Transaction Communicators: Enabling Cooperation Among Concurrent Transactions

Victor Luchangco
Oracle Labs
victor.luchangco@oracle.com

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

Abstract

In this paper, we propose to extend transactional memory with *transaction communicators*, special objects through which concurrent transactions can communicate: changes by one transaction to a communicator can be seen by concurrent transactions before the first transaction commits. Although isolation of transactions is compromised by such communication, we constrain the effects of this compromise by tracking dependencies among transactions, and preventing any transaction from committing unless every transaction whose changes it saw also commits. In particular, mutually dependent transactions must commit or abort together, and transactions that do not communicate remain isolated. To help programmers synchronize accesses to communicators, we also provide special *communicator-isolating transactions*, which ensure isolation even for accesses to communicators. We propose language features to help programmers express the communicator constructs.

We implemented a novel communicators-enabled STM runtime in the Maxine VM. Our preliminary evaluation demonstrates that communicators can be used in diverse settings to improve the performance of transactional programs, and to empower programmers with the ability to safely express within transactions important programming idioms that fundamentally require compromise of transaction isolation (e.g., CSP-style synchronous communication).

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

General Terms Design, Languages, Performance

Keywords Transactional Memory, Communication

1. Introduction

Multicore systems are making shared-memory multiprocessors ubiquitous, requiring concurrent programs to exploit their potential. But today's software engineers are ill equipped to write correct concurrent programs: the collection of tools and methodologies for cost-effective development of reliable concurrent programs is sparse. One fundamental problem is maintenance of the shared mutable state, used by threads to communicate in a shared-memory program. To ensure the integrity of this communication, various mechanisms have been devised for threads to synchronize their access to this shared state.

Transactional memory (TM) is a promising technology for facilitating concurrent programming by enabling shared-memory multiprocessors to be programmed in the transactional style, which has been so successful in database systems: A thread may execute a block of code as a *transaction*, which may either *commit*, in which case its effects become visible to other threads, or *abort*, in which case its effects are discarded. In a conventional database system, the only effects of consequence are those on the database. TM extends this protection to memory accesses.

A TM system ensures that transactions are *isolated*; that is, each transaction appears to execute without its operations being interleaved with operations of other threads. Furthermore, committed transactions appear to execute one at a time in some order, which we call the *transaction order*. We say the transaction order *explains* the execution.

Isolation can greatly simplify concurrent programming because it supports modular reasoning: a programmer can consider each transaction separately, rather than having to consider all the possible interleavings of its operations with the operations of other transactions. However, isolation also limits the applicability of transactions [16] because isolated transactions are incompatible with barriers, condition variables, and other common synchronization mechanisms and programming idioms that require communication among transactions while they are *active* (i.e., not yet committed or aborted). We believe that for transactional programming to reach its full potential, it must work with such techniques for programming concurrent systems.

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. c

In this paper, we propose *transaction communicators*, special objects through which concurrent transactions can communicate: changes to a communicator by one transaction can be seen by other transactions before the first transaction commits. Such communication compromises isolation because a transaction may see the effects of other transactions that have not committed (and indeed,

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0119-0/11/02. . . $10.00

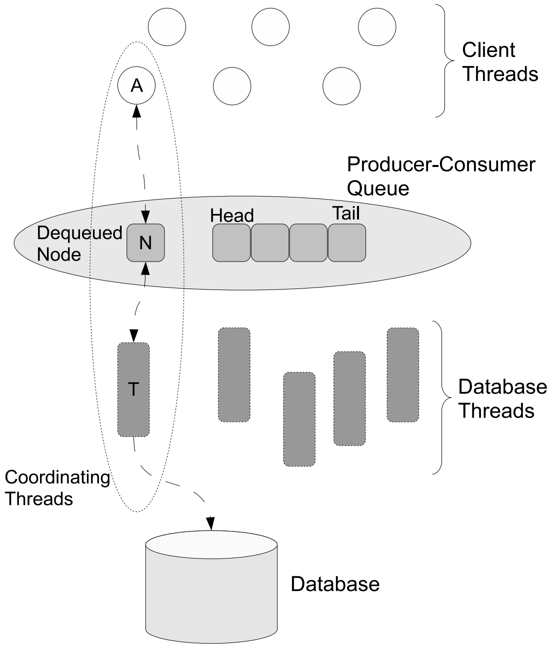

Figure 1. An example scenario where transactions may need to communicate with each other.

might abort rather than commit). We limit the impact of this compromise by preventing a transaction from committing unless every transaction whose effects it has seen also commits. Thus, two transactions that each see effects of the other transaction must either both commit or both abort. If they commit, they occupy the same position in the transaction order. Transactions that do not communicate (i.e., do not access any communicator) remain isolated.

Transaction communicators reintroduce a problem that transactions are intended to eliminate: the possibility of unsynchronized conflicting access to shared state. Although this problem is greatly reduced within transactions (such conflicts can occur only on communicators, which should be relatively rare and must be explicitly declared), programmers must synchronize access to communicators. To help programmers maintain a transactional style of programming, we provide a special *communicator-isolating transaction* that ensures isolation even for accesses to communicators.

We implemented a library-based prototype communicators-enabled STM runtime in the Maxine VM [21]. Although this prototype implementation imposes some restrictions on the use of communicators and does not include compiler support for language-level constructs, it embodies all the core ideas of communicators and also allows us to perform some preliminary experiments.

In summary, this paper makes the following contributions:

- We present the transaction communicator, a novel abstraction that permits restricted compromise of isolation of transactions to enable useful collaboration among them.
- We illustrate how to use communicators with several examples presented in a simple extension of Java with constructs to express atomic blocks and communicators.
- We describe a novel STM runtime infrastructure, in the Maxine VM, that supports communicators.
- We present preliminary evaluation of communicators to demonstrate their use in enabling interesting transaction communication idioms and in improving performance of TM applications.

