Transactions with Isolation and Cooperation

Yannis Smaragdakis Anthony Kay Reimer Behrends Michal Young

Department of Computer and Information Science
University of Oregon
Eugene, OR 97403-1202
{yannis,tkay,behrends,michal}@cs.uoregon.edu

Abstract

We present the TIC (Transactions with Isolation and Cooperation) model for concurrent programming. TIC adds to standard transactional memory the ability for a transaction to observe the effects of other threads at selected points. This allows transactions to cooperate, as well as to invoke non-repeatable or irreversible operations, such as I/O. Cooperating transactions run the danger of exposing intermediate state and of having other threads change the transaction's state. The TIC model protects against unanticipated interference by having the type system keep track of all operations that may (transitively) violate the atomicity of a transaction and require the programmer to establish consistency at appropriate points. The result is a programming model that is both general and simple. We have used the TIC model to re-engineer existing lock-based applications including a substantial multi-threaded web mail server and a memory allocator with coarse-grained locking. Our experience confirms the features of the TIC model: It is convenient for the programmer, while maintaining the benefits of transactional memory.

Categories and Subject Descriptors C.5.0 [*Computer Systems Implementation*]: General; D.1.3 [*Programming Techniques*]: Concurrent Programming—parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—concurrent programming structures

General Terms Design,Languages

Keywords transactional memory, nested transactions, open-nesting, TIC, punctuation

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

1. Introduction and Motivation

Transactions as a programming language construct have been proposed to simplify concurrent application programming and avoid programming errors. Complexity and error proneness in conventional concurrent programming models is a consequence of the essential non-locality of reasoning about lock (or monitor) acquisition order and condition signaling. Transactional programming (whether supported by a hardware transactional memory, software transactional memory, or source-to-source translation to conventional locking code) aims to reduce the programmer's burden to a single, local design decision about which sequences of actions should execute "as if atomic."

To relieve the programmer from non-local reasoning about concurrency, the programming model presented by transaction support must have certain essential properties:

- It must compose, in the sense that the decision to make a region of code in one method transactional is independent of whether called or calling methods are transactional. This further implies that any restrictions or special conditions on enclosing a method call in a transaction should be tracked by the type system, rather than requiring a programmer to inspect source code of other classes.
- It must be general enough that common operations can be (transitively) enclosed in transactions, and that common concurrent programming idioms can either be used unchanged or have suitable replacements.

Existing proposals for transaction support still fall short of these requirements. For example, atomic sections commonly need to roll back and undo their effects. (An optimistic concurrency control implementation needs to roll back when it detects interference from another transaction. A pessimistic implementation needs to roll back, in order to avoid deadlock, when it fails to acquire a lock.) Roll-back is incompatible with operations that cannot be undone, such as I/O. Although I/O can be automatically buffered (e.g., put off until the end of a transactional section) [27], this is not always consistent with the program's logic, particularly when a read follows a write. Thus, performing irreversible operations becomes a global property in transactional systems: A

transaction needs to be aware of irreversible operations in all methods it calls, directly or indirectly. Past solutions to this problem have been draconian: Either irreversible operations have been completely disallowed in atomic sections [29], or concurrency is disabled and only a single transaction with irreversible operations can be run at a time [10, 11].

A related problem is that of transaction nesting. The most reasonable semantics is *closed-nesting*: The effects of a nested transaction are not externally visible until the outermost transaction completes. However, this approach is not modular. Transactional code that needs to communicate its results to the outside world needs to be careful to avoid being used inside another transaction. This makes thread cooperation harder. Transactional models often allow depending on external conditions through guards for atomic sections [28] or through plain use of a `retry` statement that restarts the transaction, undoing its current effects [29]. These approaches suffice only at the outermost level of nested transactions. Harris and Fraser [28] propose evaluating all guards of inner transactions at the outermost level. An alternative is to restart the outermost transaction if an inner transaction's guard fails. Both approaches are insufficient for allowing threads to cooperate. For instance, consider the simplest thread coordination—a barrier:

```
void barrier() {
  atomic { count++; }
  atomic(count == NUMTHREADS) {
    /* barrier reached */
  }
}
```

(We use a Harris-and-Fraser-like Java extension for this example—note the Conditional Critical Region syntax of the second atomic section.) The guard of the second atomic section will only become true when *all* threads have finished the first atomic section. Yet if we want good composability properties, the barrier itself should be usable inside an atomic section. Such use can possibly be completely accidental, in an atomic section protecting other, unrelated data from interference:

```
atomic { ... barrier(); ... }
```

For ease of exposition, we show the result of inlining the barrier code, so that the nesting becomes syntactic:

```
atomic {
  ...
  atomic { count++; }
  atomic(count == NUMTHREADS) {
    /* barrier reached */
  }
  ...
}
```

It is hard to see what would be a reasonable semantics for the atomic sections in this case. Clearly, the Harris and Fraser approach [28] of evaluating nested guards at the outermost transaction's level does not apply. The outer atomic section cannot block at entry until the condition of the inner section becomes true—the condition itself depends on the execution of the first inner atomic section. Restarting the outermost transaction upon reaching the unsatisfied inner guard does not work either—this would undo the effect of the first inner atomic section, making the guard unsatisfiable. Indeed, the word *atomic* itself seems a misnomer in this case: The intended behavior is that execution of the outer atomic section will *not* be atomic, but will instead be interrupted, allowing other threads to observe its results, and itself observing the results of other threads at the point of evaluation of the inner atomic section's guard. This is the essence of thread cooperation: We want to allow observing the results of other threads in a controlled manner, even if the code happens to be called inside an atomic section. I/O can even be viewed as a special case of a cooperation pattern with the cooperating thread provided by the I/O device.

In this paper, we present a concurrent programming model that attempts to allow freedom in how the code is structured, while handling uniformly both irreversible operations and thread cooperation. We call the model *TIC* for *Transactions with Isolation and Cooperation*. TIC allows temporarily suspending a transaction's atomicity and isolation properties in the middle of an `atomic` block. The type system tracks all occurrences of such suspending operations and ensures that the user provides a way to recover from them. For instance, our barrier example can be written as:

```
void barrierTIC() {
  atomic {
    count++;
    Wait(count == NUMTHREADS);
  }
}
```

The `Wait` keyword is a TIC addition to a conventional Transactional Memory (TM) programming model. It signifies that transactional isolation is suspended until the condition of the `Wait` statement (in this case "`count == NUMTHREADS`") becomes true. Based on our previous discussion, the interesting case is that of a `Wait` that does not occur lexically nested inside an `atomic` section, but in a transitively called routine. In this case, TIC dictates that the type system keep track of operations that possibly `Wait`. All such operations need to occur inside a block of code designated by "`expose(<methodcall>) establish <stmt>`". Thus, our `barrierTIC` routine can be called inside an atomic section as:

```
atomic {
  ...
  expose(barrierTIC()) establish { ... } ;
  ...
}
```

The statement block following the `establish` keyword is responsible for re-establishing local invariants that the rest of the atomic section expects at this point in the execution.

Importantly, the type system does not allow invocations of `barrierTIC`, unless under an `expose ... establish`, even if these invocations occur transitively. That is, if a method `foo` calls a method `bar`, and `bar` in turn calls `barrierTIC`, then both the call to `barrierTIC` and the call to `bar` need to be under `expose ... establish` expressions. (We discuss later how this requirement is relaxed with a checked programmer-supplied type annotation.)

The above example showcases some of the main elements of the TIC model. Later sections define the full model more precisely. Note that TIC is orthogonal to many performance and implementation considerations. TIC is a programming model, not an implementation technique, and is intended to be compatible with a variety of different implementation techniques with different performance characteristics and demands on program analysis. In order to evaluate the expressiveness and ease of use of the TIC model for realistic concurrent applications, we produced initial implementations based on Harris and Fraser's libstm [28] and Dice et al.'s TL2 [17].

Overall, the TIC behavior can be emulated with traditional atomic transactions, but not without significant code reorganization. A reasonable way to view the benefits of TIC is as a way to relax the severe restrictions of existing techniques, while maintaining the assurance that the programmer has considered the impact of transaction suspension. Other authors (e.g., Carlstrom et al. [11]) have advocated mapping condition-variable `waits` in lock-based code to transaction suspension, but with no type system warning of this behavior to the authors of surrounding transactions.

In short, our paper makes the following contributions:

- We define the TIC concurrent programming model that extends transactions to also allow thread cooperation. TIC retains all the benefits of traditional transactions in the common case of `atomic` sections that do not `Wait`. At the same time, it offers a single, uniform mechanism that allows both operations that `Wait` and operations that perform irreversible actions to be used inside transactional code, while enabling the transaction to recover from inconsistencies.
- We demonstrate the simplicity of the model with several examples. The type system support of the TIC model was found to be useful for TIC implementations but also for detecting errors in lock-based code (e.g., locks held during high-latency network operations).
- TIC offers a more disciplined alternative to many uses of *open-nested* transactions. For instance, a long-running operation can be delegated to a different thread, with `Wait` used for inter-thread coordination. Nevertheless, the main TIC features are complementary to open-nesting. Indeed, our current formulation of TIC integrates open-nesting ideas to cover some interesting cases. We allow nested transactions to be open, if they occur in a method that specifies compensating actions.
- We offer an optimistic implementation of the TIC model. We use this implementation to evaluate the model in practice for realistic multi-threaded applications.

The rest of this paper is organized as follows. We first present an overview of the TIC model (Section 2) and then discuss our experience with implementing concurrent applications with TIC (Section 3). We describe in more detail the TIC type system and implementation in Section 4. Section 5 discusses the TIC nesting model, as well as connections between TIC and open-nested transactions. We then contrast with related work (Section 6) and conclude (Section 7).

