Transaction Synchronizers

Victor Luchangco
Sun Microsystems Laboratories
Burlington, MA 01803
victor.luchangco@sun.com

Virendra J. Marathe
Sun Microsystems Laboratories
Burlington, MA 01803
and
Department of Computer Science
University of Rochester
Rochester, NY 14627
vmarathe@cs.rochester.edu

ABSTRACT

Transactional memory ensures the atomicity and isolation of transactions. Although these properties greatly simplify reasoning about concurrent programs, they prevent concurrent transactions from interacting with each other. In this paper, we propose a *synchronizer* mechanism for augmenting software transactional memory implementations to allow different transactions to communicate with each other by accessing shared data. A synchronizer encapsulates shared data, which can be accessed only by those threads that synchronize at the synchronizer. All threads synchronized at a synchronizer can see the effects on that data by any concurrent threads (which must also be synchronized on that synchronizer). Such interaction necessarily compromises isolation. We limit the extent of this compromise by forcing all the threads synchronized on a synchronizer to commit or abort together. We describe how to implement synchronizers in the context of Dynamic Software Transactional Memory of Herlihy, Luchangco, Moir and Scherer.

Keywords

Transactional Memory, Concurrency, Synchronization, Isolation, Atomicity

1. INTRODUCTION

Transactional memory has been proposed as a way to share data among concurrent threads without using locks. A *transaction* is a sequence of instructions executed by a single thread. Transactions are *atomic*: each transaction either *commits* (it takes effect) or *aborts* (its effects are discarded). Transactions are also *isolated*: effects of other transactions or other external operations (i.e., operations not within a transaction) are not seen within a transaction unless these operations are determined to have happened before the transaction, and external operations do not see partial effects of the transaction.

Transactional memory supports a computational model in which each thread announces the start of a transaction, executes a sequence of operations on shared objects, and then tries to commit the transaction. If the commit succeeds, the transaction's operations take effect atomically; otherwise, they are discarded. Although transactional memory was originally proposed as a hardware architecture [5], there have been several proposals for nonblocking software transactional memory (STM) and similar constructs [1, 2, 3, 4, 7, 8, 9, 11, 12].

The atomicity and isolation of transactions significantly simplify concurrent programming. However, these properties severely restrict what can be done with and within a transaction. For example, they do not permit simple barrier (and condition) synchronization as well as producer-consumer style interactions among concurrent transactions. From a performance perspective, they do not allow concurrency within a transaction. Consider, for example, an application for booking a vacation package, which requires reserving a flight, a hotel room, and a rental car. It may be the case that none of these should be reserved unless the entire vacation is booked; that is, the entire booking should be a single transaction. In current transactional memory systems, even if each of the reservations access logically distinct data, and thus could be processed in parallel, the system must do them sequentially because they are within a single transaction.

We propose a mechanism to enable greater flexibility by relaxing the isolation of transactions, but in a way that preserves many of the benefits of atomicity and isolation. Our mechanism, a *synchronizer*, provides a way for different transactions to share data in a principled way, without compromising isolation for transactions that do not synchronize with each other.

A synchronizer encapsulates data that can be accessed only by transactions that "synchronize" at that synchronizer. Multiple transactions may synchronize concurrently at the same synchronizer, and by accessing

SCOOL 2005 San Diego, CA USA

the data encapsulated by the synchronizer, they may see each others' effects on that data, violating isolation. To limit the extent of this violation, transactions synchronized at a synchronizer must either all commit or all abort. Thus, transactions that do not synchronize at a synchronizer are still isolated; only programmers that use synchronizers need to know about them.

With synchronizers, we can easily implement the application described above for booking a vacation: create a synchronizer for each vacation package, and begin a separate transaction for each of the subtasks of booking the vacation package, with each transaction synchronizing at the synchronizer. This example is a special case of barrier synchronization, which can be implemented with synchronizers. Synchronizers can also be used to elect a leader among several transactions, or to establish a simple producer-consumer buffer between multiple transactions, among other applications.