2. Transaction Communicators

As described above, some useful synchronization patterns may require communication among concurrent transactions. In this paper, we propose the *transaction communicator*, a special object that enables desirable communication 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.

To guarantee this semantics, before a transaction T can commit, it must check whether all the transactions it depends on (i.e., those transactions whose effects it has observed) have already committed. If so, then T can also commit. Otherwise, T must wait until all the uncommitted transactions it depends on are ready to commit (if any of them abort, then T must also abort). Because there may be a cycle of dependencies in the set of transactions attempting to commit, the transactions cannot simply commit one at a time. Rather, the system must detect such cycles and commit all the transactions in any such cycle together.

In every other way, the semantics of transactions are unchanged. In particular, a transaction that sees the effects of another transaction on a non-communicator object must be ordered strictly after the other transaction. Thus, mutually dependent transactions that make conflicting accesses on non-communicator objects cannot be committed. Also, no committed transaction may see the effects of an aborted transaction.[1]

More precisely, instead of a total order of the committed transactions[2] to explain the execution, we require only a total order of "super-transactions", each consisting of one or more committed transactions that have no conflicting accesses on non-communicator objects. Every committed transaction must be in exactly one super-transaction. The super-transactions must appear to execute one at a time in the "super-transaction order", 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.

Note that based on only non-communicator accesses, transactions within a super-transaction commute. Thus, any total order of the transactions consistent with the super-transaction order explains the non-communicator accesses of the execution. That is, looking only at the operations on non-communicator objects, the transactions appear to be isolated. (Of course, doing the transactions in that order may have yielded different results for communicator operations, which may in turn have resulted in different operations on non-communicator objects being invoked.)

3. Communicator-Isolating Transactions

Because operations on communicators within transactions are visible to other transactions, transactions must synchronize access to communicators to use them effectively. Although it is possible to

[1] We do not specify the guarantees for aborted transactions, as there are several possibilities (e.g., [6, 9, 14]), and the choice is orthogonal to the issues addressed in this paper.

[2] We do not consider nested transactions for now because when transactions execute sequential code, as we implicitly assume in this paper, committed nested transactions can simply be *flattened* into the enclosing transaction.

use locks to do so, we prefer to maintain and encourage the transactional style by providing a new kind of transaction that also isolates accesses to communicators. We call these *communicator-isolating transactions* (CITs). The semantics is straightforward: All operations within a CIT, including accesses to communicators, appear to execute together without interleaving with operations of any other thread. Thus, a CIT can be used within an ordinary transaction to synchronize access to communicators, making it easy, as we illustrate in Section 4, to construct communicators of various data types using simple read/write communicators.

CITs isolate communicator accesses and ordinary transactions do not, so 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 within an ordinary transaction, can be flattened because no effects are made visible when the nested transaction commits.[3])

4. Illustrations

In this section, we illustrate the utility of transaction communicators (and CITs) with three examples. The first two examples show how use read-write communicators (i.e., communicators that support only read and write operations) to implement exchanger and queue communicators. The last example shows how a communicator can be used to improve scalability by turning transactional conflicts into dependencies.

We present pseudocode for these examples in an extension to the Java programming language with support for atomic blocks and read-write communicators in the form of specially designated fields.[4] Specifically, to an extension of Java with support for atomic blocks [10], we add two keywords: txcomm designates a field that ordinary transactions can use to communicate (i.e., the field is a read-write communicator), and txcommatomic designates a block of code that should be executed in a CIT. Note that a txcomm modifier on a reference declaration does *not* imply that the referenced object pointed to is a communicator. Transactions cannot communicate through that object unless it has one or more txcomm fields of its own.

4.1 Exchangers

Exchanger is a class of the Java concurrency library package whose instances serve as rendezvous points for threads to meet and exchange data objects. An *exchange* operation invoked within a transaction cannot complete successfully without violating isolation. In this section, we describe a simple communicator-based version of Exchanger, shown in Figure 2, that can be used by transactions.

The Exchanger declares two txcomm buffers (buf1 and buf2). A transaction that invokes the exchange method attempts to populate one of the two buffers and return the contents of the other.

The exchange method contains two txcommatomic blocks. The first is executed by all transactions to check and populate an available buffer (i.e., one that contains the value null). If neither buffer is available, the method simply returns null. The second txcommatomic block is executed only by the transaction that populated buf1. This block gets the second buffer's value and resets the values of both buffers. In each case, we use a txcommatomic block to provide atomic access to the two buffers.

```
class Exchanger {
    txcomm Object buf1;
    txcomm Object buf2;
    public exchange(Object myBuf) {
        Object otherBuf = null;
        txcommatomic {
            if (buf1 == null) {
                // populate the first buffer slot
                buf1 = myBuf;
            } else {
                if (buf2 == null) {
                    // second buffer slot available
                    buf2 = myBuf;
                    // return the buffer in first slot
                    return buf1;
                } else {
                    // both slots occupied, return null
                    return null;
                }
            }
        }
        // spin-wait for the second buffer to be populated
        while (true) {
            txcommatomic {
                if (buf2 != null) {
                    // second buffer finally populated
                    otherBuf = buf2;
                    buf1 = null;
                    buf2 = null;
                    return otherBuf;
                }
            }
        }
    }
}
```

Figure 2. Communicator-based exchanger

As is clear from the pseudocode, enforcing the dependencies is completely transparent to the programmer. Transactions that conduct an exchange on an instance of Exchanger become mutually dependent. A transaction T that observes that it cannot participate in an exchange (because both buffers of the exchanger are already populated), also depends on the transactions that last populated those buffers (i.e., the exchanging transactions), so if either of those two transactions aborts, then all three (including T) must abort. However, the exchanging transactions do not depend on T since neither sees any update by T. Because of this asymmetric dependency, the exchanging transactions need not abort if T does (perhaps because it could not participate in an exchange).

4.2 Producer-consumer queues

The producer-consumer queue is a widely used data structure that permits buffered communication between concurrent threads. We now present a communicator-based implementation (Figure 3) that can be used by transactions to communicate, as, for example, in the job-processing application described in the introduction.