2. Isolation and Cooperation

Hell is a place where people live in pairs,
tied back-to-back so they can't see each other's face.
– Eastern European popular tradition.

Hell is other people.
– Jean-Paul Sartre.

We next describe in more detail the TIC programming model. There are two interesting cases that TIC intends to handle similarly: transactions that `Wait` and transactions that call external operations incompatible with transactional semantics. Such operations typically access some resource outside the control of the transactional memory system—e.g., I/O—and either directly expose their results to other transactions, or are irreversible. We begin our discussion with the case of transactions that call `Wait`.[1]

2.1 Transactions that Wait

We discuss the TIC model in the context of an *idealized* extension of the Java language for simplicity of exposition. The same principles can be adapted to other languages and different integration techniques. Indeed, as we describe later, our actual prototype is a C++ library which hides many of the finer details but still requires manual code instrumentation.

The syntax of a TIC transaction (using the conventions of the Java Language Specification [22]) is:

AtomicStatement:
`atomic` *Statement*

An *AtomicStatement* is a Java statement and in practice the statement that follows keyword `atomic` is usually a composite statement (block):

`atomic {` *statements* `}`

Atomic sections can nest arbitrarily, both in lexical and in dynamic scope (i.e., an atomic section can include calls to methods that include other atomic sections). We follow by default a *closed-nesting* semantics, where nested trans-

[1] We use the term "to call `Wait`" for convenience, although `Wait` is a statement.

actions do not truly commit until the outermost transaction does.

For the most part, our idealized Java extension matches the decisions of the Harris and Fraser design [28]. E.g.,

- Our transactions have exactly-once execution semantics.
- All program data can transparently participate in a transaction.
- All regular Java control flow (including method returns and exceptions) can be used inside a transaction and result in normal termination (i.e., commit) of the transaction.

Nevertheless, whereas Harris and Fraser implement Hoare's Conditional Critical Regions (CCRs) [35] programming model, we do not support conditional atomic sections, as these are supplanted by our `Wait` concept. An atomic block can call a `Wait(`*Expression*`)` operation, with a boolean *Expression* representing the condition of `Wait`. This has the effect of checking the condition, and, if it is not true, committing the current transaction and suspending the thread executing it until the condition becomes true.[2] Once the condition becomes true, the transaction restarts from the statement following `Wait`. For all purposes, the transaction before the execution of a `Wait` and that after it are two separate transactions—we call them the *top* and *bottom* transactions, relative to each `Wait` statement. The starting point of the bottom transaction (e.g., for re-trying purposes) is the `Wait` statement, no matter where this is found in the program. We use the terms *suspending* and *resuming* the transaction for committing the top transaction and beginning the bottom one. We also use the term *punctuating* the transaction for the overall effects of calling `Wait`.

Clearly, having a `Wait` operation is unnecessary if it is unconditionally and directly called inside a single atomic statement. For instance, a section such as

```
atomic {
  statementBlockA
  Wait(condition);
  statementBlockB
}
```

can be written equivalently with a CCR:

```
atomic { statementBlockA }
atomic (condition) { statementBlockB }
```

The benefit of having an explicit `Wait` statement is that it can be used at any point in a transaction. Thus, a `Wait` could be invoked nested deeply inside a conditional statement, or from a method called transitively from other methods invoked inside an `atomic`. A `Wait` statement punctuates all the nested transactions in whose dynamic scope it occurs, and not only its directly enclosing transaction.

We have already seen in the Introduction an example where the use of `Wait` allows transactions to cooperate. `Wait` adds power to atomic sections, but it violates transactional semantics. Not only can a transaction that `Wait`s observe the results of other transactions (loss of *isolation*) but it also exposes its own intermediate results to other threads (loss of *atomicity*). This is an unavoidable consequence of supporting communication (through conditions) in the midst of transactional code, as preserving isolation would prevent communication. Since we cannot avoid breaching isolation where processes communicate, we must instead take measures to make this consequent evident to the programmer, even when the `Wait` appears at several removes of procedure calls from the transaction that it may punctuate. The TIC model offers type system support for keeping track of `Wait`ing operations (as well as other kinds of operations that violate transactional semantics, as discussed in the next section). The programmer can then decide whether the code can be reorganized so that the operation is performed outside the atomic section, or if the operation is safe inside the atomic section with an `establish` clause used to re-establish the invariants of the transaction after its suspension.

More specifically, we define *waiting* methods to be those that contain a call either to `Wait` or to another waiting method (that is, all methods that may call `Wait` directly or indirectly). A call to a waiting method is not legal unless it occurs inside an `expose ... establish` expression. The syntax of the `expose ... establish` expression is

`expose (`*Expression*`)` *[*`establish` *Statement]*

where *Expression* is a single call to a waiting method and the return value of the method becomes the value of the entire `expose ... establish` expression. The `establish` clause is optional—omitting it is equivalent to an empty statement following the `establish` keyword. If the call to the waiting method results in the transaction being suspended by a `Wait`, then, when the transaction resumes and the method call returns, *Statement* is executed. Subsequently, if the bottom transaction later aborts and re-tries, the `establish` clause is also always executed (on return of control flow to this method). Nevertheless, if the transaction is never actually suspended, either because its control flow does not reach the `Wait` statement, or because the condition of the `Wait` is already true when first checked, then *Statement* is not executed. For illustration, consider a waiting method `foo` and a transaction calling it:

```
void foo() { Wait(x > 0); }
void bar() {
  atomic {
    y = 0;
    expose(foo()) establish { y = 1; } ;
  }
}
```

[2] We use the same mechanism as Harris and Fraser [28] for checking when a condition has possibly changed value—namely, we observe what updates get committed to locations that the waiting transaction accessed. A desirable property for the condition expression is that it be side-effect free, but we do not currently try to enforce this automatically.

The value of `y` at the end of the atomic block will depend on whether the transaction is ever punctuated. If the transaction commits as a whole by reading an `x` greater than zero, which prevents the `Wait` statement from suspending the transaction, then the `establish` block is not executed, and the value of `y` is zero. Otherwise, the value of `y` is 1.

Note that, according to our requirement, method `bar` itself cannot be called outside an `expose ... establish` statement. The reason is that the use of `bar` in an atomic statement can cause the suspension of the transaction with its top part (up to the call to `bar`) committed and the rest of it executing independently and possibly being retried. Code preceding the call to `bar` has to establish global invariants, in anticipation of a possible suspension. The `establish` clause (and code following it) is then used to ensure consistent execution by re-establishing local invariants. (Section 3 offers usage examples in real scenarios and argues why this is a good approach for modularity purposes.)

The programmer can suppress the requirement for an `expose ... establish` around a method's calls, if the method is certain to never be used inside an atomic section. This is done with the annotation `toplevel`. The type system disallows calls to a `toplevel` method in transactions. For instance, method `bar` in our example above contains a transaction, but it could itself be prevented from ever being called inside a transaction if its type signature is changed to:

```
toplevel void bar() { ... /* as before */ }
```

In this case, calls to `bar` no longer need to be under an `expose ... establish` clause. Naturally, methods that call `toplevel` methods are also not usable in transactions—a property verified by our type system.

It is important to realize that the requirement for an `expose` clause is the type system reminder to the programmer that he/she needs to fulfill two obligations: ensure that *atomicity* can be relaxed, so that other threads can observe current results consistently, and ensure that *isolation* can be relaxed so that the current transaction can observe the effects of other threads (after the `Wait` returns) without violating its consistency properties. The first of these properties is typically handled by the code *preceding* the call to `Wait` and not by the code following `Wait` (including the `establish` clause in callers), whose purpose is to handle the second property.

2.2 Handling Suspending Operations

Waiting methods are not the only ones that require special treatment in transactions. Any method that violates either atomicity or isolation needs to be handled specially. Typically this is due to irreversible actions affecting external resources. We call all non-waiting methods that require special handling *suspending* methods. The treatment of suspending methods in TIC is almost identical to the treatment of waiting methods, described previously. This uniformity is an interesting feature of TIC. Nevertheless, there are some subtleties. First, we need to conceptually identify which methods are the *root* suspending methods so that method type signatures at the interface with system code are correctly labeled. Second, as is also common in other transactional settings, a programmer is allowed to specify "undo" actions to allow suspending methods to be safely used in transactions. Finally, the behavior of retrying a transaction is slightly different in the case of suspending methods, compared to waiting methods. We discuss these points next.

An (external) operation is safe to use inside a transaction without surrounding code being aware, if both:

1. its effects are not exposed to other threads in a way that may violate application correctness

   and

2. its effects can be reversed.

Some approaches (e.g., [16]) propose weakening the second condition to "the operation is *idempotent*: Re-executing it has the same effects and result as executing it once". However, this is not correct in a general programming model. Even if an operation is idempotent, retrying a transaction after a change to shared data can result in the operation now being outside the control-flow of the transaction, or being called with different arguments. Establishing the idempotency of an entire transaction body after a change to shared data is generally infeasible. Thus, neither reusing previous results and effects of a suspending operation nor re-running it are generally safe in the course of retrying a transaction.

