Integrating Transactional Memory into C++ *

Lawrence Crowl †
Google
lawrence@crowl.org

Yossi Lev
Brown University and
Sun Microsystems Laboratories
yosef.lev@sun.com

Victor Luchangco
Sun Microsystems Laboratories
victor.luchangco@sun.com

Mark Moir
Sun Microsystems Laboratories
mark.moir@sun.com

Dan Nussbaum
Sun Microsystems Laboratories
dan.nussbaum@sun.com

Abstract

We discuss the integration of *transactional memory* into the C++ programming language. We take a decidedly pragmatic approach in this paper: Our goal is to induce minimal changes consistent with implementability, usable transactional semantics, and the prevalent styles of C++ programs. In particular, we want to avoid designing a new language; rather, we want to enable incremental adoption of transactional memory into existing C++ code bases. The contribution of this paper is a careful analysis of the options rather than a final solution. In some cases, reasonable alternatives exist and experience in implementation and use must guide the final choice.

1. Introduction

With the advent of chip multiprocessors, it is increasingly important for software to make effective use of parallelism to exploit advances in technology. Thus, programmers must increasingly learn to write concurrent programs. This will not be easy: programmers today typically rely on a combination of locks and conditions to prevent concurrent access by different threads to the same shared data. Although this approach simplifies reasoning about interactions by allowing programmers to treat sections of code as "atomic", it suffers from a number of shortcomings.

First, data and locks are associated *by convention*; failure to follow the convention is a common source of subtle bugs that are hard to reproduce and hard to diagnose. When the convention is followed correctly, it is common to lock too conservatively, resulting in poor performance. Locks also introduce a difficult granularity tradeoff: *coarse-grained* locks use fewer locks for more data, generally resulting in simpler code but poor scalability; *fine-grained* locks protect less data, resulting in better scalability, at the cost of additional locking overhead and significantly increased programming complexity, which makes applications harder to understand, debug, and maintain. Furthermore, operations implemented with locks cannot be directly composed with each other, a significant disadvantage for software engineering.

In reaction to these problems, the *transactional* model of synchronization has received attention as an alternative programming model. In transactional programming, code that accesses shared memory can be grouped into *transactions*, which are intended to be executed atomically: operations of different transactions should not appear to be interleaved. A transaction may *commit*, in which case all its operations appear to take place atomically, or *abort*, in which case its operations appear not to have taken place at all. If two transactions conflict—that is, if they access the same object, and at least one of them writes it—then the conflict must be resolved, often by aborting one of the transactions. An aborted transaction is typically retried until it commits, usually after taking some measures to reduce contention and avoid further conflicts. System support for transactions has been investigated in hardware [2, 13, 20, 34, 35], in software [14, 15, 18, 24, 30, 40, 33, 43], and in schemes that mix hardware and software [6, 25, 27, 34, 38].

In this paper, we discuss the integration of transactional memory into the C [22] and C++ [23] programming languages. We concentrate our discussion on C++, because the issues for C++ are a superset of those for C.

We have two pragmatic goals for our design:

useful to working programmers Transactions should be integrated into C++ in a way that is usable by everyday programmers. When considering various options for integration, we attend not only to the semantics of C++, but also to the prevalent styles of C++ programming. In particular, we consider it important that programmers can begin to use transactions gradually in existing code bases, rather than having to write entire transactional applications from scratch. Such attention is crucial for the

† Work done while at Sun Microsystems, Inc.

result to be useful in practice: adoption of transactional memory will almost certainly be incremental.

sooner rather than later Several subtle issues that arise in integrating transactional memory into C++ cannot be fully resolved without experience in implementation and use. Furthermore, many facets of transactional memory design have not been thoroughly explored, and we would similarly benefit from experience in implementation and use. To gain such experience, we prefer an experimental implementation sooner rather than a highly developed implementation later. An early implementation makes it possible to gain crucial experience in preparation for more comprehensive solutions. As such, our integration with C++ should prefer minimal language changes consistent with implementability.