The data structure consists of a simple linked list with head and tail references. A producer enqueues a new item by adding a node to the tail; a consumer dequeues an item by taking it from the node pointed to by the head. Threads/processes can concurrently access the producer-consumer queue. One particularly interesting part of the producer-consumer interaction appears in the boundary cases where the queue is either empty, or full (in case of bounded queues). When a consumer requests an item from the queue, if the queue is empty, some implementations force the consumer to *wait* for some producer to enqueue an item in the queue. Similarly, when a producer attempts to enqueue an item in a full queue, the producer may be forced to wait for a consumer to dequeue an item from the queue. In both cases, coordination between the producer and the consumer becomes crucial to avoid unnecessary unbounded waiting for both.

[3] Flattening is not always possible in systems that support user-visible aborts, but this issue is orthogonal to those discussed in this paper.

[4] We advocate a language extension rather than a library because of the limitations imposed by library-based STM systems in delivering the programmability promise of TM [5].

```
class ProducerConsumerQueue {
    txcomm Node head = null;
    txcomm Node tail = null;
    public void produce(Object data) {
        Node myNode = new Node(data);
        txcommatomic {
            if (tail == null) {
                head = tail = myNode;
            } else {
                tail.next = myNode;
                tail = myNode;
            }
        }
    }
    public Object consume() {
        Node node;
        while (true) {
            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;
                }
            }
        }
    }
}
class Node {
    txcomm Object data;
    txcomm Node next;
    public Node(Object data) {
        this.data = data;
    }
}
```

Figure 3. Communicator-based producer-consumer queue

Figure 3 illustrates a communicator-based unbounded producer-consumer queue. The head and tail fields of the queue, as well as the data and next fields of the node class are declared with the txcomm modifier. The producer uses a single txcommatomic block to enqueue a node in the queue. The consumer spin-waits on an empty queue, and uses a txcommatomic block to dequeue a node from the queue if the queue is not empty.

Each txcommatomic block is designed to avoid establishing unnecessary dependencies between transactions. For example, the producer checks the value of the tail field to determine if the queue is empty, whereas the consumer checks the value of the head field to do so. If the producer had checked head to see if the queue is empty, the check would have created a dependency on the last consumer (which modified head). We observe that, while designing a communicator-based data structure, the programmer should be careful to avoid extraneous dependencies between transactions due to access of txcomm fields. Tools to improve on this programmability aspect of communicators are a subject of future work.

4.3 Maintaining the size of a concurrent collection

Many collection implementations maintain a size field containing the number of items currently in the collection. Because this field is modified by each operation that inserts or deletes an item in the collection, it can be a contention bottleneck for the collection, severely restricting the scalability of transactional applications that use it. Communicators can be used to transform "low-level" memory conflicts on the size field into dependencies, improving scalability by reducing the number of transactions that must abort or wait.

```
class RedBlackTree {
  // fields
  ...
  // the size of the collection
  txcomm int size;
  // insert method
  boolean insert(Item item) {
    atomic {
      if (!lookup(item)) {
        // item not present, insert logic here
        ...
        // now increment the size of the collection
        txcommatomic {
          size++;
        }
        return true;
      }
      return false;
    }
  }
  // delete method
  boolean delete(Item item) {
    atomic {
      if (lookup(item)) {
        // item present, delete logic here
        ...

        // now decrement the size of the collection
        txcommatomic {
          size--;
        }
        return true;
      }
      return false;
    }
  }
}
```

Figure 4. Concurrent red-black tree with communicator size field

Figure 4 presents an implementation of a concurrent red black tree (much of the pseudocode elided for clarity) as a canonical example of a concurrent collection with a size field. The size field is implemented as a communicator (i.e., declared as a txcomm field), and is modified only inside txcommatomic blocks. This implementation essentially converts potential transactional conflicts on accesses to the size field into transactional dependencies, thereby improving scalability. We support our claim with some performance analysis in Section 7.

5. Discussion

5.1 Accessibility of communicators

As described above, communicators may be accessed directly within ordinary transactions. Although we can imagine a system in which they may even be accessed nontransactionally, allowing such access raises many well-known issues related to simultaneous transactional and nontransactional access [4]. We envision that communicators will be used sparingly to enable the kinds of communication patterns among transactions illustrated in Section 4, and so it is better to avoid those difficult issues by simply forbidding nontransactional access of communicators.

One might even argue that communicators should only be accessed within CITs, to avoid the problem of unsynchronized access to them. Indeed, for expedience, our prototype implementation (Section 6) only supports such access. However, allowing direct communicator access within ordinary transactions, particularly for read-only operations, eliminates some syntactic clutter. Of course, multiple such "naked" accesses to communicators are not guaranteed isolation—CITs are necessary to enforce such isolation.

5.2 Tracking dependencies

The exact dependencies induced by operations on communicators are subtle. For example, a transaction that reads a value depends on

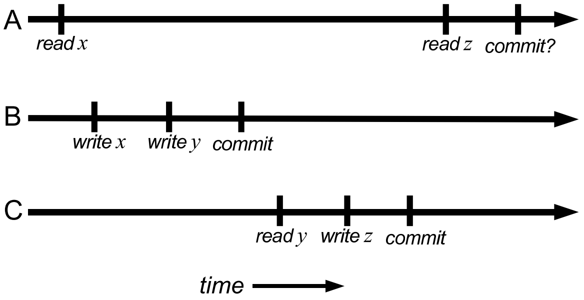

Figure 5. Execution illustrating the need for proper validation of communicators accessed by communicating transactions: x is a communicator; y and z are not. Transaction A must not commit: if it did, it must be ordered before or together with B (it sees the value of x prior to B's write), and after C (it sees the value C wrote in z), but B must be ordered before C (C sees the value B wrote).

the transaction that wrote that value, but writing to a communicator does not make the writing transaction depend on any other transaction, because the writer does not "see" any effects. Thus, we need only track *true* read-after-write dependencies, not *anti-* (write-after-read) and *output* (write-after-write) dependencies. We do not even need to track all read-after-write dependencies: a read sees only the effects of the most recent write, not those that precede it. And if the value it reads has been written more than once, then the transaction reading the value can commit as long as any of the transactions that wrote the value also commit. Determining a minimal set of dependencies is difficult (if not impossible).

