

Fast read sharing mechanism for software transactional memory

Yossi Lev

Mark Moir

Sun Microsystems Laboratories

Current multiprocessor architectures do not support synchronization primitives that can atomically access multiple memory locations. Software transactional memory (STM) aims to provide this functionality in software. With STM, a thread uses a transaction to access multiple memory locations, and then tries to commit the transaction. The transaction's operations either take effect atomically when the transaction commits, or do not take effect at all. Part of committing a transaction involves "validating" the transaction, that is confirming that none of the locations it accesses have changed during the transaction. In many cases, it is also important to validate a transaction repeatedly during its execution in order to avoid incorrect behavior due to the transaction observing inconsistent state.

In all kinds of STM, it is desirable to allow concurrent transactions to read memory locations in common, without causing each other to abort. Support for this functionality is called "read sharing". Current STM algorithms do not support read sharing in an efficient way: the simple algorithms suffer from slow validation, while more complex approaches impose a high performance penalty both for a transaction reading a location and for a transaction writing to a location which is currently being read by multiple other transactions.

This paper presents an efficient new read sharing mechanism, which can be easily integrated into most current STM implementations. Our new mechanism is simple, and imposes a minimal cost on transactional read and write operations, while substantially reducing the average cost of validating transactions.

⁰Contact author is Yossi Lev: yosef.lev@sun.com.

1. INTRODUCTION

In concurrent software it is often important to guarantee that one thread cannot observe partial results of an operation being executed by another thread. These guarantees are necessary for practical and productive software development because, without them, it is extremely difficult to reason about the interactions of concurrent threads.

In today's software practice, these guarantees are almost always provided by using locks to prevent other threads from accessing the data affected by an ongoing operation. Such use of locks gives rise to a number of well known problems, both in terms of software engineering and in terms of performance. First, the right balance of locking must be achieved, so that correctness can be maintained, but the use of a particular lock does not prevent access to an unnecessary amount of unrelated data (thereby causing other threads to wait when they do not have to). Furthermore, if not used carefully, locks can result in deadlock, causing software to freeze up. While well understood, these problems are pervasive in concurrent programming, and addressing them often results in code that is more complicated and expensive than we would like.

Software Transactional Memory (STM) allows the programmer to think of the code as if multiple memory locations can be accessed and/or modified in a single atomic step. Thus, in many cases, it is possible to complete an operation with no possibility of another thread observing partial results, even without holding any locks. This significantly simplifies the design of concurrent programs for the reasons mentioned above and others.

With STM, each thread that needs to atomically access multiple memory locations begins a transaction, accesses the memory locations using special transaction's read and write operations, and then tries to commit the transaction: If the transaction was committed successfully, then it is guaranteed that the transaction's modifications appear to take effect atomically.

Beside the Begin, Commit and Read/Write operations the STM supplies, it must also supply the Validate operation that checks if the transaction is still valid, i.e. that it can be still committed successfully. The Validate operation is usually used inside transactions to guarantee that the values read by the transaction from multiple locations are consistent. Such check is necessary to ensure correct behavior of many algorithms that use STM, and is usually performed very frequently (sometimes even after every read operation by a transaction).

In any practical STM implementation, it is necessary to allow concurrent transactions to read memory locations in common, without causing each other to abort. This property is called *Read Sharing*. Unfortunately, current STM algorithms do not support read sharing in an efficient way: the simple algorithms for read sharing result in a very slow Validate operation, while the more complex algorithms present a high performance penalty both for a transaction reading a location and for a transaction writing to a location which is currently being read by multiple other transactions.

This paper presents a new technique for supporting read sharing in STM algorithms. This new technique overcomes the drawbacks of the previous techniques by supplying a simple read sharing algorithm, with minimal performance penalty for the transaction read and write operations, and with a very short Validate operation in the common case.

The rest of the paper is organized as follows: In Section 2 we give an overview

of current implementations for read sharing in existing STM algorithms. Next, in Section 3 we present our new technique and show how it can be integrated in existing STM algorithms. Section 4 discusses some further details, and presents some optimizations and improvements to the basic technique. We conclude in Section 5.

2. OVERVIEW OF CURRENT READ SHARING TECHNIQUES

Most existing STM implementations achieve the appearance of atomic transactions by requiring a transaction to acquire exclusive *ownership* of each memory location it modifies, and by ensuring that the transaction cannot commit successfully unless it maintains ownership of all locations until the transaction attempts to commit. This way, these implementations can guarantee that no transaction sees partial results of another transaction’s modifications.

