory. In: DISC, pp. 179–193 (2006)