Fortunately, it is not necessary to determine a minimal set of dependencies: as in dependence-aware STMs [3, 18], maintaining more dependencies than necessary may cause more transactions to abort, but preserves the semantics of committed transactions. The cost of more precise tracking of dependencies should be weighed against the benefit of aborting fewer transactions. We simply need to ensure that any extra dependencies do not interfere with progress guarantees by forming a cycle of dependent transactions that cannot be committed (for example, by having transactions that conflict on non-communicator objects).

5.3 Respecting the super-transaction order

Although a transaction may see changes to a communicator by other transactions in the same super-transaction, it must not see any changes by transactions in different super-transactions. Consider, for example, the execution depicted in Figure 5. When transaction A attempts to commit, the committed value of x is neither the value that A read nor a value written by a transaction in the same super-transaction as A (C must be ordered after B and before A and they cannot be in the same super-transaction as either because it accesses common non-communicator objects with each). Thus, A must abort.

We can guarantee this semantics by checking, when a transaction validates, that no other transaction has committed a change to a communicator it has read (i.e., the same check made for non-communicator objects). Alternatively, we could enforce anti-dependencies: a transaction cannot commit unless and until every transaction it has an anti-dependency on also commits.

5.4 Transaction conflicts

Concurrent accesses of communicators by transactions create dependencies between those transactions. In contrast, concurrent accesses of ordinary shared objects, where at least one access is a write, lead to conflicts between transactions, which in turn lead to transaction stalls and aborts. It is important to address the interaction of such transaction conflicts with communicator-enabled "transaction cooperation". Specifically, what happens when transactions that access communicators conflict?

A naive approach to handling conflicts is to simply stall or abort one of the conflicting transactions, as is often done in ordinary TM systems without communicators. However, when transactions may communicate, this approach may lead to deterministic livelocks. For example, one transaction may wait for an item to be produced by another (using a communicator-based queue, for example), and the producer may require an acknowledgment from the consumer (using another communicator). Thus, the producer and consumer are necessarily mutually dependent. If they also conflict on some non-communicator object, then they can never both commit, and so they will inevitably livelock (or deadlock, if the system stalls them both indefinitely). Therefore, we need a more sophisticated approach to handle such conflicts.

We believe that such deterministic conflicts between communicating transactions are a result of *programming error*, much as if a lock-based program allowed a deadlock. A pro-active solution would be to let the system provide the programmer with appropriate feedback to help diagnose and fix the error. When a transaction detects a conflict with a transaction that it depends on,[5] the system could throw a special CooperatorConflictException. This solution returns the responsibility of handling such "dependency-violating" conflicts to the programmer, where, we believe, it belongs. In most cases, we think the programmer will be able to identify the error and find a workaround, or else identify that the conflict as a nondeterministic event, which will eventually go away if the transaction is aborted and retried.

5.5 Programmability issues

Communicators allow cooperation between concurrent transactions by enabling programmers to build specialized transaction communication channels. However, communication channels are typically a *means*, not the *end*, of passing data between transactions; the communicating transactions are actually interested in the data that is communicated.

For instance, consider a typical producer-consumer interaction: The producer usually creates the data (which may include writing to the data) that it passes to the consumer. The consumer then reads (and possibly writes) the data it received from the consumer. If the data were an ordinary shared object, this programming pattern would lead to deterministic conflicts between the producer and the consumer transactions. To enable such interactions between concurrent transactions, communicated data objects must also be communicators. We expect that this will not be a problem in most cases: the communicated data objects will likely be immutable, in which case there is no danger of conflicts, or small objects with scalar fields.

When complex structures are communicated between transactions, the programmer must ensure that access of these structures will not lead to deterministic conflicts between the communicating transactions. Although this may be doable in select cases, in general, such sharing patterns would be too difficult for the programmer get right (avoid deterministic conflicts). Thus, we advocate that programmers use communicators to communicate just simple or immutable data objects. We believe that this simple guideline—enforced either by programming discipline (our current approach), or by the compiler and runtime system (a subject of future research)—will go a long way in making communicators an acceptable TM programming abstraction.

[5] We are assuming abort-on-exception semantics for our TM system.

6. Implementation

We now describe the STM we developed to evaluate communicators. We developed this STM in the Maxine VM [21], a freely available, open-source VM written almost exclusively in Java. The base STM follows earlier STM implementations for managed-code environments [1, 11]: It is object-based: transaction conflicts are detected at the granularity of Java objects. However, speculative writes are logged at the level of primitive types (e.g., references, ints, floats, etc.) for each field of an object. Transactions write in place, locking any object that they write. A transaction maintains an *undo log* with the previous values of speculatively overwritten values to be restored in case the transaction aborts. Reads are "invisible": a transaction maintains a *read set* to ensure consistency, but it does not access other transactions' read sets, and cannot determine what locations they have read. For simplicity, we only support flat nesting of transactions, except for communicator-isolating transactions nested within ordinary transactions, which must not be flattened to preserve semantics, as discussed in Section 3.

6.1 The STM runtime

Our STM is *library-based*: we have not modified the compiler to support the language-level constructs discussed in Section 2. Thus, we must manually add the instrumentation that a compiler would do automatically. For example, we must use beginTransaction and commitTransaction methods to delimit atomic blocks, and must explicitly "open" transactional objects before they can be accessed. Although this results in messier and more error-prone coding [5], it is simpler to implement and more flexible for the kind of preliminary experimentation we want to do. In particular, we can support multiple variants and extensions simultaneously without having to modify the compiler.