Acquiring ownership of a location is usually achieved by atomically changing (using a synchronization instruction such as compare-and-swap (CAS)) a special field corresponding to that location. We will refer to this field as the *Ownership Record (orec)* of the location. In the original STM implementation, due to Shavit and Touitou [5], each memory word that can be accessed by transactions has an associated ownership record, resulting in significant space overhead.

Harris and Fraser [1] use an array of orecs, and each memory location is mapped to an orec using some hash function. Thus, space overhead for orecs is reduced, at the cost of some false conflicts due to different locations being mapped to the same orec.

Another solution, used in [2], is to use the memory locations themselves as orecs. This approach assumes we can distinguish between regular values and values indicating ownership, for example by reserving a bit for this purpose.

In the Dynamic Software Transactional Memory implementation of Herlihy *et al.* [3], the orec is a **start** field associated with each object that is accessible by transactions; the **start** field points to a **Locator** structure which holds information about the transaction currently owning this object. This **start** field is atomically changed using CAS when a new transaction tries to acquire ownership on this object.

In general, each orec may be owned by a transaction in Read or Write mode. To support read sharing, we would like to enable multiple transactions to concurrently own the same orec in Read mode. There are two main types of approaches to enabling read sharing in current STM algorithms:

- (1) *Transparent Reading approach:* With this approach, a transaction is not required to acquire ownership on a location before reading it. Instead, each transaction remembers the values it read from all locations accessed for read-only, and when committing or validating, it checks that all these locations still have the same values. This check is necessary to assure that no other concurrent transaction has changed the locations read by the transaction.
- (2) *Non-Transparent Reading approach:* With this approach, each orec keeps a set of all transactions’ IDs that have a read ownership on it (for example by maintaining a linked list of all these IDs). By having this set, a concurrent transaction that needs to acquire a write ownership on an orec in order to change some location, is required to first abort all the transactions that currently have read ownership on this orec, in order to prevent them from later committing successfully.

When using transparent read sharing solutions, the read and write operations of a transaction are relatively cheap, but validation can be very expensive, since it requires rereading all values read by the transaction. We call this procedure of rereading all values for validation *long validation*. Because read operations by transactions are generally more frequent than write operations, long validation may result in a substantial performance penalty. On the other hand, using the Non-Transparent approach enables a constant-time Validate operation, but maintaining the read set for each orec is very expensive and complicated, which may significantly slow down all other transactions' operations: The read operations due to the need to add the reader Id to the set, the Commit operation due to the need to remove it from the set, and the Write operation due to the need to Abort all transactions in the set.

In the next section we present a new technique for read sharing that overcomes these problems without imposing a substantial extra cost for the Read, Write and Commit operations, and with a Validate operation that takes constant time in the common case.

3. THE NEW TECHNIQUE

The new mechanism uses a “semi-transparent” technique. In this technique, each orec contains a counter, which we will refer to as the orec's *read counter*, specifying the number of transactions that currently have read ownership on it. A transaction acquires a read ownership on an orec by simply increasing the orec's read counter. Maintaining such a counter is much simpler and cheaper than maintaining a list of all the transactions' IDs. In addition to the read counters, the algorithm maintains a shared counter called `RWConflictsCounter`, which counts the number of times a transaction tried to get a write ownership on an orec which was owned in read mode by other transactions (i.e. with a non-zero read counter). Therefore, every time a transaction attempts to acquire write ownership of an orec, it increases this counter by 1 if and only if the read counter of this orec was not zero.

We now describe how we use the `RWConflictsCounter` to quickly validate a transaction. When a transaction begins, it reads and saves the value of the `RWConflictsCounter`. We denote the value saved as `RWCCSnapshot`. For validation, a transaction first rereads the `RWConflictsCounter` and compares the value read to `RWCCSnapshot`. If they are equal, the validation succeeds, because it is guaranteed that no transaction got a write ownership on *any* orec which was owned in a read mode since the beginning of the transaction being validated. The only case in which a transaction needs to do long validation is if it detects that the `RWConflictsCounter` was changed since the beginning of the transaction. In this case, it can use the value of `RWConflictsCounter` read during validation as the new value for `RWCCSnapshot`; this way, subsequent validations by the same transaction can again use the fast validation method. Pseudocode for the Validate procedure is given in Figure 1; this procedure is called in order to validate a transaction, and returns true if the transaction is still valid.

