Revisiting Condition Variables and Transactions

Victor Luchangco
Oracle Labs
victor.luchangco@oracle.com

Virendra J. Marathe
Oracle Labs
virendra.marathe@oracle.com

Abstract

Prior condition synchronization primitives for memory transactions either force waiting transactions to abort (the retry construct), or force them to commit (also called *punctuation* in the literature). Although these primitives are useful in some settings, they do not enable programmers to conveniently express idioms that require synchronous communication (e.g., n-way rendezvous operations) between transactions. We present xCondition, a new form of condition variable that neither forces transactions to abort, nor to commit. Instead, an xCondition creates dependencies between the waiting and the corresponding notifying transactions such that the waiter can commit only if the corresponding notifier commits. If waiters and notifiers form dependency cycles (for instance, in synchronous communication idioms), they must commit or abort together. The xCondition construct builds on our earlier work on *transaction communicators*. We describe how to use xConditions in conjunction with communicators to enable effective coordination and communication between concurrent transactions. We illustrate the use of xConditions, and describe their implementation in the Maxine VM.

1. Introduction

Locks, used to provide isolation for critical regions of code, and semaphores, used to order events (such as the execution of various critical regions), are among the earliest synchronization mechanisms in concurrent programming. Experience has shown that it is not sufficient to have two independent mechanisms. Rather, it is desirable to have a structured mechanism that integrates them, resulting in *monitors* [7], which replaced semaphores with *condition variables*. This mechanism evolved slightly in Mesa [8] and has been widely used essentially unchanged since then. As we consider using transactional memory instead of locking to provide isolation, it behooves us to consider also how to provide *condition synchronization* with transactions, that is, a mechanism that integrates transactions and the ordering of events in a manner analogous to condition variables.

We are not the first to consider this problem. Existing proposals include (i) *conditional critical region (CCR)* style transactions [4], which allow execution of a transaction only if a particular condition is satisfied (i.e., the execution of the transactions is delayed until the condition is true); (ii) a retry construct [5], which aborts a transaction that calls retry, and re-executes it only when something in that transaction's read or write set is modified; and (iii) a wait construct [10] (or condition variable [3]) that "punctuates" (i.e., commits) the waiting transaction and begins a new transaction for the waiting thread on receipt of a notification from a concurrent transaction. However, none of these proposals allow synchronous communication between concurrent transactions.

Consider, for example, a system that processes client jobs that should appear to be handled atomically. Processing some of these jobs may involve accessing a database, and these accesses should themselves appear atomic. The system may be organized as illustrated in Figure 1, with the threads handling client jobs separate from those with direct access to the database, so that a thread handling a job that requires access to the database must place a request to do so into a queue; the request will be handled by a database thread. In such a system, a thread handling a job cannot simply execute the job in a single transaction: if the job requires access to the database, the thread handling the job must communicate with the database thread that handles its request. Nor can the thread simply break the job into two transactions, one to handle the part before the request and the other to handle the part after getting the result: if the transaction for the second part aborts, then the effects of the first transaction, and of the database thread that satisfied the request, should be discarded as well.

The above-mentioned proposals do not suffice for this problem because they either require that waiting transactions do not happen at all (retry and CCRs), or break a single isolated transaction into multiple smaller transactions, via punctuation, thus compromising isolation of the original bigger transaction. A transaction-based solution requires bi-

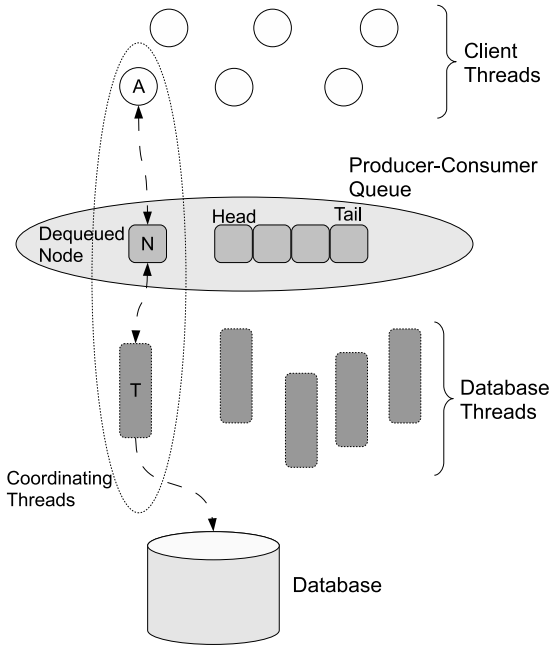

Figure 1. An example scenario where transactions may need to coordinate and communicate with each other.