To begin, we enumerate several desiderata for our design:

composability Transactions should compose, which means they must be nestable, because transactions might be behind abstraction boundaries such as functions or classes.

minimality We should minimize changes to the language, introducing as few new features as possible and preserving the style of C++ programming.

predictability The mechanisms of transactions should have effects that are predictable by ordinary programmers.

orthogonality Consequently, where possible, transactions should be orthogonal to other language mechanisms.

implementability Transactions should be within reach of current implementation technology. Although we do not discuss implementation details for most of the issues covered in this paper, practical implementability has guided our deliberations.

incremental development We generally prefer to omit any feature that can be added later without breaking correct existing code if we are not sure the feature is needed, or believe more experience is needed before deciding how to best include the feature. Similarly, the treatment of incorrect programs can be improved over time; for example, we might allow undefined behavior given a certain programming error in an early implementation, and later add support for reporting illegal behavior when it occurs, and later still add compiler support for preventing such behavior.

incremental adoption We should avoid barriers to beginning to use transactions. Thus, we eschew designs that impose significant overhead on code that does not use transactions, as well as designs that require recompilation of code not used in transactions (for example, legacy libraries).

scalability The base mechanisms should allow applications to scale reasonably.

efficiency Transactions should not induce unacceptable overhead, particularly in parts of the code that do not handle exceptions or use transactions.

The contribution of this paper is a careful analysis of the options, rather than a final solution. In some cases, reasonable alternatives exist; experience in implementation and use must guide the final choice.

2. Background

Transactional memory is adapted from the notion of transactions in database systems [11], which were originally designed to simplify error handling while protecting data from corruption: Instead of figuring out what went wrong and repairing the damage when an error or some other unexpected situation arises, the transaction is simply aborted and, if appropriate, retried.

In a concurrent system, transactions have two additional benefits: By guaranteeing an all-or-nothing semantics, transactions insulate different parts of a program from each other and increase the granularity of atomic actions. Thus, they provide a kind of modularity for concurrent programs that reduces the possible behaviors of concurrent programs, and thus simplifies reasoning about them. Also, the ability to recover from errors by aborting transactions enables a simple but powerful way to manage contention: Rather than ensuring that a transaction experiences no conflict (by holding locks for shared data, for example), we can optimistically assume that it will not, and then check this assumption before committing the transaction. If conflict did occur, we can simply abort and retry the transaction.

Herlihy and Moss [20] first proposed *transactional memory* as a hardware mechanism leveraging caching and the cache-coherence protocol to provide small transactions. Shavit and Touitou [42] introduced *software transactional memory*, and gave a nonblocking implementation for it, but one that was too inefficient and inflexible to be practical, and the idea lay fallow for almost a decade. Interest was reignited by converging developments in both software and hardware.

In hardware, Rajwar and Goodman exploited speculation machinery to elide the acquisition of locks for critical sections that encountered no conflict [36, 37]. Several other research groups also used caching and speculation, and proposed new mechanisms, to provide other transactional behavior, either explicitly or "behind the scenes" (e.g., [2, 13, 32, 34, 38], and many others). In software, Herlihy et al. [18] introduced *dynamic software transactional memory*, which was plausibly efficient and flexible enough to use in practice. They achieved this by using *obstruction-freedom* [16], a weak notion of nonblocking progress, that separates out the conditions necessary to guarantee safety and enable progress from those used to guarantee progress in practice. The latter can be provided by separable contention management mechanisms, which have since been ex-

plored by several researchers, particularly at the University of Rochester [12, 28, 30, 41]. Harris and Fraser provided the first language-level and run-time support for transactional memory [14]. Since then, transactional memory has been a hot topic of research, both in academia and industry, resulting in many advancements and new directions for hardware, software, and combinations thereof (e.g., [1, 5, 6, 7, 8, 10, 15, 17, 31, 40, 45] and many others).

In designs of transactional memory, a central question to ask is: What is the basic unit of data for the transactional memory? For hardware, this is primarily a question of granularity. However, for software, it is also a question about the underlying infrastructure. In particular, we distinguish *object-based* and *word-based* transactional memory.