In this paper, we describe how to extend the *Dynamic Software Transactional Memory* (DSTM) of Herlihy, Luchangco, Moir and Scherer [4] to support synchronizers. We present a simple interface for synchronizers in Section 2, and describe how this interface can be implemented in Section 3. We expect other STM implementations can be extended in similar ways. Finally, in Section 4, we discuss ways to extend the simple interface presented in this paper, other applications and future directions for investigation.

2. INTERFACE

In this section, we describe an interface for synchronizers that is compatible with DSTM. We first give a brief overview of the relevant aspects of the DSTM interface implemented in the Java™ programming language as described in [4], and then describe synchronizers in this context.

2.1 DSTM Interface

DSTM manages a collection of *transactional objects* that are accessed within *transactions*. Transactions are executed by instances of `TMThread`, which extends the Java `Thread` class. `TMThread` provides methods for starting, committing and aborting transactions. All data shared by concurrent threads must be encapsulated within transactional objects, which are instances of `TMObject`. The data object encapsulated within a `TMObject` must implement the `TMCloneable` interface, which requires the object to provide a public `clone` method that returns a new, logically distinct copy of the object (the object being cloned does not change while it is being cloned, so no synchronization is necessary within `clone`).

Within a transaction, a `TMThread` accesses shared data by *opening* the transactional object that encapsulates it.[1] Opening a transactional object returns a copy of the current version of the encapsulated data object (created by calling its `clone` method), which the thread may manipulate within the transaction: we call this copy the transaction's *version*. DSTM guarantees that no other thread accesses this version, so no additional synchronization is necessary. A transaction's version of an object is meaningful only while the transaction is active, so no external references to this version should be created; all references should be to the encapsulating transactional object.

A thread attempts to commit its transaction by invoking `commitTransaction`, which returns *true* if the commit is successful and *false* otherwise. If the commit is successful, then all the transactional objects opened in the transaction are updated to that transaction's version. Otherwise, the transaction is aborted, and its versions are discarded. The commit may fail because the transaction conflicted with some other transaction (i.e., they both accessed the same transactional object). DSTM guarantees that successfully committed transactions are *linearizable* [6]: their effects appear to take place instantaneously at some point during the lifetime of the transaction.

Note that the DSTM interface provided above is not an intrinsic component of the DSTM design. A simpler and more transparent interface, aided by programming language and compiler support, may be used to enable access to transactional objects.

2.2 Synchronizer Interface

A synchronizer is an instance of `TMSynchronizer`, which is similar to `TMObject`: To construct a new synchronizer, we call `TMSynchronizer` with a `TMCloneable` object that represents the shared data. `TMSynchronizer` provides a single method `synchronize`, which a thread calls within a transaction to synchronize its transaction with the synchronizer. This method returns the current version of the data object encapsulated within the synchronizer. As with data objects obtained by opening ordinary transactional objects, this version is meaningful only until the transaction completes, so no external references to this version should be created: such references should be to the encapsulating `TMSynchronizer` object. Also, like ordinary transactional objects, the thread may manipulate this version directly during the transaction. However, unlike ordinary transactional objects, this version may be accessed by any thread synchronized at the synchronizer, so additional synchronization may be necessary to ensure that it is manipulated properly. We say that transactions synchronized at the same synchronizer are *collaborators*.

Collaborators are not isolated from one another because they can see each others' effects on the data encapsulated in the synchronizer. If a synchronized transaction A aborts then A's collaborators must also abort to prevent the observation of partial effects of an aborted transaction by noncollaborating transactions. This implies that the collaborators of A's collaborators must also abort, and so on. To be more precise, we say that a transaction is a *cohort* of A if it is either a collaborator of A or a collaborator of a cohort of A. In other words, the cohort relation is the transitive closure of the collaborator relation. (The cohort relation is an equivalence relation.) If a transaction aborts, then its cohorts also

[1]For simplicity, we consider only opening for writing. We believe that allowing shared read-only access can be handled with little additional difficulty beyond that inherent in DSTM; see [4] for details.