In the TIC model, methods that have effects that violate the requirements of the transactional memory system can be labeled using the annotation `suspending` in the method declaration. This annotation is best applied to system-level operations (e.g., native methods in Java). Any method that calls a suspending method is also implicitly suspending—we use the term "root suspending operations" to distinguish the base methods that have the `suspending` annotation. For instance, a JDK implementation will likely declare method `write` in `java.io.RandomAccessFile` as:

```
public suspending native void write(int b)
  throws IOException;
```

External operations that can be reversed represent the easy case of handling suspending methods. Following the example of *open-nesting* transactional models [46, 45, 52], we allow the user to specify for each "forward" operation an "undo" operation and an "on-commit" operation (collectively called "compensating operations"). These are designated by annotations `undo` and `oncommit`, respectively, on the forward operation. Both operations are methods on the same object as the forward operation. Additionally, compensating operations accept arguments of the same type as the forward operation, plus extra arguments of the same type as the return type of the forward operation (if non-void) and any exception types the forward operation may throw. For instance, a common pattern is that of an operation `release` as

the undo operation for method `allocate` (but not vice versa). This would be specified as:

```
undo(release)
Entity allocate(String name, int length);

void release(Entity e, String n, int l);
```

Every method that has an `undo` annotation causes the outermost transaction it contains to have open-nesting semantics, relative to the method's enclosing transactional context. This means that the transaction commits at its end, independently of any parent transaction, thus making its results immediately visible to other threads. If the parent transaction (i.e., the transaction surrounding the method) needs to roll back and retry, the system calls the method's undo operation to reverse prior committed effects of the nested transaction. Root suspending operations in a method with an `undo` annotation are also handled similarly. A root suspending operation is treated as if it were an open-nested transaction. It is considered to commit if control flow reaches it, which causes (upon method completion) the undo action of the surrounding method to get registered for a possible compensating action in the future.

The TIC model for compensating actions is slightly unconventional, in that the on-commit operation in TIC is independent of nesting semantics: a method can have an `oncommit` annotation either with or without having an `undo` annotation and open-nested transactions in it. The on-commit operation is registered upon return of the annotated method. If/when the innermost open-nested transaction, or (if all transactions are closed-nested) the outermost transaction surrounding the method validates its reads and is ready to commit, the system calls the on-commit operation of the method.

We later give a more precise description of the TIC nesting model, as well as a comparison with traditional open-nesting (Section 5). Until then, we focus more on TIC's transaction punctuating features and compare the model to standard closed-nested transactions.

The interesting case of external operations concerns those that are not called under a method with an undo operation. Such suspending operations are treated much like waiting ones. A transaction can call a suspending operation only under an `expose ... establish` clause. The transaction will again be punctuated: It will commit immediately before the root suspending operation call, just as it commits before blocking on a `Wait`. The `establish` statement is expected to re-establish the transaction's invariants after the suspending call and the resulting loss of isolation. There is a subtle difference, however. In case of a transaction retry, the root suspending method call is not repeated. Instead, execution resumes from the return point of the root suspending operation, making the corresponding `establish` clause the first statement executed. For instance, consider a transaction:

```
atomic {
  if (!balanceUpdated()) {
    bal = compute();
    expose(print("Balance:" + bal)) establish {
      if (balanceUpdated()) // someone raced us
        return;
    } ;
    updateBalance(bal);
  }
}
```

The transaction computes a result based on shared variables and exposes it with an external, irreversible operation (`print`). At this point, the transaction commits and a different thread may have raced to fill in the needed result. In this case, the transaction conservatively chooses to avoid the final update. If the transaction proceeds to `updateBalance` and this encounters contention that causes a retry, then the transaction will restart from the point right after the execution of the external operation—that is, from the statement under `establish`. If the user wants to repeat the suspending operation in case of a transaction retry (perhaps with different arguments) an explicit loop should be placed around the code containing `expose ... establish`. For instance, consider the following example:

```
atomic {
  balance = compute();
  print("Your balance is " + balance);
  bet = input("How much will you wager?");
  if (bet <= balance)
    register(bet);
}
```

If `print` and `input` are irreversible operations, then this is not valid TIC code—`expose ... establish` clauses need to be used. Nevertheless, this example represents a "hopeless" case. There is no way to re-establish the transaction's invariants with an `expose ... establish` clause if the external world (human user) observes a balance that is no longer correct. The only reasonable recovery in this case is to retry the whole transaction. The user's input is based on prior output, and, hence, needs to be obtained again. We can do this with an explicit loop:

```
atomic {
  while (true) {
    balance = compute();
    expose(print ("Your balance is " + balance)) ;
    bet = expose(input("How much will you wager?"))
     establish
      { if (balance == compute()) break; };
  }
  if (bet <= balance)
    register(bet);
}
```

This creates the obligation to handle transaction suspension in all possible transactions surrounding the current code. For this example, since no recovery of any kind

is meaningful and we need to repeat the entire transaction, we have an alternative that places a lower burden on clients: We can use the `undo` annotation to prevent transaction punctuation and remove the obligation of using `expose ... establish` clauses in all surrounding transactions. An empty undo action causes the suspending operations to behave as if they are perfectly reversible. In this way we also avoid the explicit loop, in favor of the natural looping behavior of transaction retry. For instance, we wrap the `print` and `input` methods:

```
undo(doNothingString) void myprint(String s) {
  print(s);
}
undo(doNothingintString) int myinput(String s) {
  return input(s);
}
```

Using the wrapped methods in place of the originals achieves the desired effect without a need for a loop or `expose ... establish`.

3. Applications and Experience

We next discuss examples of the applicability and benefits of TIC relative to existing transactional programming models. We use C, C++, and Java realizations of TIC in our examples—see Section 4 for an implementation discussion.

3.1 Shortcomings of Traditional Models and TIC

TIC arose from our experience in implementing multithreaded applications and trying to express them or restructure them to work with transactional memory mechanisms. Despite assertions regarding the composability of transactions in comparison to locking [28, 29], we have found transactions to not compose well because of the presence of nontransactional operations. We saw an example in the Introduction, involving a barrier pattern. In practice, many common patterns have to do with system-level suspending operations and not with the need to cooperate with other threads.[3]

A general pattern that we observed several times in practice is the following: An atomic section would normally have a suspending operation inside it. With some code restructuring and minor bookkeeping, the operation may be movable outside the transaction:

```
void methodWithTransaction() {
  atomic {
    ... <set bookkeeping data> ...
  }
  <use bookkeeping data to perform external op>
}
```

Nevertheless, this has rarely been sufficient. Other code using a transaction (possibly to protect some entirely distinct data than the above transaction) often needs to call `methodWithTransaction` (and may even do so inside a loop):

```
atomic {
  ... methodWithTransaction(); ...
}
```

Now the external operation suddenly finds itself executed as part of a transaction, and possibly (erroneously) repeated when the transaction retries. The operation needs to be moved again, this time outside the atomic section in the caller method. This may require significant code restructuring: Functional abstraction may need to be violated, transactions may need to be split, etc. The unfortunate conclusion is that transactions do not compose well. A transaction can be oblivious to the synchronization strategies of methods it calls, but it cannot be oblivious to suspending operations in these methods. In short, *in transactional code, performing an irreversible operation is a global property*, just as, in lock-based code, holding locks is a global property. Suspending operations have the potential to render incorrect all transactions under whose dynamic scope they execute, and not just the immediately surrounding transaction.

The problem of transactional code not composing in the presence of waiting or suspending operations is unavoidable and TIC offers no magical solution. What the model does is expose to the programmer the points where extra "glue" needs to be applied and enable him/her to handle the composition of transactional code, by restoring only *local* invariants every time. The programmer always has the option to revert to standard techniques for handling suspending/waiting in transactional programs, such as moving code outside transactions. In many cases, however, using TIC results in significantly simpler and more modular code, in addition to helping avoid bugs. We give some specific examples from actual code.

3.2 Recovering from Suspending Operations

We reengineered the version of the Kingsley memory allocator supplied with the Heap Layers suite [6] to work with transactions, as opposed to fine grained locking. The pattern we present, however, is typical of multiple memory allocators. It is a good example for the TIC model, because it exposes complexity without being overwhelming.

The Kingsley allocator is one of the fastest general-purpose memory allocators [6]. The allocator divides free blocks into power-of-two size classes. A fragment of the code in the main allocation routine (using transactions but *not* using the TIC model) is shown in Figure 1. The code uses an atomic section to consistently access a shared data structure. In the middle of multiple accesses to shared data, the code calls operation `morecore` to get more memory from the operating system if the appropriate free list is empty. Function `morecore`, however, calls `sbrk`, which is an irreversible system operation. (Even if the operating system allows lowering the `brk` pointer, another thread could have moved it

[3] It is unrealistic to expect that eventually all system-level operations will acquire transactional semantics. Even though transactional memory allocators or transactional file systems already exist, transactional network I/O, or user I/O is nearly infeasible. In general, much of the external world is deeply not transactional, as effects cannot be undone.

```
void *kmalloc(int sz) {
  .../* determine which free list to use, based
       on size, see if free blocks are available */
  atomic {
    if ((op = nextf[bucket]) == NULL) {
      morecore(bucket);
      if ((op = nextf[bucket]) == NULL) {
        return (NULL);
      }
    }
    /* remove from linked list */
    nextf[bucket] = op->ov_next;
    op->ov_magic = MAGIC;
    op->ov_index = bucket;
    ...
  }
  ...
}
```

Figure 1. The main structure of the Kingsley allocator's `malloc` routine.

by that time.) Thus, if the transaction naively retries, `sbrk` will be called twice, leaking OS resources. The code for `morecore` is shown (only very slightly simplified) in Figure 2. The purpose of showing this code is to demonstrate where suspending call `sbrk` is in the program logic, as well as to show accesses to the shared data structure (rooted at array `nextf`) which depend on the result of the `sbrk`, yet need to be under the surrounding atomic.

Consider the code reorganization required to remove `sbrk` from inside transactional code. This would require breaking `morecore` in two parts, top and bottom, and moving `sbrk` into the body of `kmalloc`. Since the top and bottom parts of `morecore` need to communicate data, their interfaces need to include extra arguments (e.g., `nblks`, `op`). The modularity of the original code is lost: `kmalloc` now needs to be directly aware of the functionality that used to be inside `morecore`.