directional communication between concurrent transactions, and an effective means of coordinating the execution of the transactions, that is, in ordering the various parts of the transactions. We addressed the communication aspect in earlier work [9], from which this example is taken, by proposing special objects called *transaction communicators*, through which concurrent transactions can communicate. However, in that paper, a waiting thread simply spins on a communicator, which introduces the usual problems that motivate ordinary condition variables. In addition, this spinning introduces unnecessary dependencies between waiters and notifiers, especially when many transactions wait on the same condition (like the database threads in the above example).

In this paper, we propose xCondition, a new kind of transaction condition variable that does not abort or punctuate a transaction that waits on it. An *active* waiting transaction can receive a notification from an active notifying transaction, which compromises the isolation of the waiting transaction (i.e., it "knows" it did not execute in isolation because it receives the notification). As with our work on communicators, we restrict this compromise by enforcing a dependency between the waiter and the notifier such that the waiter can commit only if the notifier commits. In particular, if the notifier aborts, the waiter must also abort. Unlike simple ordering constraints, however, a pair of mutually dependent transactions (e.g., if each waits, at different times, for something the other produces, as in the example above) may both commit together. Unlike ordinary transaction communicators, however, an xCondition maintains a queue of transactions waiting to be notified, and if a transaction so notified aborts, then the signal it received must be propagated to another thread on the queue (if any).

As with ordinary condition variables we expect that xConditions will be used to protect and communicate data, and so will be used with transaction communicators. Therefore, we propose an extension to the communicators abstractions for better interoperability with xConditions.

2. Transaction communicators

In this section, we describe briefly our earlier work on transaction communicators [9].

A *transaction communicator* is a special object that enables desirable communication between concurrent transactions, but limits the impact of the resulting compromise of isolation: Updates to a communicator by a transaction are visible to other transactions accessing the communicator, even before the updating transaction commits, but a transaction that sees the effects of another transaction must not commit unless that other transaction commits, nor precede the other transaction in the transaction order. Thus, communicators induce dependencies among concurrent transactions. In contrast to ordinary transactional memory, mutually dependent transactions induced by cyclic dependencies on communicators may commit, provided that they all commit. In this case, they all occupy the same position in the transaction order.

Note that inter-transaction communication is allowed only through communicators: a transaction that sees the effects of another transaction on a *non-communicator* object must be ordered strictly after the other transaction. Also, no committed transaction may see the effects of an aborted transaction. If we think of a set of mutually dependent committed transactions as a "super-transaction", with each committed transaction in exactly one super-transaction, then these super-transactions appear to execute one at time, but the operations of transactions within the same super-transaction may appear to be interleaved in any way consistent with the sequential semantics of each transaction.

Because operations on communicators within transactions are visible to other transactions, transactions must synchronize their access to communicators. To encourage the transactional style, we provide *communicator-isolating transactions* (CITs), which isolate accesses to communicators (as well as ordinary objects). All operations within a CIT, including accesses to communicators, appear to execute together without interleaving with operations of other threads.

Because CITs isolate communicator accesses and ordinary transactions do not, we cannot simply flatten a CIT nested within an ordinary transaction into its parent: when the nested CIT commits, its effects on communicators—but not its effects on non-communicator objects—must be made visible. (A CIT within a CIT, and an ordinary transaction

```
class xCondition {
    public void txwait();
    public void txnotify();
    public void txnotifyAll();
}
```

Figure 2. The xCondition class

within an ordinary transaction, can be flattened because no effects are made visible to other threads when the nested transaction commits.)

Assuming a Java-like language with support for atomic blocks [4], we proposed two language-level constructs for communicators: the txcomm modifier to designate fields as read-write communicators, and the txcommatomic block to designate blocks of code to executed as CITs.