Object-based transactional memory [9, 17, 19, 29, 31] relies on the underlying object infrastructure provided by implementations of object-oriented languages. This approach is attractive because much of the work required to implement transactional memory can leverage the object infrastructure, which typically already maintains metadata about the objects and mediates access to those objects. Thus, object-based systems can provide transactional access with relatively little additional overhead over the cost of nontransactional access. Also, the language may also provide type safety guarantees that a transactional memory implementation can leverage to ensure certain properties of data that can be accessed transactionally.

In contrast, a word-based system is based on transactions over actual memory, not the abstractions that might be represented by that memory. In this sense, all proposed hardware transactional memory systems are word-based (however, hardware transactional memory implementations often manage data at the granularity of a cache line). A word-based software transactional memory without specialized hardware support must typically maintain its metadata separate from the data being accessed by the transactions, and insert code around every memory access to maintain the metadata and detect conflicts.

3. Basic Design

3.1 Object-based vs. word-based approaches

When integrating transactional memory into C++, we have to choose between an object-based approach and a word-based approach. As discussed earlier, an object-based approach might leverage C++'s existing class mechanism, and would likely introduce less overhead. This is the approach taken, for example, for the Rochester Software Transactional Memory Runtime [31].

However, C++ supports both object-oriented programming and the traditional procedural programming style of C. For efficiency and compatibility with C, the C++ standard [23] distinguishes two kinds of data types: *POD* (i.e., "plain old data") and *non-POD*. C++ does not maintain an object infrastructure for POD types, and indeed, not even for all non-POD types—that privilege is reserved for polymorphic types. Thus, taking an object-based approach would preclude transactional manipulation of many common data types. We chose the word-based approach for this reason.

We considered adopting a combined approach, leveraging the object infrastructure for polymorphic non-POD types, and using a word-based transactional memory for POD types and non-polymorphic non-POD types. However, this would introduce significant complexity to the design, introducing many new issues about how the two approaches interact, contradicting our principle of minimality.

3.2 Transactional control construct

Having decided on a word-based transactional memory, we need a control construct to introduce transactions. We chose the simplest and most natural construct for C/C++: a new control statement, which we designate using the keyword `transaction`:[1]

`transaction` *statement*

Typically, *statement* would be a compound statement. The transaction ends, either committing or aborting, when control exits the statement. Deciding when it should commit and when it should abort is a major aspect of the design, which we discuss in Section 3.5. By the orthogonality principle, the semantics of code executed within a transaction should, as much as possible, be the same as if it were executed outside a transaction. However, full orthogonality is not possible; we discuss various operations and their usability within transactions below.

Like other structured control constructs, a `transaction` statement introduces a new local scope: variables declared ouside a `transaction` statement are accessible within the statement, but those declared within the statement are not accessible outside the statement.

However, unlike most C++ structured control constructs, it does not make sense for a `transaction` statement to affect only the code in its static extent: doing so would prohibit the use of function calls within such statements, a severe restriction. Instead, a `transaction` statement must apply to all code executed within its dynamic extent. In this way, it is similar to the `try` statement, which affects `throw` statements within its dynamic extent. This similarity is not surprising because transactions and exceptions are both intended for restricting and recovering from anomalous execution. This overlap in intended purposes makes the interaction between transactions and exceptions more interesting, and we discuss issues that arise in Section 4.

[1] We do not use `atomic` as a keyword because the ISO C++ standards committee has a well developed proposal to use `atomic` as the name for a predefined template for machine-level atomic operations such as load, store, fetch-and-add, and compare-and-swap. Thus, using `atomic` as the keyword would likely conflict with the next release of the C++ standard.