For this example, one can also envision a solution with a pair of unstructured `beginAtomic`/`endAtomic` primitives, instead of a block structured `atomic` section. Yet this would be a very error prone programming model. Furthermore, note that (unlike with locks) a programmer cannot use a block structured `atomic` to build unstructured primitives.

The TIC approach solves the problem cleanly. The transaction is committed before calling `sbrk` and the call to `sbrk` is never repeated, even if the rest of the transaction retries. The potential consistency problem with suspending the transaction at the point of calling `sbrk` is that a different thread can race and may happen to replenish the same bucket. The original code overwrites the link to such updated entries in the bucket, as it assumes that `nextf[bucket]` is `NULL`. An easy rewrite of the part of `morecore` following the call to `sbrk` is enough to ensure that the result of `sbrk` is consistently added to the data structure, even if another thread has changed the bucket. The new code does not assume that the bucket is still empty after the potentially suspending `sbrk` call. Indeed, with the rewritten code, even an

```
static void morecore(int bucket) {
  register union overhead *op;
  register long sz;   /* size of desired block */
  long amt;           /* amount to allocate */
  int nblks;          /* how many blocks we get */

  sz = 1 << (bucket + 3);
  if (sz <= 0)
    return;
  if (sz < pagesz) {
    amt = pagesz;
    nblks = amt / sz;
  } else {
    amt = sz + pagesz;
    nblks = 1;
  }
  op = (union overhead *)sbrk(amt);
  /* no more room! */
  if ((long)op == -1)
    return;
  /*
   * Add new memory allocated to that on
   * free list for this hash bucket.
   */
  nextf[bucket] = op;
  while (--nblks > 0) {
    op->ov_next = (union overhead *)((caddr_t)op + sz);
    op = (union overhead *)((caddr_t)op + sz);
  }
}
```

Figure 2. The routine to get more system memory inside the Kingsley allocator. Moving the `sbrk` call outside the enclosing transaction (in the calling routine, `kmalloc`) requires major code reorganization.

empty `establish` clause is sufficient. The result is correct regardless of whether the data structure was concurrently modified by another thread:

```
static void morecore(int bucket) {
  union overhead *fst = NULL;
  ... // as before
  op = expose ((union overhead *)sbrk(amt));
  fst = op;
  while (--nblks > 0) {
    op->ov_next =
      (union overhead *)((caddr_t)op + sz);
    op = (union overhead *)((caddr_t)op + sz);
  }
  op->ov_next = nextf[bucket];
  nextf[bucket] = fst;
}
```

Similarly, the `kmalloc` routine is easily fixed to be unaffected by the transaction suspension caused by `sbrk` inside `morecore`. An empty `establish` clause would be sufficient, but note that the suspending operation `morecore` will cause the punctuation of any user-level transactions that happen to use `kmalloc`. Even though the depth of transaction nesting is expected to be low in typical applications [16], we still would not want to impose on users the burden of handling punctuation at every level of their transactions every time

they call kmalloc. Instead, it is easy to supply an undo routine for kmalloc (a wrapper around kfree) so that its atomic section commits independently of any surrounding transactions, and has its results reversed if the surrounding transaction retries. The changes are shown below.

```
undo(myKfree) void *kmalloc(int sz) {
  ... // as before
      expose (morecore(bucket));
  ...
}
```

3.3 Cooperating Threads

The TIC ability to Wait inside a transaction enables thread cooperation without disrupting transactional coding patterns. Barrier patterns, such as the one shown in the Introduction are a simple case of applicability for TIC. TIC makes expressing barriers easy by allowing a Wait statement to circumvent atomicity by exposing results, and to disable isolation so that effects of other threads can be observed. This means that a barrier call requires special handling in all enclosing transactions, with an expose ... establish clause.

It should be noted that the semantics of Wait in TIC is consistent with prior experience in re-engineering multithreaded applications. Chung et al. [16, 12] rewrote 35 lock-based applications to use transactions. They note that the most reasonable simulation of condition variable waits in the transactional world is to "map wait to an END marker (end previous transaction) and a BEGIN marker (start new transaction) pair" [16]. This is directly analogous to our treatment of Wait. In a different study, the same authors write: "we have never seen a benchmark or system that exhibits a problem treating wait as commit" [12]. Nevertheless, they also note that "if we treat wait as a commit, it is easy to come up with contrived programs that will not match the previous semantics". The TIC ability to identify these cases and recover with an expose ... establish clause is unique, to our knowledge.

For an actual example where recovery is easy but necessary, Figure 3 shows two methods, rewritten in a transactional form, from the code of the Zimbra Collaboration Suite. (The routines are slightly simplified—intermediary methods were removed, as was the code for throwing and handling some database exceptions.) There are two atomic sections, one in each method. Method setConnectionProvider has a transaction that may be suspended at various points. There are three expose ... establish clauses shown in the code. Two of these are for suspending operations, such as destroy or start. One of the calls is to getConnection, which is a waiting method. If a connection is not available in a shared pool, the thread will wait until another thread returns a connection to the pool. In all cases, the transaction only needs to worry about its own invariants in case of suspension. Recovery is quite easy: The routine can just return if any other thread has raced to overwrite the connectionProvider shared variable.

```
public static void
setConnectionProvider(ConnectionProvider provider) {
  atomic {
    if (connectionProvider != null) {
      ConnectionProvider old = connectionProvider;
      expose(connectionProvider.destroy()) establish {
        if (connectionProvider != old) return;
      } ;
    }
    connectionProvider = provider;
    expose(connectionProvider.start()) establish {
      if (connectionProvider != provider) return;
    } ;
    // Now, get a connection to determine meta data.
    Connection con = null;
    con = expose(connectionProvider.getConnection())
          establish {
            if (connectionProvider != provider)
              return;
          } ;
    setMetaData(con);
    ... // multiple other uses of con
  }
}

public Connection getConnection() {
  ConnectionWrapper con = null;

  while(true) {
     atomic {
       // if shutting down, don't create connections
       if (shutdownStarted) return null;
       Wait(connectionAvailable);
       con = getCon();
       if(con != null) {
          con.checkedout = true;
          con.lockTime = System.currentTimeMillis();
          return con;
       } // else someone got it before us, try again
     }
  }
}
```

Figure 3. getConnection is a waiting method. Calling it requires an expose ... establish clause, which is quite easy to write. connectionProvider is a shared variable.

3.4 Temporary Violations of Atomicity

Long-running operations in the middle of a transaction increase the probability of the transaction aborting due to contention. TIC allows a long-running computation to move to an independent thread and the transactions to coordinate using Wait. In the easiest case, the long-running operation only needs to signal its completion to the main transaction. TIC then also allows labeling the operation as suspending, in which case the transaction will commit just before performing it and a new transaction will start after it.

The latter also corresponds to a common lock-based programming pattern: releasing a lock only to reacquire it after an operation. We counted at least 8 instances of this pattern

for different tasks in AOLserver, all for long-running operations. (AOLserver is an open-source web server, originally by America Online. See http://www.aolserver.com .) For instance, AOLserver uses code of the following form in its interface to the Tcl interpreter.

```
lock(l);
  ...
  do { ...
    unlock(l);
    ... // call Tcl interpreter with script arg
    lock(l);
  } while(cond);
  ...
unlock(l);
```

In the TIC model, this corresponds to committing a transaction and starting a new one after the Tcl interpreter invocation, without any need for establishing consistency. That is, the Tcl invocation operation is labeled `suspending`, and called under an `expose` call with no `establish` clause.

3.5 Type System Warnings

In the TIC model, the type system does not guarantee transaction safety, but serves as a reminder to ensure that the programmer has not overlooked waiting or suspending operations. This is often sufficient for detecting serious functionality or performance errors. We encountered a representative example in our rewrite of the AlphaMail server [38], which we re-engineered to use transactions.

AlphaMail's functionality of interest is an IMAP web cache: a middleware system that facilitates communication between the web server software and an IMAP server. AlphaMail uses a cache data structure that holds data about recently accessed IMAP data folders, including a persistent connection to the network folder. This is a C++ `map<string,shared_ptr<IMAPFolder> >` data structure: an associative map from strings to reference-counted pointers to `IMAPFolder` objects. A cleaning thread runs periodically over the data structure to remove entries corresponding to folders not accessed recently. This traversal is a standard data structure removal:

```
atomic {
  ...
  for(i=cache_map.begin(); i!=cache_map.end(); i++)
    if(getIdleTime(i->second) > timeout)
      cache_map.delete(i->first);
}
```

The seemingly innocuous code results in undesirable interactions with synchronization code. The `delete` call removes the item from the map, and, if this was the last reference to the item in the program, then the `shared_ptr` destructor deletes the `IMAPFolder` object itself. Irreversible operations (network flush and close) can then occur through a complex chain: Destroying the `IMAPFolder` destroys an `iostream` object, which destroys a `streambuffer` object, which shuts down a `TCPStream`, which contains the offending operations. This is a standard case where our type system warns of suspending operations, similarly to other examples we discussed earlier. Although the operations are deeply nested, an `expose ... establish` clause is needed at every level to allow them to be used in a transaction. Interestingly, however, the above sequence was also a serious bug in an earlier lock-based version of AlphaMail. The code is holding a lock while the connection is being closed, which prevents concurrency during a long-running operation. In the worst case, the connection to the IMAP folder is experiencing network problems, making hundreds of other users hang until the network connection times out.