3. The xCondition

We now describe the xCondition abstraction in more detail in the context of the Java-like language with support for atomic blocks and communicators described in the previous section. We define xCondition as a class with a public interface as shown in Figure 2. The txwait, txnotify and txnotifyAll methods correspond respectively to the wait, notify, and notifyAll methods of the Object class in Java.[1] As discussed below, we only support calling these methods from within (the dynamic extent of) a txcommatomic block.

Like an ordinary condition variable, an xCondition maintains an abstract *wait list* of transactions waiting for a notification on the xCondition. However, an ordinary condition variable does not work within a transaction because a transaction is supposed to appear to run in isolation: a thread that waits within a transaction would never be notified. With xConditions, as with transaction communicators, we relax the isolation of transactions to enable communication between concurrent transactions.

Abstractly, a transaction calling txwait on an xCondition is added to the xCondition's wait list, and suspended until it is notified. When a transaction invokes txnotify on an xCondition, the runtime determines if there is a transaction on the xCondition's wait list, and if so, notifies such a transaction, removing it from the wait list and scheduling it for execution. It also makes the waiter dependent on the notifier, so that the waiter cannot commit unless the notifier does. If a transaction invokes txnotifyAll, then all transactions on the wait list are notified.

Note that the dependency between a waiter and a notifier is one-way: notification does *not* make the notifier depend on the waiter; the notifier can commit even if the waiter it notifies aborts. Nonetheless, cyclic dependencies may be introduced if, for example, a notifier subsequently waits on a condition that is satisfied by the the transaction it notified, as in the case of our job processing example: a client transaction waits for a response from the database thread that processes its request. As with transaction communicators, transactions in dependency cycles must either all commit or all abort, and if they commit, they occupy the same position in the transaction order, appearing to other transactions as a "supertransaction". Dependencies introduced by xConditions are treated exactly the same as dependencies introduced by ordinary communicators, and a cycle may consist of both kinds of dependencies. Enforcing these dependencies can be done as described in our paper on communicators [9].

However, enforcing the dependencies between waiters and notifiers is not enough. If a waiter is notified and is subsequently aborted, the notification must not be lost: it must be "forwarded" to some other transaction, if any, that was waiting on the xCondition. (This is not true if the notification occurred due to a txnotifyAll, because in that case, all waiting transactions were already notified.) We describe how we implement notification forwarding in Section 5.

3.1 xCondition and ordinary communicators

Although xConditions share many characteristics with our earlier transaction communicators [9], there are also some significant differences, and it is not possible to implement xCondition directly with communicators.

First, implementing the wait list of a xCondition using ordinary communicators would introduce many more dependencies than the simple one-way dependence of a waiter on the transaction that notified it. Such an implementation would almost always result in many transactions being mutually dependent because of reading and writing communicators used to implement the wait list.

More fundamentally, whereas a communicator encapsulates shared data, which transactions can use to communicate, a condition variable captures a communication *event*, that is, the notification. Thus, when a transaction aborts, any notification it received must be restored and forwarded to another waiter (if any) that may have received it instead. There is no equivalent read, write, or dependency forwarding mechanism in communicators.

Also, the txwait method of a xCondition is *blocking*: it does not return until some other transaction calls txnotify. Thus, although a txwait is necessarily invoked *before* the notification it receives, it is dependent on, and semantically ordered *after* that notification, and so cannot be conceived as a single atomic operation. In contrast, operations on communicators always return, even if only to indicate that there is no valid value to be read, and can be thought as atomic operations on encapsulated data. This makes the semantics of communicators easier to describe, but does not allow the flexibility of txwait, which allows it to avoid introducing the unnecessary dependencies described above.

Because txwait is blocking, the naive semantics do not make sense within a txcommatomic block: a thread that

[1] Alternatively, we could make xConditions like Java monitors, where there is no explicit xCondition class and the above methods are added to the Object class. Such considerations are orthogonal to the xCondition functionality.

appears to run in isolation cannot receive a notification from another thread. We discuss this issue further in Section 3.3.