The `transaction` statement is not quite sufficient: it cannot express that a constructor with a member initializer list should be executed atomically. That is, in the constructor

```
myclass::myclass() :
    membervar(30), another(20) { ... }
```

the member initializations are not part of a statement and hence cannot be wrapped by a `transaction` statement. Still, because constructors may execute arbitrary code, the programmer might want the member initializations and the body of the constructor to be executed atomically using a transaction. A similar problem arises with catching exceptions in member intializations, which is solved by inserting `try` into the constructor syntax, as in, for example:

```
myclass::myclass() try :
    membervar(30), another(20) { ... }
    catch ( ... ) { ... }
```

We adopt the analogous solution to enclose a constructor's member initializations and body within a transaction. That is, we can write:

```
myclass::myclass() transaction :
    membervar(30), another(20) { ... }
```

An alternative to using `transaction` statements for introducing transactions is to support "transactional functions", that is, functions whose call and return are the boundaries of a transaction. Such a function would be convenient for defining a transaction that returns a value. However, although transactional functions can be implemented with `transaction` statements, because functions may be granted `friend` access to some classes: splitting out a block of code into a separate function may make critical variables inaccessible to that code.

In addition, function boundaries are too coarse. In particular, argument evaluation is not considered part of the called function, so transactions that desire to include the argument evaluation must be moved to the calling function. For example, given a transactional function `f`, in the function call

```
f(x++);
```

the increment of `x` is not part of a transaction. With transactional functions, there is no mechanism for including argument evaluation in the transaction without rewriting the surrounding code, splitting out the call into a separate function. On the other hand, a `transaction` statement can easily include or exclude the argument evaluation.

```
tmp = x++; transaction{ f(tmp); }
                            // args not subsumed
transaction{ f( x++ ); } // args subsumed
```

Therefore, by the principle of minimality, we omit transactional functions from our initial design.

3.3 Strong vs. weak atomicity

An important issue concerns the interaction between transactions and nontransactional memory accesses, i.e., those not executed inside a transaction. One possibility is to provide *strong atomicity* [3], which guarantees that nontransactional memory accesses are atomic with respect to transactions. Implementations that do not make this guarantee are considered to provide *weak atomicity*, in which weak or no semantic guarantees are made if the same memory locations are *concurrently* accessed by transactions and by nontransactional memory accesses, essentially requiring programmers to avoid such behavior.

Strong atomicity provides simpler semantics, and thus is preferable if we can achieve it efficiently. This is possible with appropriate hardware support, or in an object-based software transactional memory, where all memory accesses can be mediated by a run-time system (i.e., in a managed language). One conceptually simple approach is to treat any nontransactional memory access as a single-access transaction. However, in C and POD C++, without special hardware support, we generally cannot transform a nontransactional access into a transactional access at run time.

An alternative is to recompile all code, transforming all ordinary memory accesses into short transactions. This approach has two significant problems. First, transforming all ordinary memory accesses into transactions imposes significant overhead on the *entire* application, not just the parts that use transactions. While optimizations may reduce this overhead, even very aggressive optimization is unlikely to eliminate it entirely, and in any case is incompatible with our goal of achieving initial implementations sooner rather than later.

Second, requiring recompilation of all code is not compatible with our desire to support incremental adoption: we would like to enable programmers to start using transactions while only recompiling the code that uses transactions or may be called from within transactions.

With most transactional memory implementations, as long as it is guaranteed that no variable is *concurrently* accessed both by transactions and by nontransactional accesses, there is no problem. This raises the question of how to follow this rule. Static methods for enforcing this rule—for example based on having transactional and nontransactional types—are too inflexible. The approach would be analogous to the familiar C++ problem of introducing `const` qualification into a program written without `const`: the introduction of one `const` qualifier may require its introduction in every function header in a call chain, which is sufficiently disruptive that it has earned the term *const poisoning*. Dynamic solutions have been proposed that aim to handle these issues transparently to the programmer [26], but these impose some overhead on all code, not just transactions, and are not sufficiently mature for inclusion in an

initial implementation, especially because, as developed so far, they assume an object infrastructure.

Primarily for these reasons, we decided to require programmers to follow the rule that forbids concurrent transactional and nontransactional accesses to the same data, but not to enforce it, thereby allowing transactional memory implementations that do not enforce strong atomicity. Although this approach is less safe, we believe that is consistent with the spirit of C/C++ programming. Furthermore, tools to help programmers avoid and/or detect violations of the weak atomicity rules may eventually be successful enough that strong atomicity is not required.

