On the Relationship Between Delaying Operators and Language-Level Semantics *

Wenjia Ruan, Yujie Liu, and Michael Spear
Lehigh University
{wer210, yul510, spear}@cse.lehigh.edu

Abstract

The notion of "atomicity" implies that it is safe to rearrange memory accesses within a transaction. In this paper, we sketch a mechanism for postponing contentious transactional operations until commit time, where they become impervious to aborts. We then contemplate the interplay between such a mechanism and language-level semantics. Though preliminary, our algorithms and recommendations should prove useful to designers of transactional compilers and languages.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Design, Performance

Keywords Transactional Memory, semantics, compiler optimizations

1. Introduction

Language-level transactions are often described as providing atomicity, in the sense that the operations that comprise the transaction appear to happen "all at once", as an indivisible operation without any intervening memory operations from concurrent transactions. While Hardware Transactional Memory (HTM) [6] is largely able to provide this illusion for small transactions, Software Transactional Memory (STM) typically cannot, due to the high overheads that arise [7].

The unfortunate consequence is that compilers must be conservative when reordering memory accesses that occur within a transaction. In recent work, we analyzed and transactionalized the memcached web cache [**?**]. When replacing locks with transactions, we discovered that increments to statistics counters, which occurred in the middle of a transaction, dramatically increased contention.

These counters are interesting, in that the transactions that perform the increments never use the return values of their increment operations. The compiler transforms these increment operations into sequences entailing a shared memory read, local computation, and a shared memory write. Thus at the point where a counter increment occurs in the dynamic instruction stream, a transaction becomes susceptible to conflicts with any other transaction that accesses the counter. This can lead to frequent aborts, and in the worst case serialization to ensure progress.

Clearly it would be beneficial to move these increments to the end of the transaction (perhaps in a manner similar to Abstract Nested Transactions [5]). Moreover, doing so would not compromise the programmer's notion of atomicity, nor would it affect the correctness of memcached. Unfortunately, in a complex program, such a transformation is beyond the ability of a compiler with separate units of compilation, and manually transforming the code would severely hinder maintainability. Note, too, that the use of `onCommit` handlers [8] is not appropriate, as these handlers run after the transaction commits, and are not atomic. The purpose of this paper is to consider the complexities that might arise if a run-time system automatically and dynamically reordered operators. We consider a setting in which concurrent transactions' uses of the locations involved in those operators might entail publication [7].

Our focus is on intentional reordering of operations by the programmer or compiler for the sake of improving performance. It is largely complementary to efforts to ensure the correctness of TM in the face of hardware reordering [**?**]. We also focus exclusively on

* This work was supported in part by the National Science Foundation through grants CNS-1016828 and CCF-1218530.

lock-based semantics, as opposed to the stronger but more restrictive case of "strong isolation" [9**?**]. Current trends in the effort to standardize TM in C++ [**?**] suggest that this level of semantics is most likely to appear in C and C++. Note, too, that the concerns we raise are immaterial to implementations that require static separation [1].

The remainder of this paper is organized as follows. In Section 2, we briefly discuss the implementation of a reordering framework for operators in STM. Section 3 then presents an example of how reordering might violate publication safety. Section 4 concludes.

2. Implementation

The guiding principle when reordering operators is that both the read and the write that comprise the operator must be delayed until commit time. While compiler analysis or run-time profiling (as in [**?**]) might be able to identify which operators to reorder, our concern is only on discussing correctness. Without loss of generality, we assume that (a) the programmer will manually annotate those operators she wishes to reorder, and that (b) all operators are ++ operators on 32-bit integers.

The pseudocode for a simple implementation of delayed operators appears in Algorithm 1. The *oplog* data structure stores the addresses to which `operator ++` should be applied at commit time.

The subtlety of this code lies in when operators are applied at commit time, and how they are dynamically downgraded to regular loads and stores via `TxPromote()`. There are four cases to consider:

- read-after-operator: In this case, the transaction attempts to read a location to which there is a pending ++ operation. To ensure processor consistency, the result of the increment must be visible to the read. To achieve this, at the point of the read we transform the increment into a load, increment, and store.
- write-after-operator: Since writeback precedes the replay of the *oplog*, a write that follows an operator can lead to the appearance that writes are reordered. For example, if `x == 0` and the sequence `x++; x = 7;` is executed, the correct answer is `7`, not `8`. To fix the problem, we promote any delayed operation to its corresponding load and store if the location is subsequently overwritten by the same transaction.
- operator-after-read: Typically an STM implementation that uses timestamps and ownership-records only acquires a transaction T to acquire a write lock on a location if that location hasn't been acquired by another transaction since T's last validation. In this manner, a subsequent read-set validation can ignore those locations for which it holds a write lock: any reads to that location were valid when the lock was acquired, and no subsequent changes by other transactions are possible. However, the benefit of delaying operators is that concurrent transactions with conflicting delayed updates to the same counter can both commit. Thus acquiring locks for the *oplog* should not carry this constraint, and instead should be performed in a manner that allows read-set validation to easily determine when a read that was subsequently locked for an increment was also concurrently updated by another transaction.
- operator-after-write: There are no concerns in this case: the operator will be correctly applied to the updated value, since writeback precedes the application of operations stored in *oplog*.