3.6 On-Commit Operations

Often we can postpone suspending operations until the surrounding transaction can commit. This can be done with an on-commit operation. We consider an example from AlphaMail [38]. AlphaMail is written in C++ and is linked against a non-transactional memory allocator. (Although memory allocation can be built so that it integrates seamlessly with transactions [37], there may be performance reasons to prefer a multithreaded allocator utilizing fine-grained locking, e.g., [5]. Furthermore, there are always low-level libraries that use their own allocation routines (e.g., OpenSSL's `SSL_CTX_new/delete`). It is not reasonable to expect a close integration for all such libraries.)

The interface of an application with a memory allocator is narrow, consisting only of operators `new` and `delete` (or `malloc` and `free`). Therefore, it is reasonable to expect that combining a transactionally implemented application with a fine-grained locking allocator should be easy. Yet, although it is relatively easy to write an undo routine for `new` (using `delete`) it is not similarly easy to write a satisfactory undo routine for `delete`. (Reversing a `delete` is not possible even if the allocator internals are known: once the object has been reclaimed, some other thread could have raced and reused the space.) `delete` is also quite hard to handle since it is a lightweight enough operation that it is likely to be used in many transactions (unlike I/O operations that have high latency and will likely need to be moved outside of critical sections). One of the uses of `delete` in AlphaMail is in a reference counted shared pointer class. As commonly expected, the assignment operator of a shared pointer decrements the reference count of the object that the pointer used to point to and calls `delete` on it if the count is zero. It is relatively easy to move the call to `delete` outside the atomic section, by introducing variables to remember whether the object should be deallocated and doing so later. The relevant code fragment is shown in Figure 4. (The code is slightly simplified—notably it does not include the common intrusive reference counting optimization [3, ch.7].)

Nevertheless, this hardly fixes the problem. Shared pointers are used in several places in the application inside transactions. Consider a seemingly innocuous statement such as:

```
template<class T> class shared_ptr {
...
public:
  shared_ptr<T> &operator=(const shared_ptr<T> &b) {
    bool delete_old_object = false;
    int *old_count;
    T  *old_obj;

    atomic {
      old_count = shared_count;
      old_obj = obj;

      obj = b.obj;
      shared_count = b.shared_count;
      (*shared_count)++;
      (*old_count)--;
      if(*old_count == 0)
        delete_old_object = true;
    }

    if(delete_old_object) {
      delete old_count;
      delete old_obj;
    }
  }
private:
  T *obj;
  int *shared_count;
};
```

Figure 4. A shared pointer class that makes sure the deallocation is performed outside the `atomic` section, since `delete` is not transaction-safe.

```
atomic {
  ... p = q; ...
}
```

For `p` of type `shared_ptr<int>`, the assignment calls `shared_ptr<int>::operator=` which contains the call to `delete`. If the transaction retries, `delete` will be called twice. Fixing this problem by moving code requires destroying the encapsulation of the shared pointer class and moving some of its functionality outside all transactional code. Thus, this is the standard problem we discussed in Section 3.1: Calling `delete` is a property visible to all clients of the operator. In this case, we can postpone the results of the `delete` call until the end of the transaction. This is easy to do by just creating an on-commit operation for `shared_ptr<int>::operator=`. The transaction code is then free of suspending operations, but makes a record of deleted objects available to the on-commit operation. In the current formulation, the easiest way for the two routines to share data is through arguments and return values. (In the future, one can imagine adding richer support for sharing data with compensating actions—this is an aspect orthogonal to TIC's main features.) Thus, we can make an intermediate routine `release_item`, which `operator=` calls with the objects to delete as arguments. The on-commit operation, `free_item` is attached to this routine. The system stores the arguments and makes them available to the on-commit operation, which performs the deletion:

```
template<class S>
oncommit(free_item<S>) void release_item(S *c) {
  // no-op. Transaction system records params
}

template<class S>
void free_item(S *param1) {
  delete(param1);
}
```

The above on-commit operation does not need any concurrency control, as it accesses no shared data. An important point, however, is that the on-commit operation can contain transactions, which execute open-nested in the current context. Thus, it can roll back and retry, which renders suspending operations problematic. Therefore, on-commit operations themselves can have the same need as regular code to include `expose ... establish` clauses, in order to recover from transaction punctuation. The exact nesting model of TIC, as well as the interactions between nested transactions, compensating actions, and transaction punctuation are discussed in detail in Section 5.

4. Type System and Implementation Discussion

We next discuss more precisely some aspects of the TIC design, as well as our prototype implementations.

4.1 Language Summary and Type System

To summarize the previous sections, the elements of the TIC programming model are:

- The `atomic` keyword to designate transactions.
- The `Wait` keyword to explicitly suspend transactions until a condition is satisfied.
- The `toplevel` method annotation, which makes a method unusable inside a transaction.
- The `expose ... establish` syntax for calling waiting or suspending methods.
- The `suspending` method annotation designating a root suspending method.
- The `undo` and `oncommit` method annotations that specify compensating actions for the method and (in the case of `undo`) cause a method to commit its transactions independently of external nesting.

Note that many of these do not have run-time semantics, but only static semantics. That is, they exist purely for typing purposes. They enable the type system to keep track of code that requires special handling in transactions, in order to remind the programmer appropriately. Overall, our type system is straightforward, as it is *propositional*: It only adds three true/false flags to program methods. The first flag denotes waiting operations, the second denotes suspending op-

erations, while the third denotes operations guaranteed to be unusable inside transactions. The flags propagate as follows:

- A `Wait` statement sets flag *waiting* for the method that contains it, unless the method also has a *toplevel* flag.
- A `suspending` method annotation sets flag *suspending* for the method. A method with a `suspending` annotation cannot also have an `undo` annotation or a `toplevel` annotation.
- A `toplevel` method annotation sets flag *toplevel* for the method.
- On a method call, if the callee has a *suspending* flag, the same flag is set on the caller, unless the caller has the *toplevel* flag or an `undo` annotation.
- On a method call, if the callee has a *waiting* flag, the same flag is set on the caller, unless the caller has the *toplevel* flag.
- On a method call, if the callee has the *toplevel* flag, the same flag is set on the caller, unless the caller has an `undo` annotation.

(Note that we phrased the rules as inferences with negation—e.g., "has ... unless the caller has...". In general this might lead to ambiguity. The reader can verify that negation is stratified, however, hence we can get a consistent flag assignment by letting the rules run to fixpoint.)

As discussed earlier, the consequences of these flags are straightforward. A method flagged *toplevel* cannot be used in a transaction. A method flagged *waiting* or *suspending* needs to be under an `expose ... establish` clause when used in a transaction.

The above rules assume a known caller-callee graph. In an object-oriented language our type system needs to be conservative, in order to support dynamic dispatch and an unknown set of subclasses: Overriding methods are only allowed to be more broadly applicable than the methods they override. Then we need to introduce an explicit method annotation `waiting` and some additional rules:

- A `waiting` method annotation sets flag *waiting* for the method, unless the method also has a *toplevel* flag.
- A method with the *suspending*, *waiting*, or *toplevel* flag cannot override one without the same flags.
- A method with an `undo` annotation cannot be overridden by one without it.

Note that the last two are *not* propagation rules. The rules do not cause the overridden/overriding method's flag to be set. Instead they dictate that if the flag is not set under the propagation rules, the overriding is illegal.

These rules are safe, but restrictive. E.g., they force every client of an interface method to make a call under an `expose ... establish` if even one implementation of the method is suspending.

4.2 TIC Prototypes

We described our language extensions in an idealized setting (as new keywords with full language support). As is standard practice, however, we approximate these features with simpler extensions that offer an easier transition path from existing languages. Our original prototype was a back-end C library, based on Harris and Fraser's libstm back-end library [28]—a fully optimistic concurrency implementation, with read and write logging. Our changes to the library implement the main back-end features of TIC—namely, the full continuation support, explained next, and the TIC nesting model (including open nesting support) described in Section 5. The libstm implementation gave us a way to evaluate a prototype quickly. Nevertheless, it was not ideal for practical use, mainly because of its lazy validation policy: Since the library does not detect inconsistent data reads until explicit validation time, client programs needed to be hardened in multiple ways (typically using signal handlers, but also by ensuring no infinite loops occur) to avoid anomalies from reading invalid data. For a more practical library, our current working prototype is based on TL2 [17]: an optimistic concurrency implementation with eager read validation.

To experiment with our library in actual applications, we created a C++ wrapper library, containing macros (for `atomic`, `expose`, `establish`), and a set of classes to support threads, semi-automatic nesting, and compensating actions. The user still needs to carefully ensure that the desired memory actions are performed through the transaction system, but the C++ wrapper offers rudimentary syntactic sugar and safety checks. (As for other C libraries, the biggest challenge for seamless use in C++ is that the user needs to explicitly compensate for the implicit semantics of C++ operations—e.g., destructors—that are not preserved by our runtime manipulations.) We have not yet created a mature implementation for Java (currently the user needs to directly call the C back-end) but it is straightforward to employ the standard approach of Java 5 method-level annotations [22, section 9.7] for syntax extension and bytecode transformation for adding semantics, without needing to change the source compiler. Our propositional type system easily translates to existing constructs in the Java type system (e.g., require an `expose ... establish` clause through Java's static check for exception catching). The recent literature is rich with mechanisms applicable in our context for translating transactional extensions down to regular Java [1, 30, 31, 34, 40, 46]. Therefore, this aspect of the implementation is well-understood, and we concentrate next on elements unique to TIC that are currently captured by our back-end library.