Choosing not to support strong atomicity initially is consistent with our goals of incremental development and adoption. We can decide later based on experience whether strong atomicity is needed and programs that are correct under weak atomicity will remain correct if strong atomicity is adopted.

3.4 Nesting

The desire to use programmer-defined abstractions within a transaction implies the need to permit nested transactions. Otherwise, the use of a transaction as an implementation detail would be exposed in the interface. Basic composition can be achieved with simple *flat nesting*, in which a nested transaction is simply considered to be part of the dynamically enclosing (i.e., *parent*) transaction. Thus, a nested transaction does not abort or commit when it is completed; rather, control simply passes to the parent transaction. Flat nesting can be implemented with a simple nesting depth counter, so that we commit only the outermost transactions, which subsumes all nested transactions. Given this simplicity, an initial implementation should support *at least* flat nesting.

A variety of richer nesting models have been proposed. For most of them, we believe more research is needed to achieve robust semantics and implementations, and more experience is needed to determine whether they are needed (see Section 3.8 for some examples).

One exception is *closed nesting*, in which nested transactions commit or abort upon exit. If a nested transaction commits, its effects become visible only to the parent transaction; if it aborts, the parent transaction "stays alive", while the effects of the the nested transaction are discarded. The nested transaction can then be retried independently of its parent.

While an initial implementation could provide just flat nesting, it is likely that we would want to support closed nesting sooner rather than later. For very basic transactions, when a nested transaction aborts, we could simply abort the parent and retry the whole transaction. This is achieved by simple flat nesting and is indistinguishable from closed nesting apart from possible performance differences. However, various language features we may want to consider require "real" closed nesting. For example, if exceptions abort the transactions they escape (as described in Section 4.2), then we must abort such transactions, while keeping alive an enclosing transaction that catches the exception. Similarly, it may be desirable to be able to explicitly abort a nested transaction (see Section 5.2) and either retry it or try an alternative. Other features such as the `orElse` construct proposed for STM Haskell [15] similarly require closed nesting.

3.5 Control flow into, out of and within transactions

We now consider how transactions affect the flow of control. As long as control stays within (the dynamic extent of) a transaction, the semantics should be the same as for nontransactional code, as in existing C++. Thus, we need only consider when control crosses a transaction boundary.

Entering a transaction

Entering a transaction is mostly straightforward because transactions are introduced by a structured control construct. We believe that jumping into the middle of a transaction should not be allowed. This restriction is unlikely to be a problem in practice, and it greatly simplifies implementation: code to set up a transaction can simply be emitted at the beginning of the code implementing the transaction. Furthermore, we can relax this restriction later, if necessary, without breaking correct programs for existing implementations.

There are two ways to "jump into" a transaction, via `goto` and via `longjmp`. C++ already restricts jumping via `goto` into a compound statement with non-POD variables; it is simple to extend this restriction to `transaction` statements. Since `goto` must jump to a label in the same function, the compiler can verify that this restriction holds.

It is not so simple, however, for `longjmp`, which jumps to the point of the thread's most recent invocation of `setjmp` with the same "jump buffer", restoring the environment saved by that `setjmp` (the behavior is undefined if the function that called `setjmp` has already returned). We want to allow the `longjmp` in each of the following cases:

- the corresponding `setjmp` was called within the same transaction
- it was not called within any transaction
- it was called within some ancestor transaction (i.e., `longjmp` was called in the dynamic extent of the transaction that called the `setjmp`).

In the first case, no transaction boundary is crossed, so, by orthogonality, the behavior of `longjmp` should not be changed. In the latter two cases, the thread exits the current transaction (and possibly others, if the transaction is deeply nested), but it does not enter any transaction; we discuss the semantics of exiting a transaction below. In any other case, doing the `longjmp` requires entering a transaction without executing the beginning of that transaction, which we do not want to support.

We can distinguish these cases using a simple mechanism based on unique transaction identifiers, which are present in

most implementations. `setjmp` stores the current transaction identifier (assumed to be nonzero) in the jump buffer; if the `setjmp` is not within a transaction, it stores zero instead. When `longjmp` is called within a transaction, if the current transaction identifier is the same as the last one stored in the jump buffer, then we are in the first case above: no transaction is entered or exited. If the buffer has a zero instead, then we are in the second case.