In addition to per-object metadata, we maintain a *transaction descriptor* that represents a transaction. The transaction descriptor contains fields that record its status (i.e., whether it has committed or aborted or is still active), its read set, its write set, and its undo log. (The write set is used to detect conflicts at object granularity; the undo log tracks speculative writes at fied granularity.) To support communicators, we introduce two new status values, Protected and Validated, which are used in the commit protocol described in Section 6.4. We also maintain separate read sets and undo logs for the communicators the transaction has accessed, and a list of transactions it depends on (depList).

To implement txcomm fields, we provide a communicator class for each primitive type (e.g., TxCommObject for reference types, TxCommInt for integer types, etc.), which the programmer uses to declare communicators. For example, the declaration

```
    txcomm int count;
```

is written in our communicator-enabled STM as:

```
    TxCommInt count;
```

6.2 Communicator-isolating transactions (CITs)

We implement a communicator-isolating transaction (CIT) as a closed nested transaction within an ordinary transaction. In addition to performing the usual locking/logging/validation/etc. operations on ordinary shared object accesses, a CIT performs similar operations on communicator accesses (using special get and set methods of the communicators) to guarantee their isolation from concurrent CITs (recall that in our implementation, communicators can be accessed only from CITs). The separate read and write sets and undo log for communicator accesses are used in this context. In particular, a CIT acquires exclusive ownership of all the communicators it writes, and shares (with concurrent CITs) read-only accesses to communicators by making these reads "invisible" to concurrent writer CITs, and validating the reads (using the communicator read set) when the CIT attempts to commit. This behavior is like that of closed nested transactions, with a key difference: when a CIT commits, it releases the exclusive ownership of all the communicators it modified (note that we implement flat nesting semantics for CITs nested within other CITs; thus the inner CIT's commit essentially becomes a nop). This enables subsequent communicator accesses, potentially done by concurrent transactions, to "see" the updates of the CIT. A committing CIT merges its communicator undo log entries in the enclosing (parent) transaction to enable rollback in case the transaction aborts.

6.3 Tracking depedencies

To track dependencies, each communicator has a lastWriter field, which points to the transaction that most recently wrote it. A transaction that accesses a communicator adds that communicator's last writer to its depList. Thus, our implementation tracks output dependencies as well as true dependencies.

To avoid the potential bad behavior exhibited in Figure 5, we maintain two version numbers: writeVersion, which every transaction that writes the communicator increments and stores in its write set; and commitVersion, which a committing transaction updates with the version number it stored in its write set (provided that the number in its write set is greater than the current commitVersion). When a transaction first reads a communicator, it also reads and logs the communicator's writeVersion in its communicator read set. Then, during its final validation (when it is committing), the transaction compares the commitVersion of each communicator that it read with the writeVersion that the transaction logged for that communicator in its communicator read set, and aborts if the former is greater than the latter. This check permits a reader of a communicator to commit only if no subsequent writer of that communicator has committed. This has the effect of indirectly tracking some anti-dependencies between transactions.

6.4 Transaction commits and aborts

To detect cycles of dependent transactions, we use two new status values, Protected and Validated, in the commit protocol. When a transaction T attempts to commit, it first switches its status to Protected. From this point on, no other thread may abort T directly. However, T must abort itself if its read set is not valid or if some transaction it depends on aborts.

After becoming Protected, T constructs the set S of all transactions it directly or indirectly depends on as follows: Starting with the transactions in its depList, remove any that are Committed, and for any Protected or Validated transaction A in S, add the transactions in A's depList. (A's depList will not change after this because A has begun its commit protocol.) If any transaction in S is Aborted then T also aborts.

When all transactions in S are either Protected or Validated (and their depLists have been added to S), then T validates its read set. If the validation fails, then T aborts. Otherwise, T changes its status to Validated, and waits for all the transactions in S to be Validated. (As before, committed transactions are removed from S, and T must abort if any transaction in S aborts.) When all transactions in S are Validated, T changes its status to Committed. Because our STM writes in place, nor further clean-up is needed.

When a transaction T aborts, we must roll back its speculative writes. For any object that only T has written, this is straightforward, because T stored the value it overwrote in its undo log. However, if T wrote a communicator, its abort may lead to a cascade of aborts by other transactions that directly or indirectly depend on T, and several of these transactions may have written the same communicator. In this case, we must be careful to restore the correct value. To that end, we implemented the following conservative solution: T waits for the transaction that wrote the communicator immediately before T—we call that transaction T's *predecessor*—to either commit or abort. If T's predecessor commits, then T restores the value written by the predecessor. On the other hand, if T's predecessor aborts, T should not do its rollback (which would restore the value written by its aborted predecessor).

7. Evaluation

We now describe our experience using communicators in three benchmarks. The first benchmark is a transactional red-black tree with a size field that contains the number of nodes in the tree. This benchmark illustrates how communicators can improve scalability by transforming transaction conflicts into dependencies. The second benchmark was inspired by a specific component (to be elaborated later) of SPECjbb2005 [20], an industry-standard JVM evaluation benchmark. This benchmark illustrates the use of communicators to enable parallelism in large transactions. The third benchmark is a variant of the job-processing example from Section 1. This benchmark illustrates communicators as enablers of explicit synchronous communication between concurrent transactions. Our results also give insight into the performance characteristics of our implementation. Together, they demonstrate that communicators are an effective tool for helping programmers express a diverse range of programming idioms that were impossible to express, or required specialized solutions (e.g., open nesting [17], irrevocability [23], etc.).

Our experiments were conducted on a multiprocessor system consisting of eight 2.8GHz Quad-Core AMD Opteron$^{\text{TM}}$ chips. Thus, the system supports 32 hardware threads.

7.1 Red-black tree with size field

Concurrent collections are frequently used in parallel programs, and TM runtimes are often evaluated on concurrent collections such as red-black trees and hash tables. However, the collection implementations do not accurately reflect versions used in real systems, which support a size() operation that returns the number of items currently in the collection. Typically a collection implementation maintains a size field, which is modified appropriately when items are inserted into or deleted from the collection. Accesses to the size field of a collection may significantly increase contention between transactions performing insert/delete operations even on disjoint regions in the collection.