4.3 Implementation Discussion

The TIC model has slightly higher implementation requirements than a standard transactional programming model. This is due to the need for full continuations when a transaction needs to retry after it is suspended. Consider our earlier

example of kmalloc with an atomic section that contains a call to morecore, which contains a suspending operation.

```
undo(myKfree) void *kmalloc(int sz) {
  ... atomic {... expose(morecore(bucket));...} ...
}

void morecore(int bucket) {
  register union overhead *op;
  register long sz;   /* size of desired block */
  long amt;           /* amount to allocate */
  int nblks;          /* how many blocks we get */
  union overhead *fst = NULL;
  ...
  op = expose ((union overhead *)sbrk(amt));
  ...
}
```

Just before the call to sbrk, the transaction consisting of all program actions from the beginning of the atomic block up until the sbrk statement commits. A new transaction is started, immediately after the call. If that new transaction encounters contention and needs to retry, its starting point is immediately after the sbrk call *even though function* morecore *has returned*. This means that the transaction system needs to have captured the full continuation corresponding to the state right after the sbrk call. This should include the state of stack variables, such as sz, bucket, op, etc. (We assume a conventional stack/heap state split, although clearly a runtime system may choose any alternative implementation.)

The requirement for full continuations is only a modest increase from the bookkeeping required in standard (non-punctuating) transactional models. The heap portion of a program's state, as well as the state of the topmost stack frame, need to be tracked by conventional transaction mechanisms anyway. For instance, consider the above routine kmalloc with a standard block-structured atomic section. On a transaction retry, the implementation still needs to be able to restore the stack state of kmalloc as of the beginning of the atomic block. However, a conventional implementation does not need to track the stack state of suspending methods called by kmalloc.

This modest cost of creating and using full continuations is actually incurred rarely. Full continuations are needed only in atomic sections that have an expose clause or a Wait operation, and only if transaction suspension actually occurs—that is, if the condition of the Wait operation is false, or an innermost suspending operation is indeed executed.

Our implementation relies on the existing TL2 mechanisms for handling conflicts when concurrent atomic sections access global or heap data and to discover when a transaction needs to be aborted. Changes were necessary only to handle saving and restoring local state (i.e., registers and stack frames) when a transaction commences and aborts, respectively.

Our implementation specifics (modulo open-nesting, described in the next section) are fairly straightforward. Read and write operations to global and heap locations are implemented with calls to the underlying library's word-level read and write primitives. Entry to and exit from an outermost atomic section results in the beginning and attempt to commit, respectively, of a TL2 transaction. Innermost (i.e., root) suspending operations attempt to commit the transaction. If the transaction successfully commits, the suspending operation is executed and a new transaction begins immediately after it. In case of a Wait, we first test the Wait condition and attempt to commit the transaction if it is false. The new transaction begins with an evaluation of the Wait condition. We had to enhance the base TL2 library to include a blocking primitive, which was modeled after STMWait in libstm.

When a conflict is discovered, the commit operation cannot complete, and the current transaction must be rolled back to its initial state. The base library takes care of restoring global variables and heap data; to restore the state of any local variables and the program counter we use our own implementation of continuations. This is architecture specific, but straightforward, as we are not concerned with heap space. The implementation saves continuations when transactions are punctuated (via the expose and establish macros). The management of continuations is handled behind the scenes by calling semi-portable routines, such as memcpy (to copy the stack), setjmp, and longjmp. As discussed, continuations are created only when a Wait statement is reached and its condition is false or a call is made to a root suspending operation.

Our measurements show that the cost of saving/restoring full continuations is modest, and becomes negligible if one considers how rarely it is incurred (only on actual suspension). For our kmalloc example, we measured a cost of 230 cycles for saving the registers and stack frames for a full continuation (all numbers are for single-threaded execution on a 2.16GHz Intel Core 2 Duo and report the median of seven runs, each averaged over 30,000+ iterations). However, the overhead of setjmp is already incurred by TL2 on all transactions as part of setup for possible transaction retries. This means that the cost of a continuation is reduced to the cost of a memcpy for the current thread's active stack frames. For example, a transaction containing no function calls and two writes takes 900 cycles. The same code with the transaction punctuated between the two writes takes 1650 cycles, which is almost identical to two separate single-write transactions at 1640 cycles.

5. Nesting in TIC

We next discuss topics concerning TIC and transaction nesting. We first examine the relationship of TIC to the idea of open-nesting in general. Then, we describe in more detail the TIC nesting model. We also discuss the safe use of open-

nesting transactional systems in general, and interesting interactions of transaction punctuation with open-nesting.

5.1 Relation to Open-Nesting

The main feature of the TIC model is transaction punctuation through waiting or suspending operations. Nevertheless, TIC also integrates open-nesting features, such as open transactions and compensating (undo and on-commit) actions. It is interesting, therefore, to ask how TIC compares with open-nested programming models. The answer is twofold:

- TIC has distinctly different goals than open nesting: Rather than addressing scalability and performance concerns, TIC aims to support thread communication and irreversible operations, while maintaining the high-level properties of transactions. Nevertheless, TIC can sometimes be used to address performance concerns, as an alternative to open-nesting. Consider the example of a fairly independent but long-running operation that needs to be executed in the middle of a transaction. (We saw such examples in Section 3.4.) Open-nesting allows long-running operations to commit independently, in an open transaction. TIC allows them to move to a different thread and have the two threads coordinate using `Wait` statements. Alternatively, TIC allows the programmer to label the long-running operation with a `suspending` annotation, which punctuates the main transaction and returns to it on completion.

  In principle, open-nesting could also be used for some of the main TIC tasks of thread communication and irreversible operations. Yet open-nesting offers a *lower-level* programming model, delegating to the user the responsibility for establishing higher level properties using (regular or abstract) locks [13, 45].[4] Without user intervention, the default semantic guarantees of open-nesting are much weaker than those of TIC. Agrawal et al. [2] offer examples where open-nesting violates fundamental properties of transactional memory, such as serializability and composability. The difference between TIC and open-nesting is *not* in the violation of serializability, however. Transaction punctuation also violates serializability for a punctuated atomic section as a whole, guaranteeing instead serializability for individual transaction parts. The difference is that open-nesting also violates *program order* (i.e., the logical order of operations in a single thread). For instance, when an open-nested transaction commits memory changes, the preceding changes in parent transaction data remain uncommitted. In this way, the effects of an open-nested transaction may appear to take place *before* parent transaction actions that caused the open-nested transaction's execution.

  Additionally, with open-nesting there is no guarantee (again, without explicit user intervention) that composing individually atomic operations in a single atomic section will yield an atomic operation. The excellent Agrawal et al. example [2] is illustrative: An open-nested transaction can be checking some shared memory location, m, and storing the result in a local variable, c, thus affecting the control (or data) flow of its parent transaction. Nevertheless, the transactional system is not aware that c is invalidated when m's contents change. The problem affects all (closed-nested) transactions that contain the open-nested one, making the operation containing the open-nested transaction non-composable with others. TIC punctuation raises similar issues, but, unlike in open-nesting, the programmer does not need to look inside the composed operations to determine that they may cause violations of atomicity: The type system warns when this is the case and requires the user to supply `expose...establish` clauses.

  In a sense, open-nesting *punctures* a transaction instead of *punctuating* it. Open-nesting exposes results both to and from the parent transaction, which can violate isolation and atomicity, respectively. We believe that a disciplined approach calls for transaction punctuation when this occurs. Nevertheless, this forces on the user the obligation to write correct `expose...establish` clauses at every nesting level. This may be undesirable, even though each level's reasoning is local (i.e., deals only with that level's invariants).

- There are elements that TIC just inherits from open-nesting models, since the main TIC feature (punctuation) is largely orthogonal to open-nesting. We profitably used TIC open-nesting features in our examples to stop propagation of the need for `expose ... establish` clauses in caller methods. This is the main use of open-nesting in TIC: When an externally visible operation can be reversed (with an undo action, possibly combined with an on-commit action to postpone some effects) the operation can be safely used in transactions without punctuating them and forcing the programmer to re-establish invariants. (For this to be valid, the possibility of other threads observing the external effects should not affect application-level correctness, as discussed in Section 2. This is a strict condition, which could be relaxed by adding locks to the model to let the user prevent operations that might conflict.) We believe that this is a modest, but desirable, use of open-nesting, which is well-aligned with the principles of correct open-nesting usage [45]: The open-nested transaction is at a separate, lower level of abstraction than its parent transaction.

  The TIC open-nesting model is currently limited. For instance, our set of compensating actions only contains on-abort (i.e., undo), and on-commit actions. More handlers (on-validate, on-top-commit) may offer extra power to

[4] Note that, originally, in the database setting, the term *open-nested* referred to "the 'anarchic' version of multi-level transactions" [23], which have no semantic restrictions between parent and child transactions (i.e., no semantic locking). Some authors follow this distinction in the transactional memory literature [13] but most, like us, use the term to include the possibility of locks for expressing high-level constraints [44, 45, 46].

open-nesting models, especially in conjunction with locking support (which brings out the need to distinguish validation from commitment) and with more advanced data sharing between forward and compensating actions. We have not found the current limitations to be seriously constraining, since transaction punctuation can replace many uses of open-nesting. In the future, the TIC open-nesting features can be enriched without affecting the main elements of the model.

5.2 The TIC Nesting Model

We next describe more precisely the current TIC nesting semantics. This allows us to answer questions such as "what is the execution context and concurrency model of an undo operation?" and "what happens when a suspending operation occurs inside an open-nested transaction?" This also exposes in more detail the current open-nesting support of TIC. We believe that this behavior can largely be tuned without affecting the main features of the model, but the current specification fits well with our notion of correct open-nesting usage, as we will demonstrate. We make an effort to distinguish the description of the nesting behavior from our current implementation, so we avoid referring to implementation artifacts (e.g., locks or read/write logs) except when explicitly comparing implementation techniques.