For flat nesting, in which a nested transaction is subsumed by its parent (and thus does not have a separate transaction identifier), this covers all the cases we want to allow; if the buffer contains a transaction identifier other than the current one, we are in the unsupported case.

To support closed nesting, we can maintain a stack of identifiers of transactions that have not yet completed (i.e., the current transaction and its ancestors), and use this stack to determine whether the identifier stored in the buffer belongs to an ancestor of the current transaction. If so, we are in the third case above; otherwise, we are in the unsupported case. If the `longjmp` is not within a transaction, then it is an error unless the jump buffer contains zero.

Exiting a transaction

For exiting a transaction, it is simplest if the transaction commits: the desired semantics is simply the semantics of an ordinary C++ program in which the entire transaction happens to execute without conflicting with any other threads. Thus, we mandate this behavior for all "normal" exits. This begs the question, what constitutes a normal exit? Also, we must specify what happens when a transaction aborts. In particular, we must specify which effects are discarded, and what code is executed after the transaction aborts. We consider the various ways a thread may exit a transaction.

Reaching the end of the `transaction` statement is clearly a normal exit. Similarly, `return`, `break`, `continue` and `goto` statements are used as ways to control flow within a single function body, and most C++ programmers would be surprised if these were not treated as normal exits. Thus, we choose to commit a transaction that is exited using one of these statements.

On the other hand, `longjmp` discussed above is typically used to abandon a large task without ending the entire process. This pattern is inconsistent with committing the transaction, so we choose to abort transactions when they exit via `longjmp`. We can use the algorithm described above to detect when a `longjmp` would exit a `transaction` statement and abort the appropriate transaction. In this case, all the effects of the transaction are discarded, and the `longjmp` is executed afterwards. The only information transferred out of the transaction is the `int` passed as an argument to `longjmp`. Control resumes at the appropriate `setjmp`.

A transaction may also be exited by throwing an exception. This is a more subtle case, and we discuss it in detail in Section 4.

Finally, a transaction can be aborted due to a conflict with another thread, or be interrupted by an asynchronous signal. We believe that the programmer should not worry about either of these cases: the transaction should be simply aborted and retried.

In the first case, we cannot choose to commit the transaction, since it was already aborted due to the conflict; we discuss some other variants that allow the programmer to control whether the transaction is retried in Section 5, but we believe that the default behavior should be to retry without involving the programmer.

In the second case, we could consider the asynchronous signal as another way to exit a transaction, but then returning from the signal handler would effectively be jumping into the middle of a transactional statement, which we have strived to avoid in general. Alternatively, we could consider disabling signals for the duration of a transaction, but we believe that this is impractical. A more pragmatic solution is to abort the transaction, handle the signal, and then retry the transaction, as if the signal occurred before the transaction began. While it is possble to avoid the overhead of retrying by handling a signal without aborting a transaction that is executing when it occurs, this would require more sophisticated support from the transactional memory infrastructure than we would like to require for an initial implementation.

3.6 Privatization

Many transactional memory implementations exhibit the so-called "privatization problem" [8, 21]: The implementation allows transactions that have logically completed to modify a piece of memory that another transaction intended to make private—for example by removing the last shared reference to it—and thus safe to access nontransactionally. A particularly troublesome example occurs when the memory is freed and subsequently reallocated. Clearly is not reasonable to forbid nontransactional access to freshly allocated memory.

This is primarily an implementation issue, but has some bearing on the programming interface: some privatization mechanisms that aim to address this problem require the programmer to *explicitly* say that a piece of memory is now believed to be private, while others solve the problem *implicitly*, i.e., without special treatment by the programmer.

The privatization problem was identified only recently. Early indications are that practical solutions for both implicit and explicit privatization exist, but implicit privatization entails a significant cost in performance and scalability, which naturally tempts us to burden the programmer with explicit privatization. We resist this temptation, requiring implicit privatization in an initial implementation, for the following reasons:

- It provides an easier path for programmers to adopt transactional programming styles because it avoids requiring them to understand a new issue that has no analogue in traditional concurrent programming.