Because write operations in transactions are usually much less frequent than reads, we expect that most calls to the Validate procedure will return without needing to do long validation. Note that in such cases, the time complexity of the Validate procedure is constant, and does not depend on the number of locations the transaction has read so far. Considering that the Validate procedure is often executed after every read done by the transaction, we expect the new technique to perform substantially better than the Transparent read sharing mechanism. Fur-

thermore, it is much simpler than existing Non-Transparent read sharing mechanism: there is no need to maintain a list of readers, nor for a transaction acquiring write ownership of a location to individually abort each transaction that currently owns it for reading.

The technique described above is quite general; we believe it is applicable to most existing STM algorithms that use orecs for representing ownerships of transactions on memory locations. We are currently working on building our technique into two STM systems we are working on, and we expect it to improve performance considerably.

Finally, it is important to note that the `RWConflictsCounter` is subject to the ABA problem: Suppose transaction `T` begins and reads a value x from `RWConflictsCounter`. Then, the counter may be incremented, wrap around, and have the value x again. In this case, transaction `T` is unable to detect that the counter was changed, which may result in an incorrect behavior of the `Validate` operation. Therefore we must assure that the `RWConflictsCounter` is large enough (or use some other method like bounded tags [4]) to assure this problem won't arise.

4. IMPROVEMENTS AND FURTHER DETAILS

4.1 Integration with transparent reading

Our technique does require some extra overhead for conflicting write operations, due to the `CAS` needed when incrementing `RWConflictsCounter`. While this extra cost will probably be negligible compared to the benefit of faster validation in transactions that need to validate frequently, it might slow down short transactions that do not. Note, however, that the new technique can live together safely with the old transparent read technique, as long as the two types of transactions do not share the same orecs. Therefore in systems that use the STM mechanism for multiple types of data structures, we can choose to use the new technique only for the data structures which might benefit from it, as long as we can guarantee that

```
bool Validate() {
    if (<Transaction Status> == Aborted)
    {
        return false;
    }
    else
    {
        currCounter = <RWConflictsCounter>;           // Read RWConflictsCounter
        if (<Transaction's RWCCSnapshot> == currCounter) // Snapshot was taken at
        {                                               // beginning of transaction
            return true;
        }
        else
        {
            <Transaction's RWCCSnapshot> = currCounter; // Override the old snapshot
            readsValid = VerifyReads();                 // Do long validation
            return readsValid;
        }
    }
}
```

Fig. 1. Pseudocode for the `Validate` procedure with `RWConflictsCounter`

the memory locations of the two data structures won't share the same orecs. This is naturally true for STM implementations that have a specific orec for each location or object, including [3; 2]. The mapping approach of [1] may require separate orec arrays for separate data structures.

Also note that the overhead of incrementing `RWConflictsCounter` is incurred only if some transaction is concurrently reading from the location being modified; thus the overhead is incurred only in cases in which there is a transaction executing that can benefit from fast validation.

4.2 Write upgrade

In some STM algorithms, there is a special type of ownership acquisition called *Write Upgrade*, which is when a transaction acquires an orec for write for which it already holds a read only ownership. In this case, `RWConflictsCounter` should be increased only if the read counter of the orec was greater than one (since otherwise the current transaction is the only reader and therefore there is no real read/write conflict).

4.3 Contention Management

Herlihy *et al.* advocate using contention management mechanisms to assure progress in their obstruction-free dynamic STM implementation [3]. When using our new read sharing approach, the point at which a transaction is about to attempt to acquire write ownership of an orec already owned for reading by other transactions is an excellent point to consider delaying the writing transaction for contention management: Not only does it allow the reading transactions an opportunity to complete without aborting, but it also helps them to avoid long validation.

5. CONCLUSIONS

We have presented a new efficient technique for supporting read sharing in STM mechanisms. The new technique is simple and can be easily integrated into most existing STM algorithms.

REFERENCES

- [1] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2003), ACM Press, pp. 388–402.
- [2] HARRIS, T., FRASER, K., AND PRATT, I. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing* (2002), pp. 265–279.
- [3] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, W. Software transactional memory for dynamic data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing* (2003).
- [4] MOIR, M. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing* (1997), pp. 219–228.
- [5] SHAVIT, N., AND TOUTOU, D. Software transactional memory. *Distributed Computing* 10, 2 (February 1997), 99–116.