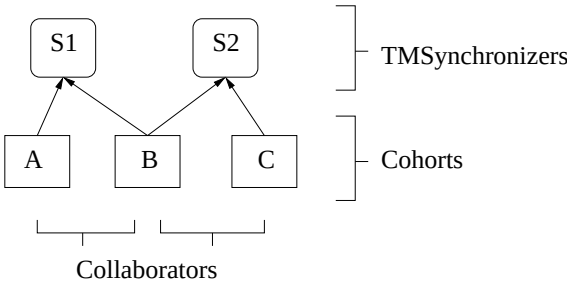

Figure 1: A cohort set of three transactions A, B and C that synchronize on `TMSynchronizers` $S1$ and $S2$

abort. Figure 1 gives an example of collaborators and cohorts.

When a transaction attempts to commit, it must wait until either all its cohorts have attempted to commit or at least one of them is aborted. A transaction commits if and only if all of its cohorts commit, which implies that all the cohorts attempted to commit and none has aborted. To any thread not executing a cohort of a transaction, that transaction and all its cohorts appear to commit together; their effects all appear to take place at some instant during the lifetimes of all the cohorts. Once a transaction has committed, it is no longer synchronized at any synchronizer (otherwise, it would need to be aborted if any other transaction synchronizes at that synchronizer and then aborts).

3. IMPLEMENTATION

This section describes how we extend the DSTM design to support synchronizers.

3.1 DSTM Implementation Overview

A DSTM transaction is represented by a *transaction descriptor*, which contains a *status* field. The value of this field may be `ACTIVE`, `ABORTED`, or `COMMITTED`. A thread begins a transaction by allocating a new transaction descriptor with its status `ACTIVE`. A thread commits its transaction by atomically changing its status from `ACTIVE` to `COMMITTED`. A transaction is aborted by atomically changing its status from `ACTIVE` to `ABORTED`. Once the status of a transaction is `ABORTED` or `COMMITTED`, it never changes again.

A `TMObject` consists of a reference to a *locator*, which in turn consists of three references: one to the transaction that most recently opened the `TMObject`, one to that transaction's version of the data object encapsulated within the `TMObject`, and one to the version of the data object that was current when the transaction opened the `TMObject`. We call these two versions the *new version* and *old version* respectively. The current version of the data encapsulated by the `TMObject` is determined by the status of the transaction: if the status is `ACTIVE` or `ABORTED` then the old version is current; otherwise, the new one is. Thus, when a transaction's status changes from `ACTIVE` to `COMMITTED`, the current versions of the `TMObject`s that it opened all change atomically from their old versions to their new versions.

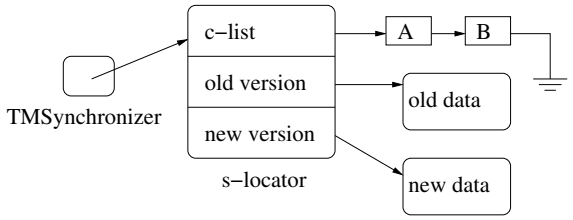

Figure 2: A `TMSynchronizer` shared by two transactions

To open a `TMObject`, a transaction reads the `TMObject`'s locator to determine the current version of the encapsulated data object, and creates a new locator pointing to its transaction descriptor, and whose old version is the current version, and new version is a new clone of the current version (created using the `clone` method of the data object). The transaction then attempts to atomically replace the `TMObject`'s old locator with this new locator.

To guarantee isolation, the DSTM ensures that for any `TMObject`, at most one `ACTIVE` transaction has opened it. If a transaction A attempts to open a `TMObject` that another `ACTIVE` transaction B has already opened, then we say that the transactions *conflict*. In that case, A must either abort B or wait until B commits or aborts for some other reason.