3.2 Combining xConditions with communicators

When a thread is notified after waiting on a condition variable, it typically reads some data to determine whether the condition it was waiting for is indeed satisfied. Because xConditions, like condition variables in general, do not encapsulate data, this data must be in other objects. And because transactions cannot see the effects of other threads on ordinary objects, such data flow must occur through transaction communicators. For example, consider the simple condition synchronization idiom, in which X is a xCondition, one thread executes the following transaction:

```
atomic {
    ...
    while (!canProceed) {
        X.txwait();
    }
    // access data protected by X
    ...
}
```

and another thread executes the following code:

```
atomic {
    ...
    // initialize data protected by X
    canProceed = true;
    X.txnotify();
    ...
}
```

That is, the notifier wakes up the waiter after modifying the shared state that the waiter accessed before waiting (i.e., canProceed), and will access after waking up (data protected by X). Unless canProceed is a communicator, this will lead to a conflict that must result in at least the waiter aborting.

The code above can suffer from *lost notifications*, a well known problem with condition synchronization: the notifier's update of canProceed and notification on X may occur between the waiter's access of canProceed and its subsequent txwait call. With ordinary condition variables, losing notifications is avoided by protecting the data with a lock that is held by the thread before calling wait. This lock is released only after the thread is added to the wait list, and it is acquired again when the thread is notified. We wish to avoid locking and instead use communicator-isolating transactions, which introduces some subtleties discussed in the next section.

3.3 xCondition and communicator-isolating transactions

To avoid lost notifications in the example in the previous section, we must synchronize the xCondition operations with the checking and establishing of the desired condition. To retain a transactional style of programming, we want to accomplish this synchronization using a communicator-isolating transaction (CIT) rather than a lock. Thus, for the example in the previous section, the waiter should be written as follows:

```
atomic {
    ...
    txcommatomic {
        if (!canProceed) {
            X.txwait();
        }
        // access data protected by X
    }
    ...
}
```

Similarly, a notifier should execute the following code:

```
atomic {
    ...
    txcommatomic {
        // initialize data protected by X
        canProceed = true;
        X.txnotify();
    }
    ...
}
```

However, as mentioned before, the naive interpretation of txwait does not make sense within a CIT. We must, therefore, define the semantics in this case. In choosing a semantics, we consider how condition variables are used. In particular, ordinary condition variables are typically used *only* in patterns like the one described in the previous section (sometimes enforced by the language), to avoid the lost notification problem. So our semantics must support this pattern. Indeed, in analogy with the restriction on the use of ordinary condition variables, we only support the use of xCondition operations within the dynamic extent of a txcommatomic block.

There are several desiderata we have for the pattern above: First, in keeping with the spirit of condition synchronization, we want to admit an implementation that allows the waiting thread to be descheduled until it is notified. Second, we want to retain the appearance of isolation for the txcommatomic block. Third, we want to avoid establishing unnecessary dependencies among the transactions that access the xCondition. Fourth, we want to ensure that notifications are not lost.

We achieve these goals by adopting a "retry-on-txwait" semantics, with a twist: When a thread invokes txwait within a CIT, it behaves as though it invoked retry as proposed by Harris et al. [5], aborting and re-executing the nested transaction (i.e., the CIT). However, rather than retrying the transaction when any data that it read changes, it retries upon being notified. If the desired condition is satisfied, the waiting thread will not invoke txwait when it re-executes the nested transaction. In a sense, it is like the txwait occurs at the beginning of the CIT: the notification is the first event of the CIT, and all the subsequent accesses appear to take place atomically upon receiving it, without the interleaving of operations of any other thread.

If, upon re-executing the nested transaction, the thread again invokes txwait (possibly on a different xCondition), then the nested transaction is again aborted, and the thread again waits to be notified. Because the nested transaction is aborted, the notification it received is then forwarded to some other waiting transaction (if there is any).

When a thread commits a nested CIT after having waited on an xCondition, its enclosing transaction depends on the transaction whose notification it receives, as well as any transaction whose effects it observes (e.g., that wrote a communicator that it read) within the CIT. However, it is *not* dependent on transactions whose effects it observed in previous aborted attempts to execute the txcommatomic block, including those whose notifications it received but forwarded because the subsequent CIT aborted.[2] Note that because some notifier may not change the program state to enable any of the existing waiters to make forward progress, all the waiters may end up waiting after they received the (possibly forwarded) notification. The runtime system should be able to ignore (drop) such notifications.