Let us first define how language constructs affect open- or closed-nesting. There are two kinds of relevant scoping language constructs, which nest dynamically. The first is methods with an `undo` annotation, as well as undo and on-commit actions themselves—we call their scope an *open* context. The second is atomic sections and root suspending operations—an *atomic* context. That is, we treat transactions inside an undo or on-commit action as open-nested (as Ni et al. [46] do), and we treat root suspending operations as if designating a separate transaction for the purposes of nesting behavior. The behavior of the system in each context is determined by the contexts surrounding it in dynamic scope: we process contexts from outer to inner, or caller to callee. The rules are simple: (We write "*context1* + *context2*" to mean that the rule applies for code whose immediate context is *context2* when *context2* is dynamically nested directly inside *context1* with no other context between.)

- *atomic* + *atomic* : The inner atomic section is closed-nested (more precisely, *flat-nested* [2]) in the outer one, forming a single transaction, for all intents.
- *atomic* + *open* : The code in the *open* context is outside the control of the transactional system. (If such code needs concurrency control, it should contain atomic sections, thus creating an *atomic* context.) The parent transactional context is recorded for possible later use. If the *open* context is a method with an `undo` annotation, the undo action is registered at the method's point of invocation. The action is not enabled, however. (It will not be unless the method includes an *atomic* context.)
- *open* + *open* : No change. Code in the inner context remains non-transactional, the recorded parent transactional context remains the same.
- *open* + *atomic* : The atomic section starts a new transaction, open-nested in the current parent transactional context. If the *open* context corresponds to a method with an `undo` annotation, the undo action at the point of method call return is enabled for execution during the parent transaction roll-back process.

Most of our treatment of open-nesting and compensating actions follows standard conventions. E.g., undo actions are called in the reverse order they are registered, an outer undo action prevents inner ones from being executed, etc. For issues of read-write and write-write conflicts between an open-nested transaction and its parent, we follow an approach similar to Ni et al. [46] (e.g., updating the parent's log). Nevertheless, some elements require clarification or are unconventional. These are listed next:

- Transaction punctuation affects only the current real transaction—i.e., all atomic sections up to the innermost open-nesting boundary, or up to the top-level atomic section (if no open-nesting has taken place). This is also consistent with our type system definitions of Section 4.
- All open-nested transactions roll back to their beginning point and never cause the parent transaction to abort. Recall that, per our previous definitions, open-nested transactions are started at the top-level atomic section of a method with an undo operation, as well as a method that is itself called as an undo or on-commit action.
- Open-nested transactions can see the local (i.e., non-shared memory) effects of their parent transaction, as well as a consistent view of shared memory. This permits multiple implementations: A consistent view of shared memory can be the current committed state, or the state that would result if the parent transaction were to commit at this point—in case, of course, its actions are still valid. (An explicit validation call is required for an implementation with lazy validation.)

To see the rationale for the above behavior, we next consider some examples.

It is a general requirement in open-nested transactions that on-abort actions (i.e., our undo operations) should be able to abort and commit independently (i.e., restart from their beginning when retrying, instead of restarting the parent transaction). The requirement comes from the use of undo operations: it makes no sense for an undo to cause a parent abort, since it was the parent abort that necessitated the undo in the first place.

In TIC, it is natural to extend the above requirement from just undo actions to all kinds of open-nested transactions because of the possibility of transaction punctuation. Consider the following example: (We try to keep the examples concise

by using pseudo-keywords. We use "openatomic" in place of a separate method with an undo annotation and an atomic section in it, and "undo" for an undo method with an atomic section in it.)

```
atomic { ...
  openatomic {
    ... p = expose(sbrk()); ...
  } undo { ... }
}
```

The top part of the open-nested transaction can commit independently at the point of suspension (call to sbrk—used just as an example of a suspending operation). If, however, the bottom part of the open-nested transaction (i.e., after the sbrk) needs to abort and retry, it cannot restart at the top of its parent transaction (since this would repeat the top part of the open-nested transaction, as well, and the suspending operation has already committed its results to memory). Instead, the punctuated open-nested transaction commits and aborts independently of its parent.

As stated above, open-nested transactions see a consistent view of shared memory (i.e., cannot see partial transaction results, unless these come from the parent transaction and have not been invalidated). Typical implementations of open-nesting are pessimistic/lock-based with *undo logging*: Information is kept to allow undoing shared memory effects. In this case, shared memory already reflects the uncommitted effects of the parent transaction, so it is reasonable for the open-nested transaction to access them. An optimistic implementation, however, will typically store the parent transaction's effects in a log until the parent commits (*redo logging*). In this case, it may not be reasonable to allow the open-nested transaction to see the uncommitted effects of its parent. For instance, in the case of an undo operation, the parent transaction's state is not a valid state. (The undo operation is called exactly because the parent transaction encountered interference and aborted *after* performing operations that may be invalid.) Furthermore, the possibility of exposing state that only exists in the parent's log makes the programming model awkward. Consider the following example: (Interestingly, a very similar example was shown independently in [44].)

```
// n originally 0
atomic {
  n = 1;
  openatomic {
    n++;
  } undo { n--; }
  ...
}
```

The decrement operation is not a correct undo action for the increment. Incrementing the shared memory variable inadvertently exposes the effects of the parent transaction. A correct undo requires reverting to the original value of the variable. This is not possible unless we expose to the user a richer set of values than just the parent transaction's view of a variable. (Other alternatives include enabling the above code to execute correctly by causing cascading aborts for all transactions that happen to read exposed data. This is complex and suffers from heavy overheads, however.)

To circumvent the above problems, in our optimistic implementation we do not allow an open-nested transaction to observe the uncommitted shared memory effects of its parent transaction. Instead, the open-nested transaction only observes the latest committed values to shared memory. This is somewhat counter-intuitive, but in line with correct use of open-nesting. Moravan et al. [44] proposed the following discipline condition for open-nested transactions: An open-nested transaction should not write any data written by the parent transaction. Indeed, we believe that the condition should be even stronger: *An open-nested transaction should never perform a write that is (control- or data-)dependent on shared data written by its parent transaction*. Without the stronger condition, it is easy to violate global invariants (that depend only on shared data, and which all transactions would respect if they were isolated) by having an open-nested transaction expose uncommitted state. (With the stronger condition, an open-nested transaction can inadvertently only violate invariants that involve both global and local data.) An easy way for the programmer to ensure the condition is to not allow an open-nested transaction to access any shared data written by the parent. In this case, our requirement for a "consistent view of shared memory" in an open-nested transaction is sufficient to make the code oblivious to implementation specifics, such as whether the parent transaction's uncommitted writes are visible.

The above principle also covers well our intended use of open-nesting in TIC: We employ open-nesting to hide operations that are reversible at the application level. For an operation to be reversible, it should not be exposing its parent transaction's data to other threads (which is a potentially irreversible effect) in a way that affects application correctness. Thus, it is a good property for an open-nested transaction to never access shared data from its parent. This follows the conventional wisdom about strict abstraction separation of open-nested actions: Quoting Moss [45], "in the open nesting case the parent and child execute *at different levels of abstraction*." Our kmalloc example of Section 3 is exactly such a case.

6. Related Work

Modularity is a central recurring theme in the development of models and mechanisms for concurrent programming, from supplanting raw semaphores with conditional critical regions [35] and then monitors [36], through refinement of monitor and condition variable semantics to reduce the fragility of process coordination [39], through introduction of the transaction concept [21] to decouple maintenance of consistency from definition of individual data structures, up

to more recent work on programming language support for transactions.

The TIC model draws inspiration and ideas from several earlier models, both transactional and monitor-based. The programmer requirement to reestablish the global invariant just *before* `Wait()` and to reestablish local invariants just *after* follows directly from the correctness reasoning already established in the earliest definitions of monitors [36]. Our approach to transaction suspension is somewhat analogous to mechanisms that punctuate atomicity in monitors in a controlled way (e.g., *serializers* [4]). Punctuated transactions are also somewhat analogous to *chain transactions* in database systems [23] and their variants for persistent programming [7] and workflow systems [48], but after the first part of a punctuated transaction commits at `expose`, it is independent of subsequent parts, exposes its results, and can never be rolled back. Our type system handling of suspending operations is close to conventions for monads in Haskell, and the Haskell type system has been used before for the purpose of identifying non-transactional operations [29].

Herlihy and Moss proposed transactional memory systems more than a decade ago [32], and design of hardware support has accelerated in response to the widespread availability of multi-core computer hardware. Several hardware transactional memory projects, including logTM [43], TCC [26, 12, 42] and others [32, 47, 15, 49, 8, 51] are exploring the design space for hardware support for transactional memory. Proposals vary regarding whether conflicts are detected eagerly or lazily, whether changes are made directly to memory (with old values stored elsewhere in case a transaction is aborted) or written only when committing (making abort cheap but commit more expensive), how external or non-transactional actions are treated (e.g., whether and how to support open nesting), etc. In principle, a programming model should hide from the programmer whether the underlying mechanisms are partly or wholly implemented in hardware, as well as operational details of the implementation. In practice, it is unlikely that a programming model can entirely hide these design choices, at least insofar as they impact cost.

A key and troublesome interaction between a high-level program model and the underlying implementation involves interaction of transactions with memory accesses outside transactions. *Strong atomicity*, as defined by Blundell et al. [9], essentially treats otherwise unguarded accesses as small transactions. Most implementations, however, can be expected to provide only some form of *weak atomicity*, which (like memory models weaker than sequential consistency) opens a plethora of difficult questions, right down to interactions between high-level transactions and the memory model governing individual accesses. Grossman et al. [25] present a classification and examples of isolation and ordering anomalies that may or may not be allowed under varying weak atomicity models, and provide the beginnings of an approach to reasoning about weak atomicity and memory models with (weak and strong) happens-before relations.