3.2 Synchronizer Implementation

A `TMSynchronizer` is similar to the `TMObject` in structure: It contains a reference to a special *s-locator* object. The s-locator in turn contains references to the old and new versions of the encapsulated data object, and instead of a reference to a single transaction (as in the locator), the s-locator contains a reference to a list of collaborator transactions, the *c-list*, that have synchronized on that `TMSynchronizer`. Figure 2 depicts a `TMSynchronizer` with two collaborators. A transaction synchronizes on a `TMSynchronizer` by atomically inserting itself in the `TMSynchronizer`'s c-list.

Because cohorts must commit or abort together, there is a race between committing cohorts and newly synchronizing transactions: Cohorts that have decided to commit must not do so if a new `ACTIVE` transaction synchronizes with them in the meantime. Similarly, transactions that intend to synchronize on a `TMSynchronizer` must verify if the `TMSynchronizer`'s existing cohort set has decided to commit, and either synchronize with them, or synchronize with the "next version" of the `TMSynchronizer`. Also, conflicts with synchronized transactions need to be handled differently than the typical conflict resolution method in DSTM.

3.2.1 Determining Cohorts

To enable a transaction to find all its cohorts, we maintain in the transaction descriptor, a list of all the synchronizers that the transaction synchronized at; we call this list the *s-list*. Considering cohorts and the synchronizers that they synchronized at as nodes, and the

s-lists and c-lists as adjacency lists, we define the *cohort graph* of a cohort set. A transaction can find its cohorts by traversing the cohort graph it belongs to.

3.2.2 Committing Cohorts

A synchronized transaction that is ready to commit can commit only if all its cohorts are also ready to commit. Thus, a synchronized transaction must convey its "intention" of committing to its cohorts. We enable this functionality by allowing the status of a transaction to take a new value, `COMMITTING`. To commit, a transaction that synchronizes at any synchronizers changes its status from `ACTIVE` to `COMMITTING`, and waits for its cohorts to also change their status to `COMMITTING`.

If all transactions in a cohort set C switch their state to `COMMITTING`, they may attempt to commit. These cohorts cannot, however, commit as soon as they determine that they are all in `COMMITTING` state: Another transaction A, not in C, may synchronize at a synchronizer that was accessed by a tranasction in C, thu joining the cohort set. Thus, there is a race condition between A, trying to join the cohort set C, and the transactions in C, trying to commit. Any transaction in C that commits before A is linked to C will not know about A, which may subsequently abort, forcing all the transactions in C to abort. But some of the transactions of C would have already committed!

To avoid this race between committing cohorts and newly synchronizing transactions, we introduce the notion of a *seal*. The seal acts as a delimiter between cohort sets of different versions of the `TMSynchronizer`. We implement the seal as a special node that is atomically inserted by a transaction at the head of a `TMSynchronizer`'s c-list. A transaction that has synchronized on a `TMSynchronizer`, must seal the `TMSynchronizer`'s current version before committing. A `TMSynchronizer`'s version is sealed only when all its collaborators have switched to `COMMITTING` state. At this point, any collaborator may atomically seal the `TMSynchronizer`.

In order to commit, a synchronized transaction A must ensure that all its cohorts are in `COMMITTING` state, and all the `TMSynchronizer`s linked to the cohorts are sealed. If so, A and its cohorts have already logically committed. A makes this commit explicit by switching its state to `COMMITTED`. If, for each unsealed `TMSynchronizer` S linked to A's cohorts, all transactions synchronizing on S have switched to `COMMITTING` state, A tries to seal S's current version by atomically inserting a seal node at the head of S's c-list. If this insert is successful, A has effectively sealed S's current version. However, if there exists at least one transaction (synchronized on S) that is in `ACTIVE` state, S's current version cannot be sealed unless that transaction switches to `COMMITTING` state. A waits for all such `ACTIVE` transactions to switch to `COMMITTING` state. If at least one transaction has switched to `ABORTED` state, A must also abort since its fate is tied to its (aborted) cohort. Every transaction in A's cohort set, that is intending to commit must perform the same actions as A does. The last successful seal operation logically commits *all* the cohorts.