This semantics allows us to improve the usual pattern for condition synchronization by eliminating the explicit loop (see the above example): In our semantics, a waiter consumes at most one notification for each committed CIT, whereas with ordinary condition synchronization, a waiter may consume multiple notifications (the notifier might not make the condition true, or some other thread might intervene and falsify the condition) before it finds its desired condition satisfied.

Note that because a transaction (including a nested CIT) may always be aborted, we could have adopted a semantics in which notifications are *never* received: the nested CIT is simply aborted upon txwait, and just happens to be re-executed at some time when it can commit without invoking txwait. Thus, txwait would be exactly equivalent to retry. Indeed, an implementation that does this is correct according to our semantics. We believe that our semantics is truer to the spirit of condition synchronization, but, in any case, the only difference between the two semantics is whether the notification is "consumed" and a dependency is established, and even the latter is unlikely to be an actual difference because the waiter will likely read some data written by the notifier, and so be dependent on it in any case.

4. Illustrations

We now illustrate the use of xCondition, in conjunction with communicators, in two different applications: the client-server application from Figure 1, and a scenario where error conditions encountered in a transaction need to be "forwarded" to an error logger thread (in our case, a transaction). Both examples show the effectiveness of expressing idioms that were difficult or impossible to express with previous condition synchronization constructs. We first present a way of coding a producer-consumer queue using xConditions and communicators. This queue is later used in our client-server application example.

[2] We are grateful to the anonymous reviewer who pointed out that an earlier description of our semantics made the waiting transaction dependent on all transactions whose notifications it received, and that this was undesirable.

```
class ProducerConsumerQueue {
    txcomm Node head = null;
    txcomm Node tail = null;
    xCondition xc = new xCondition();
    public void produce(Object data) {
        Node myNode = new Node(data);
        txcommatomic {
            if (tail == null) {
                head = tail = myNode;
            } else {
                tail.next = myNode;
                tail = myNode;
            }
            xc.txnotify();
        }
    }
    public Object consume() {
        Node node;
        txcommatomic {
            if (head != null) {
                // queue is not empty, do the dequeue
                node = head;
                if (head.next == null) {
                    // dequeuing last node
                    head = tail = null
                } else {
                    head = head.next;
                }
                return node.data;
            } else {
                xc.txwait();
            }
        }
    }
}
class Node {
    txcomm Object data;
    txcomm Node next;
    public Node(Object data) {
        this.data = data;
    }
}
```

Figure 3. xCondition-based producer-consumer queue

A producer-consumer queue Producer-consumer queues are used widely in concurrent applications. In our earlier work on communicators [9], we showed how to implement a producer-consumer queue communicator using read-write communicators (i.e., the txcomm fields). However, in that implementation, a consumer waiting for an item to be produced simply "spun" until the queue was nonempty, so the consumer could claim the item. A transactional producer-consumer queue can be implemented with the retry construct. However, that would preclude communication idioms such as the one illustrated in Figure 1.

Figure 3 depicts the new xCondition-based producer-consumer queue. Note that the queue fields head and tail, and the queue node fields data and next are all designated as communicators because all the producer and consumer transactions read and write these fields. Both the producers and consumers access these fields in a txcommatomic block. A consumer calls txwait when it finds the queue to be empty. As per our semantics, calling txwait essentially aborts the the consumer's txcommatomic block, and forces it to wait for a notification. A producer always sends a notification (to some waiter, if there is any) whenever it adds a new node to the queue.

```
// a producer-consumer queue used by the client and database
// transactions to communicate with each other
ProducerConsumerQueue pc;

// client side transaction
//
atomic {
    // some computation
    ...
    // post a request
    pc.produce(myRequest);

    txcommatomic {
        // myRequest.response is a txcomm field
        if (myRequest.response == null) {
            // wait for the response to show up
            myRequest.condvar.txwait();
        }
    }

    // more computation based on the response
    ...
}

// database side transaction
//
atomic {
    // wait for a client request
    clntRequest = pc.consume();
    // process the request
    localResponse = process(clntRequest);
    txcommatomic {
        clntRequest.response = localResponse;
        clntRequest.condvar.txnotify();
    }
}
```

Figure 4. Using xConditions and communicators to enable interaction between the client and database transactions.