Algorithm 1: Pseudocode for delayed operators

```
1  TxRead(addr)
2      if addr ∈ oplog then oplog.TxPromote(addr)
       // remainder of TxRead unchanged
3  TxWrite(addr, v)
4      if addr ∈ oplog then oplog.TxPromote(addr)
       // remainder of TxWrite unchanged
5  TxOp(addr)
6      oplog ← oplog ∪ {addr}
7  TxPromote(addr)
8      val = TxRead(addr)// assume TxRead skips line 2
9      val ++
10     TxWrite(addr, val)// assume TxWrite skips line 4
11     oplog.remove(addr)
12 TxCommit()
13     writeset.lockAll()
14     oplog.lockAll()
15     readset.validate()
16     writeset.writeback()
17     oplog.performOps()
18     writeset.unlockAll()
19     oplog.unlockAll()
```

3. Impact on Semantics

Menon et al. proposed several levels of transactional semantics [7], which trade serialization at the boundaries of transactions with increasing restrictions on the

Initially: data == 42, ready == false, val == 0

Thread 1:	Thread 2:
1	1 `transaction {`
2	2 `  tmp = data;`
3 `data = 1;`	3
4 `transaction {`	4
5 `  ready = true;`	5
6 `}`	6
	7 `  if (ready)`
	8 `    val = tmp;`
	9 `}`

Can val == 42?

Figure 1: Basic publication example (reproduced from Figure 1 of Menon et al. [7]).

Initially: data == 42, ready == 0, val == 0

Thread 1:	Thread 2:
1	1 `transaction {`
2	2 `  ready++;`
3	3 `  tmp = data;`
4 `data = 1;`	4
5 `transaction {`	5
6 `  ready = 1;`	6
7 `}`	7
8	8 `  if (ready > 1)`
9	9 `    val = tmp;`
10	10 `  else`
11	11 `    ready--;`
12	12 `}`

Can val == 42?

Figure 2: Publication violation example with delayed operators.

programming model. The work primarily focused on racy idioms for initializing a datum and then making it visible to transactions. However, the two least restrictive levels, "Asymmetric Lock Atomicity" (ALA) and "Encounter-time Lock Atomicity" (ELA), are both applicable to C++, where racy code is erroneous. These models differ primarily in whether the compiler can reorder reads of a datum that might be concurrently initialized outside of a transaction.

The canonical example appears in Figure 1, where ALA ensures that the race accessing data is benign, and does not produce the erroneous output `val == 42`, whereas ELA does not. Note that when all transactions are protected by a single global lock, the race is benign, as 42 can never be used by Thread 2. Note, too, that this example might arise under aggressive optimization.

Let us now consider an extension to the code, in which `ready` is a counter, where zero indicates that `data` is not initialized, and any other value is the number of transactions that have used `data` in a successful transaction. This code appears in Figure 2. While admittedly contrived, we hope the reader agrees that this code is not unrealistic: programs might optimistically increment the reference count early, and then undo the increment under situations that are expected to be rare.

ALA is defined as providing the illusion that all read locks are acquired at transaction begin, but write locks can be delayed all the way to commit time. ELA, in contrast, gives the appearance that all read locks are acquired immediately before the first read of the corresponding location (write locks are acquired as in ALA).

In the absence of delayed operators, the example in Figure 2 is correct for both ELA and ALA. However, note that the read of `ready` by Thread 2 will result in a call to `TxPromote`, and effectively cause a read and write of ready to occur *after* Thread 2's transaction commits, instead of before `data` is read. Thus an STM implementation that only provides ELA semantics cannot delay the `++` operator without risking publication violations. Note that this finding extends the work of Menon et al., which showed that dependent reads cannot be reordered above non-dependent reads under ELA. This example demonstrates that assumptions about atomicity and program order forbid additional orderings. We leave as future work determination of whether the error is due to reordering two reads, or reordering a read before a preceding write. Another promising direction is the discovery of generalizations to guide implementation of delayed operators under ELA. Of course, STM implementations that provide ALA semantics (or stronger) are immune to the problem, since all reads by Thread 1's transaction appear to occur when the transaction begins.

Under some circumstances, it may be possible to make delayed operators behave correctly under ELA semantics. Consider a timestamp-based STM, such as TL2 [3] or TinySTM [4]: in these algorithms, the lock that protects `ready` also stores a version number, which corresponds to the time at which the most recent update to `ready` occurred. Suppose that *oplog* were extended to store pairs, where the second value was the version of the lock at the time when the operator was called, and that `TxPromote` only succeeded if the value of the lock was no greater than the value stored in the *oplog*. With these changes, `TxPromote` would

succeed only if the corresponding load and store appeared to happen at Line 1, before Thread 1's transaction modified `ready`. Otherwise, `TxPromote` would fail, and the transaction would abort.

4. Conclusions

The relationship between delayed operators and semantics is subtle, and much work remains before implementations can be sure that delaying operators is correct. Of particular concern is that our proposed solution for ELA semantics seems to require versioned writes: consequently, it may not be possible for all STM implementations to be compatible with delayed operators. In the case of pessimistic STM algorithms with read locking, some [2] have sufficient space in their lock implementations to store monotonically increasing counters as the writer version, while others [10] do not.

Another concern is that while delayed operators prevent aborts and transaction-level contention, they do not eliminate the true memory contention of the underlying application. If concurrent transactions must update the same shared data, then there is no disjoint access parallelism, and even with delayed operators, the best case performance will be limited by coherence traffic on shared data.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, San Francisco, CA, Jan. 2008.

[2] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[4] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[5] T. Harris and S. Stipic. Abstract Nested Transactions. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[6] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[7] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[8] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, Oct. 2008.

[9] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[10] M. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.