3.2.3 Synchronizing Transactions

As mentioned earlier, a transaction may synchronize on a `TMSynchronizer` by atomically inserting itself in that `TMSynchronizer`'s c-list. However, this simple transaction synchronization mechanism works only if the `TMSynchronizer`'s current version is unsealed. If the `TMSynchronizer`'s version is sealed a transaction goes through a protocol, similar to the commit protocol, to synchronize at the current version of the `TMSynchronizer`.

A transaction A, intending to synchronize on a `TMSynchronizer` S, which has already been sealed, cannot immediately synchronize with the sealed version of S – Although S's current version is sealed, all transactions linked to it (all its cohorts) may not have logically committed. Thus, A will have to look up the cohort set linked to the sealed version of S to check if all cohorts have switched to `COMMITTING` state, and all the `TMSynchronizer`s linked to these cohorts have been sealed. If so, it means that all cohorts have indeed logically committed (although their state may not yet be `COMMITTED`). Additionally, since a synchronized transaction may switch to `COMMITTED` state only after its cohorts have all switched to `COMMITTING` state and all the linked `TMSynchronizer`s are sealed, if A encounters a cohort that is in `COMMITTED` state, it can assume that the last set of collaborators synchronized on S have also committed. Note that if a transaction switches to `COMMITTED` state, all its cohorts must be either in `COMMITTING` or `COMMITTED` states; none can be in `ACTIVE` or `ABORTED` states. Therefore the new data object in S's s-locator is the current data version. A creates and synchronizes on a new version of S as follows: A creates a new s-locator, initializing its c-list to contain just A, and setting the old version reference to point to the new data object version in S's current s-locator, and the new version reference to point to a clone of this data object. A now atomically swaps its new s-locator into S. S is thus reinitialized and *unsealed* by A; A in turn has synchronized on S's new version.

If at least one cohort is in `ACTIVE` state, A must either wait for that cohort to switch to `COMMITTING` state, or simply abort it. In other words, A has encountered a conflict with one of the cohorts. A contention manager may decide whether A has to wait for, or to abort this conflicting cohort. If A aborts this conflicting cohort, it must reinitialize and unseal S as above (except that the old version in the new s-locator will point to the old data object version in S's current s-locator, and the new version reference will point to a clone of this data object).

3.2.4 Conflict Management

Conflicts between unsynchronized transactions are resolved in the DSTM-style fashion using contention managers. Conflicts with `ACTIVE` synchronized transactions are resolved similarly. However, if a transaction A detects a conflict, over a `TMObject` O, with a synchronized transaction B, which is in `COMMITTING` state, A must determine whether B has logically committed. A checks B's cohorts and the `TMSynchronizer`s that these cohorts have synchronized on. If all the cohorts are in states

COMMITTING or COMMITTED, and the TMSynchronizers are all sealed, A can proceed opening O by assuming that B has already committed. If an unsealed TMSynchronizer has all its collaborators in COMMITTING state, A explicitly tries to seal that TMSynchronizer on behalf of that TMSynchronizer's collaborators. However, if any of B's cohort is still ACTIVE, A has to decide whether to abort that cohort or wait for it to switch to COMMITTING state. This decision can be made by a contention manager. If A aborts B's ACTIVE cohort, it has effectively aborted B. Thus A can open O assuming that B has already aborted. If any of B's cohorts has already aborted, A can proceed opening O assuming that B has already aborted.

An aborted synchronized transaction, say A, may be waiting for a cohort to interact with. However, since A, and hence its cohort set has been aborted, no cohort may show up to interact with A. In this case, A may end up waiting for a cohort indefinitely. It is the programmer's responsibility to validate that A has not aborted in such a situation. Thus, the validate method must be incorporated in DSTM's interface.

4. CONCLUSION AND FUTURE WORK

In this paper, we have described how to extend software transactional memory with synchronizers that relax the isolation property of transactions by allowing different transactions to collaborate by accessing shared data. This relaxation allows us to write programs with barrier synchronization within transactions, and programs with more concurrency than would otherwise be possible. Because collaborating transactions must abort or commit together, and because they can only see each others' effects on data protected by synchronizers they explicitly synchronize at, we believe that the use of synchronizers should not make concurrent programs too difficult to reason about, as long as the collaboration itself is not difficult. Furthermore, transactions that do not collaborate are still isolated, so programmers that do not use synchronizers do not need to know about them.