Figure 6 illustrates our point: The red-black tree with an ordinary size field does not scale beyond 4 threads—updates to size quickly become a contention bottleneck, and throughput drops significantly as the number of threads increases. In these tests, we used the Polite contention manager [12], which employs exponential backoff before aborting the conflicting transaction. We believe that no contention manager will perform significantly better because updates to size become a serialization bottleneck.

Making the size field a communicator (much like in the code in Figure 4—no other fields or objects are communicators, so transactional accesses of non-size fields of the red-black tree and its nodes are subject to ordinary TM conflict detection and management policies) dramatically improves performance: This implementation scales much better, deteriorating in performance only slightly after peaking with about 10 threads. The communicators infrastructure imposes a latency cost of approximately 15% at low thread counts, but this loss is more than made up for by scalability as the number of threads increases.

7.2 Emulating intra-transaction concurrency

Concurrency within transactions has been advocated by researchers as a means to improve performance of large transactions [22]. Communicator can be used to emulate large transactions with internal concurrency, and thereby achieve the same performance improvements, by combining several transactions into super-transactions.

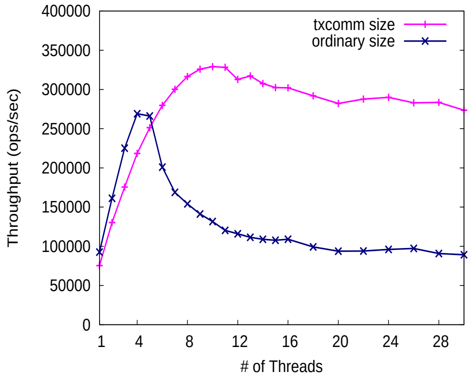

Figure 6. Throughput results of test runs on Red-Black implementations that contain the size field. The ordinary size graph depicts the throughput in the case where the size field is an ordinary shared object. The txcomm size graph depicts the throughput in the case where the size field is a implemented as a communicator. In both implementations, the red-black tree can contain up to 128K nodes, and usually averages to about 64K nodes. Each thread repeatedly executes one of the insert/delete/lookup operations on the red-black tree, with a 10%/10%/80% distribution respectively. Each operation of the tree is executed in a transaction. The performance numbers reported here were averaged over 3 runs of 30 seconds each.

For example, communicators can be used to emulate master-slave parallelism within a transaction: A master transaction initiates slave transactions to do computations on their respective data partitions (shared data is partitioned between the slave transactions), and then waits for all the slaves to finish. (In the communicators-based equivalent of master-slave parallelism, programmers must be careful to avoid any overlap in the data partitions that may lead to deterministic conflicts between the slaves.) We demonstrate this potential with a benchmark inspired by our observation of SPECjbb2005, an industry-standard benchmark used to evaluate server-side JVMs. It emulates a 3-tier client/server application. The tiers are for input selection, middle tier business logic, and backend computation that represents a database in the form of a set of Java collections (TreeMaps and HashMaps).

The application simulates a store-management system, in which the store consists of a group of warehouses (each controlled by a distinct thread), and each warehouse contains an inventory of different types of items, a customer pool, a supplier pool, etc. Seven types of SPECjbb "transactions" represent typical activities that appear in such applications: new orders, stock updates, customer updates, etc. One of these transactions—the DeliveryTransaction—does bulk deliveries for all possible outstanding orders within a warehouse. This is a big transaction: it walks through the entire HashMap of new orders, fulfilling any outstanding orders it encounters. Although this transaction is executed infrequently (about 3% of the SPECjbb transactions are deliveries), it consumes a large portion of the application's running time (approximately 50% in our experiments). We believe that such heavyweight operations are a good target for intra-transaction concurrency.

To that end, we have developed a benchmark that mimics just the delivery transaction of SPECjbb. The benchmark consists of a pre-populated, 32K entry, hash table. A transaction (not the SPECjbb "transaction") can invoke an enumeration operation on the hash table, which walks through all the items in the table and randomly updates their values. The hash table is split into disjoint partitions, each of which is processed by a slave transaction.

```
class MasterSlaveCoordinator {
  // flag used by the master to instruct the slaves
  // to process their data partitions
  txcomm boolean go;
  // the count indicating how many slaves have
  // completed their task
  txcomm int slavesDone;
}
```

Figure 7. The master instructs the slaves to process their hash table partition by setting the go flag. Each slave increments the slavesDone counter once it has processed its hash table partition, and then attempts to commit. The master transaction spin-waits for the slavesDone counter to become equal to the total number of slaves, and then attempts to commit. All coordination is done within nested txcommatomic blocks.

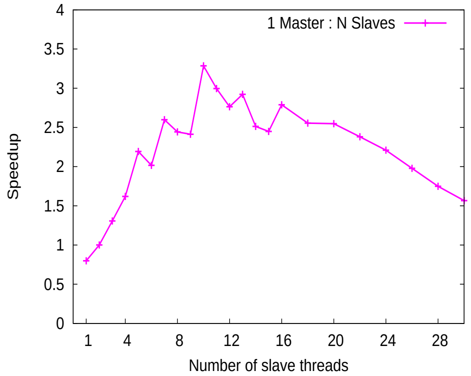

Figure 8. Speedup (over single-threaded transaction runs) results of our intra-transaction concurrency improvement benchmark. The performance results were averaged over 3 test runs. The hash table used had a maximum capacity of 32K entries, with the table being approximately half full at all times of the test runs.

After initializing the hash table (to about half the capacity of 32K), the master thread repeatedly runs enumeration transactions (we ran a total of 50,000 transactions for each test). In the non-concurrent version, the master thread itself invokes the enumeration. In the concurrent version, the master instructs its slaves to process their respective hash table partitions, and then waits for all the slaves to finish. The coordination between master and slaves uses two communicators, shown in Figure 7.