TIC inherits a weak atomicity model from the underlying libstm and TL2 libraries [28, 17], and so can behave in unintuitive ways if shared variables are accessed outside transactions. Despite recent reports of achieving strong atomicity at modest cost, through extensive optimization [50], our expectation is that (as in memory consistency models) performance considerations will continue to make weaker atomicity guarantees a practical necessity. The basic `Wait` and `establish` features of TIC should not introduce complications beyond those of other closed nesting transaction systems, except that `Wait` is only guaranteed to notice conditions changed by transactions. `suspending` methods, on the other hand, could be subject to ordering anomalies described by Shpeisman et al. [50] and require special scrutiny.

Nearly all STM proposals require some form of transaction roll-back, either abandoning a temporary, thread-local record of memory writes (where conflict detection is performed lazily, at commitment time), or else undoing the effects of writes to memory. Schemes that use write locks but not read locks must re-validate reads before committing, and may be forced to roll back memory effects. Schemes that use both write and read locks (as in strict 2-phase locking) may be forced to roll back by deadlock [20]. All such approaches face the basic problem of data dependencies with I/O (read-after-write) [24]. The only approaches that can avoid this problem are purely pessimistic approaches that also statically prevent deadlock, rather than dynamically detect it [41, 19, 33], but these require complex program analysis which is apt to be non-modular (hence expensive) or else rely on extensive program annotation. The safe (but sometimes inconvenient) way to perform I/O in TIC is with punctuated transactions, but the back door of open nesting is ajar, with the usual risks.

Harris has described an approach to external operations with effects directly to the heap but isolated from other transactions, and using two-phase commit and buffering to obtain transactional behavior for I/O [27]. Unlike either open nested transactions or TIC, Harris's approach does not expose intermediate state. On the other hand, in addition to requiring a good deal more machinery to properly wrap and protect external operations, this approach is not sufficient for read-after-write idioms, or for complex external operations that both mutate state and return a value. Consider an operation like `sbrk` in the example we saw in Section 3. `sbrk` returns a value that the application needs to use, yet performs a state change in the process. Thus, it can neither be delayed (due to the read) nor replayed (due to the state change). Even if the interface of `sbrk` is changed to be idempotent (e.g., by adding a version number), the problem remains: A transaction that retries might not call `sbrk` the second time, and no other transaction might be able to consume that memory.

Chung and colleagues' study of common patterns in current (locking-based) programs [16] shows that, if we simply translate current code to transactions, the average nesting depth is modest and non-transactional operations are not common. It remains to be seen whether these distributions hold when programmers can use transactions directly, and one may surmise that, as hierarchical composition is one of the chief motivations for adopting a transactional style of programming, typical nesting depth could increase in programs written directly in that form. At the very least, though, Chung's results imply that measures taken to accommodate waiting and external effects in nested transactions must not have a substantial cost for the (so far) common case of transactions that are only shallowly nested and affect only memory.

Checkpoints have an interesting relation to transaction roll-back. The purpose of both is to return a system to a globally consistent state. Checkpointing does this by either maintaining a global snapshot, or retaining a set of local snapshots that form a "consistent cut" representing a state that the system could have reached (even if the local snapshots were never current at precisely the same time) [14, 18]. Recent work developing transparent checkpoint facilities for Concurrent ML [53] is an example of maintaining enough dependence information to roll back several interacting threads to a consistent state; this is what in database terms would be called *cascading abort*. In contrast, all STM systems to our knowledge are designed to make cascading abort unnecessary. For example, systems that use write locks but not read locks may need to roll back if validation at commit time reveals stale read values, but the write locks (which permit anti-dependence but not dependence between uncommitted transactions) ensure that memory writes can be backed out without aborting other transactions. Recall that in TIC our `establish` statements are only for reestablishing local consistency; global consistency is established before `expose`.

7. Conclusions

A key goal of transactional programming is to avoid or minimize non-local reasoning about thread interactions. To achieve this goal, it is essential that transactions are compositional in the sense that the decision to make one method transactional does not require "looking inside" other methods to determine whether they are also transactional. At the same time, transaction support must be sufficiently general that common operations can be enclosed in transactions. For standard imperative programming patterns, external operations such as I/O cannot be poison pills that prevent using transactions in the whole tree of calling methods. This creates a tension between isolation (to simplify reasoning) and communication (to get work done). The TIC model mostly retains the closed nesting model of transactions, with a small number of careful extensions to better accommodate common programming idioms like barriers and conditional waiting as well as other operations that break the standard closed nesting semantics.

While the TIC model necessarily punctuates transactions, sacrificing isolation for communication at controlled points, the type system prevents *unanticipated* interaction between threads by making the possibility of suspension visible in method signatures, and by requiring the programmer to acknowledge the potential interruption at the point of the method call.

We have reengineered substantial existing lock-based programs to use the TIC model, confirming that it meets our goal of providing a more general, convenient programming model while preserving the main benefits of transactional memory.

Acknowledgments

This work was funded by the NSF under grant CCR-0735267 and by LogicBlox Inc.

Earlier discussions with Tony Hannan on transactional memory and LihChyun Shu on concurrency control helped form the background of this paper.

We would like to thank the anonymous reviewers for many valuable comments that helped improve the paper.

References

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, Ottawa, Ontario, Canada, 2006. ACM Press.

[2] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 70–81, San Jose, California, 2006. ACM Press.

[3] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[4] Russell Atkinson and Carl Hewitt. Synchronization in actor systems. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 267–280, Los Angeles, California, 1977. ACM Press.

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, November 2000.

[6] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 114–124, 2001.

[7] Stephen Blackburn and John N. Zigman. Concurrency - the fly in the ointment? In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3)*, pages 250–258, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[8] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer architecture*, pages 24–34, San Diego, California, USA, 2007. ACM Press.

[9] Colin Blundell, E. Christopher Lewis, and Milo M. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17, 2006.

[10] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr 2006.

[11] Brian D. Carlstrom, JaeWoong Chung, Hassan Chafi, Austen McDonald, Chi Cao Minh, Lance Hammond, Christos Kozyrakis, and Kunle and Olukotun. Transactional execution of java programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. Oct 2005.

[12] Brian D. Carlstrom, JaeWoong Chung, Hassan Chafi, Austen McDonald, Chi Cao Minh, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Executing Java programs with transactional memory. *Science of Computer Programming*, 63:111–129, 2006.

[13] Brian D. Carlstrom, Austen McDonald, Michael Carbin, Christos Kozyrakis, and Kunle Olukotun. Transactional collection classes. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 56–67, San Jose, California, USA, 2007. ACM Press.

[14] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[15] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *ASPLOS-XII: Proceedings of the 12th International conference on Architectural support for programming languages and operating systems*, pages 347–358, San Jose, California, USA, 2006. ACM Press.

[16] JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. The common case transactional behavior of multithreaded programs. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*. Feb 2006.

[17] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In Shlomi Dolev, editor, *Distributed Computing, 20th International Symposium (DISC)*, volume 4167 of *Lecture Notes in Computer Science*. Springer, 2006.

[18] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[19] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL '07: Symposium on Principles of Programming Languages*, pages 291–296. ACM Press, 2007.

[20] Robert Ennals. Software transactional memory should not be lock free. Technical Report IRC-TR-06-052, Intel Research Cambridge, 2006. Available from `http://berkeley.intel-research.net/rennals/`.

[21] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[22] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Prentice Hall, 2005.

[23] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[24] Dan Grossman. The transactional memory / garbage collection analogy. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Essays Track*. ACM SIGPLAN, October 2007.

[25] Dan Grossman, Jeremy Manson, and William Pugh. What do high-level memory models mean for transactions? In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 62–69, San Jose, California, 2006. ACM Press.

[26] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[27] Tim Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005.

[28] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, pages 388–402, Anaheim, California, USA, 2003. ACM Press.

[29] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton-Jones. Composable memory transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, Jun 2005.

[30] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming language design and implementation*, pages 14–25, Ottawa, Ontario, Canada, 2006. ACM Press.

[31] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented programming systems, languages, and applications*, pages 253–262, Portland, Oregon, USA, 2006. ACM Press.

[32] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[33] Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.

[34] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 82–91, San Jose, California, 2006. ACM Press.

[35] C. A. R. Hoare. Towards a theory of parallel programming. In *International Seminar on Operating System Techniques*, 1971.

[36] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[37] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. Mcrt-malloc: A scalable transactional memory allocator. In *ISMM '06: Proceedings of the 2006 international symposium on Memory management*, pages 74–83, Ottawa, Ontario, Canada, 2006. ACM Press.

[38] Anthony Kay. AlphaMail. `http://sourceforge.net/projects/alphamail`, January 2007.

[39] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, 1980.

[40] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time java. *rtss*, 0:62–71, 2005.

[41] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 346–358, Charleston, South Carolina, USA, 2006. ACM Press.

[42] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer architecture*, pages 69–80, San Diego, California, USA, 2007. ACM Press.

[43] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006.

[44] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logTM. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, San Jose, California, USA, 2006. ACM Press.

[45] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, Dec 2006.

[46] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Processing*, San Jose, California, March 2007.

[47] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[48] Andreas Reuter and Friedemann Schwenkreis. Contracts — a low-level mechanism for building general-purpose workflow management-systems. *Bulletin of the Technical Committee on Data Engineering*, 18(1), 1995.

[49] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO '06: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[50] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design Implementation*, pages 78–88, San Diego, California, USA, 2007. ACM Press.

[51] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer architecture*, pages 104–115, San Diego, California, USA, 2007. ACM Press.

[52] Gerhard Weikum and Hans-Jorg Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. 1992.

[53] Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional programming*, pages 136–147, Portland, Oregon, USA, 2006. ACM Press.