The synchronizer mechanism discussed in this paper can be used for plain barrier synchronization among concurrent transactions. Although this is useful for increasing concurrency within atomic blocks, it does not provide a clean solution to condition synchronization among concurrent transactions. The reason being that condition synchronization indirectly relies on conditional binding of transactions, where transactions bind to others based on certain run-time conditions. For example, consider an `exchange channel` [10], where two arbitrary threads can exchange private data items. In a simpler transaction synchronizer based version of the exchange channel if three transactions happen to synchronize on the channel at about the same time, two will successfully exchange data items leaving the third transaction unnecessarily bound to the first two transactions. Now if the third transaction happens to abort, the other two transactions must also abort. In general, implementations of such constructs using transaction synchronizers seems non-trivial.

There are several ways in which the simple synchronizers we described can be extended to enable conditional binding. For example, the synchronize method might be permitted to fail, returning *false* if the requesting transaction is not actually synchronized at the synchronizer. This could be helpful, for example, in the vacation example discussed in the introduction, if there are multiple possible flights, hotels and rental cars, and only the first of each kind should succeed in registering at the synchronizer. We could also allow transactions to "unsynchronize" if they have not modified the encapsulated data. This would be similar to the "early release" functionality provided in DSTM. Alternatively, a synchronized transaction could discard the changes it made to its synchronizers and abort a collaborating transaction or a set of collaborating transactions that have observed these changes.

Finally, we could provide an "observe" method, which occurs before actually synchronizing. This would enable the decision of whether to actually synchronize or not to depend on the data encapsulated in the synchronizer and possibly on what transactions are already registered at that synchronizer.

Alternatively, the synchronizer interface could be extended with two methods: "register" and "unregister". The register method simply registers (does not synchronize) a transaction at a synchronizer. A registered transaction has the right to observe the synchronizer and decide to either synchronize on it or unregister. This enables registered transactions to see each other, along with the synchronized transactions. The observe and register/unregister mechanisms areee also similar to the early release functionality provided in DSTM.

Although these extensions show the potential for conditional binding among concurrent transactions, they expose the data encapsulated by the synchronizer to unsynchronized transactions. Thus these extensions provide richer functionality albeit several difficulties for enabling conditional binding.

Acknowledgements

We are grateful to Mark Moir for his comments and suggestions on an earlier draft of our paper.

5. REFERENCES

[1] Barnes, G. A Method for Implementing Lock-Free Shared Data Structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures* (1993), pp. 261–270.

[2] Fraser, K., and Harris, T. Practical Lock-Freedom. Tech. Rep. UCAM-CL-TR-579, Cambridge University Computer Laboratory, 2004.

[3] Harris, T., and Fraser, K. Language Support for Lightweight Transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications* (2003), pp. 388–402.

[4] Herlihy, M., Luchangco, V., Moir, M., and William N. Scherer, I. Software transactional memory for dynamic-sized data structures. In

Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (2003), pp. 92–101.

[5] Herlihy, M., and Moss, J. E. B. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (1993), pp. 289–300.

[6] Herlihy, M. P., and Wing, J. M. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (1990), 463–492.

[7] Israeli, A., and Rappoport, L. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing* (1994), pp. 151–160.

[8] Marathe, V. J., Scherer III, W. N., and Scott, M. L. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing* (2005), pp. 354–368.

[9] Moir, M. Transparent Support for Wait-Free Transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms* (1997), Springer-Verlag, pp. 305–319.

[10] Scherer III, W. N., Lea, D., and Scott, M. L. A Scalable Elimination-based Exchange Channel. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages* (2005).

[11] Shavit, N., and Touitou, D. Software Transactional Memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (1995), pp. 204–213.

[12] Turek, J., Shasha, D., and Prakash, S. Locking without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems* (1992), pp. 212–222.