Figure 8 shows the speedup of our communicator-based solution over the single-threaded version. The cost of using communicators, as seen in the 1-slave configuration, is approximately 20%. As the number of threads increases, our parallelized version delivers up to 3.2x speedup, with throughput peaking with 10 threads, before slowly deteriorating. Figure 9 shows that this deterioration is caused by the increasing costs of enforcing dependencies between transactions: The cost of the commit protocol, fueled by the complex transaction dependencies, increases significantly with more slave transactions. With 4 slaves, over 40% of the time is spent in the commit protocol. With 28 slaves, only 15% of the time is spent in doing useful work, and most of the remaining time (about 65%) is spent in the commit protocol. This highlights the need to improve the commit protocol in our system. Note that virtually no transaction aborts in this benchmark (less than 0.5% time is spent in aborted transactions).

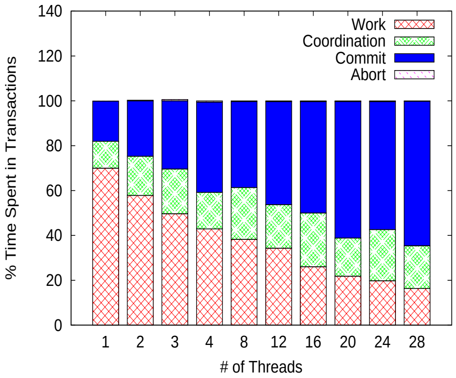

Figure 9. Distribution of percentage of time spent in a transaction. Work refers to the amount of time spent doing real work (enumerating through the hash table partitions); Coordination refers to the time spent in coordination between the master and the slave threads; Commit refers to the time spent committing the interdependent transactions together; and Abort refers to the time spent executing transactions that eventually abort.

7.3 A client-server application

We developed a simple variant of the job-processing example from Section 1, in which a client requests a server to execute a task, and the server returns a response. The request and response occur through communicators, so the client and server operations appear to happen together, as a super-transaction. Shared data is partitioned between client and server transactions. All client transactions access a shared client-side hash table, and all the server transactions access a shared server-side hash table. Local computations are simulated using idle spin loops. Client and server threads are paired together during initialization, and communication happens only between these pairs.

Each client thread repeatedly executes client-side transactions. A client-side transaction (i) executes an operation (insert, delete, or lookup) on the client-side hash table, (ii) posts a request to the server-side transaction (this request instructs the server-side to execute an operation on the server-side hash table), (iii) does some local computation (including a 100-microsecond idle loop), (iv) waits for the response from the server-side, and finally, (v) based on the response (which is a success/failure flag), executes another operation on the client-side hash table.

Each server thread also repeatedly executes server-side transactions. A server-side transaction (i) waits for a request from the client-side, (ii) does some local computation (a 100-microsecond loop), (iii) performs the requested operation on the server-side hash table, and finally (iv) posts the response of the hash table operation.

The data structure used by the client and server transactions to communicate is depicted in Figure 10.

Unlike previous benchmarks, there is no obvious implementation to compare against: Without communicators, client and server transactions cannot communicate. One alternative approach to the problem is to use *irrevocable* transactions [23], where a client-side transaction becomes irrevocable when it posts a request, and instead of using transactions, server-side threads use alternate synchronization mechanisms (e.g., locks) to ensure atomicity of the server-side operations. Thus the client-side transaction, after it becomes irrevocable, can interact with a server-side thread. Although workable, irrevocability does not scale. Furthermore, replacing transactions with a different synchronization mechanism on the server-side may be a nontrivial endeavor. Nonetheless, we compare this alternative with our client-server benchmark.

Figure 11 shows the throughput results of our benchmark with 1 through 16 client theads (and the same number of server threads). Clearly the irrevocability-based solution does not scale and our

```
class ClientServerAnchor {
  // request type: insert, delete, lookup
  txcomm int requestType;
  // request hash table key
  txcomm int requestKey;
  // response: indicates the success/failure of
  // requested operation
  txcomm boolean response;
  // flag to indicate if the client request is ready
  txcomm boolean clientRequestReady;
  // flag to indicate if the server response is ready
  txcomm boolean serverResponseReady;
}
```

Figure 10. The client places its request in the requestType and requestKey fields (which means that it wants the server transaction to execute the operation of type requestType for the key requestKey on the server-side hash table). The client signals the server to proceed by setting the clientRequestReady flag. Similarly the server transaction places its response in the anchor's response field, and signals the client by setting the serverResponseReady flag.

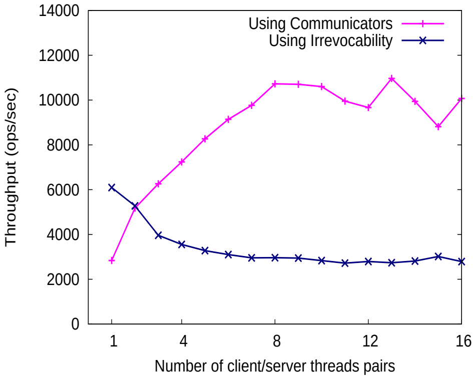

Figure 11. Throughput results of the client-server application. Results are averaged over 3 test runs, each running for 30 seconds.

communicator-based solution scales well (by a factor of almost 4). However, the irrevocability-based solution outperforms our communicator-based solution in single-thread runs (by a factor of about 2), because of the high overheads associated with the coordination of operations on communicators, and the high-latency commit operation. Furthermore, the server-side hash table in the irrevocability-based solution is the high-performance concurrent HashMap from the java.util.concurrent package, which is much faster than our transactional hash table.

Notice that the transactions for which we present performance results here are quite big (more than 100 microseconds long). We experimented with shorter transactions as well and found that the coordination/commit overhead for communicators was too high for our approach to outperform the irrevocability-based approach, again emphasizing the need for to improve the performance of our communicator infrastructure.