Revisiting the client-database transaction application Building on the producer-consumer data structure just presented, we now present a solution to the example application from Figure 1. Recall that the application consists of a client transaction that collaborates with a concurrent database transaction to process a request atomically. A client transaction creates a request and posts it in a producer-consumer queue. A database transaction gets a client's request from the producer-consumer queue, processes it, and posts a response for the client transaction's consumption. The client transaction gets the response and may process it further. All this computation must happen atomically. Figure 4 depicts the high level pseudo code of the client and database transactions. The client (first half of Figure 4) posts its request in a shared producer-consumer queue, and uses a txcommatomic block, along with a TxCondVar, to wait for a response from the database transaction. The database transaction (second half of Figure 4) gets a request from the producer-consumer queue, and waits for one, using the queue's TxCondVar, if it needs to. It then processes the request, and posts a response, which includes sending a notification on the request's TxCondVar. The mutual dependencies generated between the two transactions force them to commit or abort together as a super-transaction.

Logging error conditions Error logging is a very useful tool for postmortem analysis of program execution. By "error" we do not mean a program error that must be avoided at run time. Instead, the errors we refer to here are legitimate program states that need to be recorded by the application in a special way. For example, compilation errors generated when a compiler compiles a program, or logging of bank account overdrafts in a banking transaction (which might require rolling back of some program state in case overdrafts are prohibited for the bank account).

```
// transaction performing real work
atomic {
    // do some work
    if (error) {
        sendError(errorDescription);
        waitForResponse();
        // undo some work
    }
}

// transaction doing the error logging
atomic {
    errorDescription = waitForLoggingRequest();
    logErrorInLocalBuffer(errorDescription);
    notifyRequester();
}
```

Figure 5. Error condition reporting example. The upper half shows a transaction performing real work that reaches an error condition, which needs to be logged for postmortem analysis. sendError uses a CIT and communicators (embedded in errorDescription) to send the error description to the logger transaction. The communication from the worker transaction to the logger transaction can happen either using a producer-consumer queue as in the previous example, or using a dedicated channel between the worker transaction and the logger transaction. waitForResponse uses a CIT to wait on a xCondition that is used by the logger transaction to notify the waiter once it completes the logging (lower half of the pseudo code). We omit details because of space restrictions.

Programs that support error logging are sometimes configured in such a way that errors are logged by specialised logger threads, which need to be informed about error conditions by threads that perform real work [2]. Figure 5 illustrates an example transaction that does some work, then reaches an error condition, reports the error to a logger thread, waits for the logger thread to finish its logging, and then may need to "undo" some (not all) of the changes it made that may have lead to the error. Note that simply aborting the transaction is not desirable here because we would like to preserve some of its effects in spite of the error condition. Furthermore, the "partial undo" of the transaction is required to happen in the same transaction in order to preserve program invariants.

Both the described applications are impossible to code using the retry construct because it has the effect of the retrying transaction not executing at all (hence no request generated by the client or worker transaction in the first place). Furthermore, the punctuated transactions approach is problematic because waiting commits the part of the enclosing transaction completed so far, thus breaking the transaction into "transaction fragments", the collection of which are no longer guaranteed to execute as a single atomic unit (as was

orignally intended by the programmer). This may break program invariants in arbitrary ways, and re-enforcing them may be a non-trivial endeavour. xConditions and communicators provide a very convenient solution to both problems.

5. Implementation

We added support for xCondition in our transaction communicators framework that was implemented in the Maxine VM [9]. Briefly, transaction communicators were integrated in an STM that follows earlier STM implementations for managed-code environments [1, 6]. The STM does object level locking and conflict detection, and field level logging. Reads are invisible. Writes happen in place. Transactions maintain read, write and undo sets. The STM is implemented as a library, and not integrated with Maxine's compiler, and hence users must manually add instrumentation for transactional operations. For simplicity, we only support flat nesting of transactions, except for communicator-isolating transactions nested within ordinary transactions.