8. Related work

The idea of relaxing isolation to allow concurrent transactions to cooperate was embodied in our earlier work on *synchronizers* [16]. However, synchronizers enforced bidirectional dependencies between communicating transactions, which introduced several difficulties. For instance, a set of transactions about to commit could be forced to abort by another transaction that belatedly reads or writes a synchronizer that the transactions accessed. As a result, idioms such as synchronous exchange channels (exchangers) cannot be easily implemented with synchronizers. Suggested workarounds place a significant burden on the programmer.

Tracking dependencies among transactions is not original to our work. Rather, it has been proposed as a way to reduce aborts in transactional memory implementations [3, 18]. However, those systems maintain isolation, so transactions with cyclic dependencies must all abort, whereas with communicators, such transactions may commit (as long as they all commit).

Transactional events [7] enable composition of synchronous communication between threads by adding transactional all-or-nothing semantics to sequences of communication events. The communication messages are sent by threads via "event synchronization transactions". The fates of such synchronously communicating transactions are joined in that they commit or abort together. Composing synchronous communication operations significantly simplifies idioms such as 3-way synchronous operations. Transactional events appear to be similar to synchronizers, though their use is targeted toward a smaller problem domain—composition of multiple synchronous send/receive operations to simplify otherwise complex synchronous communication protocols.

Accesses to communicators, particularly when done within a nested txcommatomic block, can be viewed as a form of *open nesting* [2, 17], because updates made to txcomm fields within the txcommatomic block become visible to concurrent transactions once the txcommatomic block commits, even if the outer atomic block has not yet committed. However, there is a crucial difference: we track dependencies between transactions due to communicator accesses, and the system ensures that a transaction cannot commit unless all transactions that it depends on also commit. Furthermore, if the transaction for the outer atomic block aborts, the effects of the nested txcommatomic transaction are also rolled back. In contrast, the effects of a committed open-nested transaction are not rolled back; instead, the programmer must provide "compensating actions" to reverse these effects.

The TIC model [19] also attempts to enable cooperation between concurrent transactions by introducing the Wait primitive that "punctuates" the calling transaction. Punctuation breaks the isolation of a transaction by splitting it into two distinct transactions, one before and one after the Wait call. To address the violated isolation, TIC proposes several new constructs and type system extensions to "expose" the violation to the calling contexts, and to help the programmer restore program invariants broken during such violations. This model of punctuation and restoration is somewhat similar to open nesting and compensating actions, albeit with better type system support. However, the complete lack of isolation between the two halves of a punctuated atomic block can present significant programmability challenges in some contexts (e.g. the job scheduling problem from Figure 1). Dudnik and Swift [8] take a similar punctuate-on-wait approach in their condition variable proposal. They do not, however, support mechanisms to propagate punctuation information and enable restoration of broken program invariants, as TIC does.

Recently, and independently, Lesani and Palsberg developed *transactors* [15], which combines the actor model with transactions. Transactors can send and receive messages within transactions, which create dependencies between transactions by different actors. As with communicators, transactions cannot commit unless the transactions they depend on commit, and cyclic dependencies require all transactions in the cycle to commit or abort together.

9. Conclusion

Although transactional memory is a promising mechanism to help address the challenge of writing concurrent programs, we believe that the scope of transactional programming will be limited unless it works with commonly used programming idioms and concurrency control techniques that are incompatible with isolation of transactions. To that end, we have presented transaction communicators, which enable programmers to have concurrent transactions communicate in a controlled fashion, sufficient to enable programming idioms such as CSP-style synchronous communication [13], producer-consumer interactions, etc.

Our preliminary evaluation demonstrates that communicators can be useful in a wide range of settings—to alleviate contention bottlenecks, to break down and parallelize large transactions, and to perform explicit inter-transaction communication. We believe that communicators open a new research frontier for transactional programming. In addition to the programming idioms explored in this paper, we expect to find many more useful and interesting ways of programming TM applications using communicators (e.g., condition variables, enforcing transaction ordering, etc.).

Acknowledgments

We are grateful to the Maxine team for their technical support, to Dhruva Chakrabarti and the anonymous reviewers for their many helpful questions and suggestions.

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] K. Agrawal, C. E. Leiserson, and J. Sukha. Memory Models for Open-Nested Transactions. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2006.

[3] U. Aydonat and T. Abdelrahmen. Serializability of transactions in software transactional memory. In *TRANSACT*, 2009.

[4] C. Blundell, E. C. Lewis, and M. Martin. Deconstructing Transactions: The Subtleties of Atomicity. In *the 4th Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2005.

[5] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and Limitations of Library-Based Software Transactional Memory in C++. In *2nd ACM SIGPLAN Workshop on Languages, Compilers and Hardware Support for Transactional Computing*, 2007.

[6] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, To appear. Earlier version appeared in Refine 2009.

[7] K. Donnelly and M. Fluet. Transactional Events. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 124–135, 2006.

[8] 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.

[9] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Morgan Claypool, 2010.

[10] 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.

[11] 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.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[13] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[14] D. Imbs, J. R. Gonzalez de Mendivil, and M. Raynal. Brief announcement: Virtual world consistency: A new condition for STM systems. In *Proceedings of the 28th Annual ACM Sympoisium on Principles of Distributed Computing*, 2009.

[15] M. Lesani and J. Palsberg. Communicating Memory Transactions. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2011.

[16] V. Luchangco and V. J. Marathe. Transaction Synchronizers. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages*, 2005.

[17] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[18] H. E. Ramadan, I. Roy, and E. Witchel. Dependence-Aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 246–257, 2008.

[19] 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.

[20] Standard Performance Evaluation Corporation (SPEC). SPECjbb2005 Java Server Benchmark. http://www.spec.org/jbb2005/.

[21] The Maxine Virtual Machine. http://research.sun.com/projects/maxine/.

[22] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *The 23rd European Conference on Object-Oriented Programming*, pages 123–147, 2009.

[23] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and Their Applications. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 285–296, 2008.