Communicators are implemented as wrapper classes around primitive types. Thus, the programmer can use communicators as individual fields in Java classes. CITs are implemented as closed nested transactions inside ordinary transactions. In addition to the typical locking/logging/validation/etc. operations on ordinary memory accesses, CITs also perform similar operations on communicators, via special getter and setter methods. Although implemented in a closed nested fashion, when they commit, CITs release ownership of all the communicators they accessed. This enables them to expose their effects on communicators even when the enclosing transaction is still active. The enclosing transaction however inherits the read and write "communicator-sets" of a committed CIT so as to be able to perform validation and roll back in case the enclosing transaction aborts. As CITs access communicators they build the dependency list of active transactions' whose effects on communicators were observed. This dependency list is transitively used by the outermost transaction in its 2-phase commit protocol to ensure that transactions respect all the dependencies, including the cyclic ones. For more details see [9].

We implemented the xCondition as a Java class with the aforementioned methods. We also leveraged the existing Java monitors API in the implementation of txwait, txnotify, and txnotifyAll. A call to txwait aborts the enclosing CIT (recall that xConditions can be accessed only within CITs) and forces the caller thread to wait on the target xCondition (done by calling Object.wait). Before waiting, the waiter also adds itself to a wait list inside the xCondition. The notifier uses this wait list to pick a waiter to notify. But before sending the notification, the notifier adds itself to the waiter's dependency list[3]. If the waiter happens to call txwait again, the last notification it received is forwarded to another concurrent waiter (see below). The corresponding dependency is also discarded. Note that txnotifyAll simply applies txnotify on all the waiters.

While it is still waiting, if the waiter becomes doomed to abort because a concurrent transaction conflicts with the waiter, it is woken up by the transaction (using a Java monitor notification). On waking up, the waiter revalidates itself to check if it must abort. If so, it aborts. If not, it goes back to wait on the same xCondition it was originally waiting on. Here the waiter has to be able distinguish between a notification received by a notifier and a notification received by a conflicting writer. This distinction is made by using a flag, called the waitFlag, in the waiter's descriptor. This flag is set by the waiter just before it waits on an xCondition, and is reset by the notifier during the notification. Thus, the waiter, when it is woken up, simply needs to check its waitFlag in order to determine if it was notified by a notifier.

As suggested in Section 3, in the event of aborting waiters, we need to be able to forward notifications (and establish corresponding dependencies) correctly. To do just that, we added a version number, version, to each xCondition, which is incremented during a txwait call. Each waiter logs all the xConditions (and their corresponding versions) it has waited on in its xCondition-list. In addition, each node in the xCondition-list also contains another version called the notifyVersion which indicates the version of the corresponding xCondition at the time the waiter received its notification. The idea here to to snapshot the "notification time", which is used in the notification forwarding process to determine if there exists a waiter that was waiting when the notification was originally delivered to the aborted waiter. If there were such waiters, the notification is forwarded to one of them. If there are no such waiters, the notification is (correctly) dropped. Notification forwarding comprises making the new waiter dependent on the original notifier, and explicitly notifying (via Java's notify call) the waiter.

6. Conclusion

In this paper we have presented a new way of doing condition synchronization between concurrent transactions that, instead of aborting or committing the waiting transaction as in prior proposals, enforces dependencies between waiter and notifier transactions – waiters always end up depending on notifiers. The dependency relation being transitive, circular dependencies can form, and force the participant transactions to commit or abort together. Our illustrative examples demonstrate that our condition variable, the xCondition, can be used very effectively, in conjunction with communicators, to help programmers easily express programming idioms that require synchronous communication between concurrent transactions, a feature that prior proposals did not provide.

[3] This is the same dependency list that is used to enforce dependencies between communicator accessing transactions. Hence no more additions were required in the implementation to enforce the waiter-notifier dependencies.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, 2006.

[2] H.-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism*, 2009.

[3] P. Dudnik and M. M. Swift. Condition variables and transactional memory: Problem or opportunity? In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[4] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 388–402, 2003.

[5] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[6] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 14–25, June 2006.

[7] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17:549–557, October 1974.

[8] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, Feb. 1980.

[9] V. Luchangco and V. J. Marathe. Transaction communicators: Enabling cooperation among concurrent transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 169–178, 2011.

[10] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 191–210, 2007.