- Research on privatization is in its infancy; it is too early to conclude that its overhead is fundamentally excessive.
- Practical solutions exist, and although they do impose a performance and scalability cost on transactions, they do not impose overhead on the rest of the application.

Note that this choice is *not* consistent with our desire to make decisions that we can change without breaking existing code. Thus, if an explicit privatization model is adopted later, it should be chosen explicitly (for example with compiler options), and the system should continue to support implicit privatization for programs that do not make this choice.

A possible intermediate position is to implement only explicit privatization, and require `free` to ensure that a block of memory is privatized before being freed. Programmers would still be required to avoid nontransactional accesses to an object that has been accessed by a transaction and not since freed, but at least nontransactional accesses to freshly allocated memory would be safe.

Note that an implementation that provides strong atomicity by turning nontransactional accesses into mini-transactions does not exhibit the privatization problem: there are no nontransactional accesses. However, many of the optimizations that can reduce the overhead of strong atomicity also reintroduce the privatization problem, so we advise against conflating these issues, even if we eventually decide to support strong atomicity.

3.7 Input/output, system calls, and libraries

Code executed within transactions must generally provide means for detecting conflicts, as well as for undoing the effects of the transaction in case it aborts. This means that calling code that has not been compiled to execute within transactions can result in incorrect behavior, especially if such code has side effects: these side effects may be noticable even if the transaction aborts. An obvious example is code that performs I/O: if visible output has already occurred, it is not possible to abort the transaction and give the impression that it never executed.

For these reasons, we recommend against supporting transactions that perform I/O, invoke system calls, or call library functions that have not been compiled by a transaction-aware compiler.

Some system calls and library functions (e.g., math library functions) have no side effects and can therefore be safely called within transactions, even though they are not aware of transactions. In other cases, by making the implementations aware of transactions, we can make them safe to include in transactions. In other cases, we will likely want to forbid certain system and library calls in transactions for the forseeable future. Similarly, some kinds of I/O can be supported within transactions (for example, a transactional file system appropriately interfaced with the transactional memory implementation can allow file I/O within transactions).

In an initial implementation, we might impose the rule that none of these features can be used within transactions, and leave behavior undefined if this rule is broken. As the implementation becomes more mature, it will be important to provide support to help the programmer avoid breaking these rules. Over time we can provide increasingly robust support for flagging errors (for example, we might implement runtime error messages first and later provide support for compile-time checking of conformance to the rules). We might also relax the rules over time, as more system calls and library functions are certified as "safe to call" or if operating system or library changes are made to make them safe. Observe that each step in such a progression can be made without breaking existing correct code.

While a variety of techniques for allowing I/O within transactions are known for a variety of contexts, it does not seem possible to have a general solution that covers all cases and does not introduce significant detrimental interactions with other features and with performance. Therefore, it is preferable to gain experience with limited models that do not allow I/O in order to learn whether supporting I/O in the long run makes sense, and if so what kinds of I/O for what kinds of applications. Given the tradeoffs that are inevitably introduced by attempting to support I/O in transactions, it does not make sense to impose a choice in this regard without clear guidance.

3.8 Other advanced features

A host of features have been proposed for use in transactions. It is clearly undesirable and probably infeasible to support them all simultaneously, and many of them introduce tradeoffs that we should not make without clear guidance that the feature is needed. Therefore, we recommend against supporting any of these more advanced features initially, so that we may gain experience and make informed decisions about whether and how to support them later. We briefly discuss a few such features below. In all cases, we are comfortable with our decision to omit such features initially, because they can be added later without breaking existing code.

As mentioned earlier, richer language features have been proposed that provide additional flexibility beyond what basic transactional statements allow. For example, the `retry` and `orElse` constructs of Haskell STM [15] allow programmers to express alternative code to execute in case a (nested) transaction aborts, to wait until some condition holds before proceeding, etc. It may make sense to support similar features in C++ in the future, but not initially.

Open nesting allows a nested transaction to commit or abort independently from its parent. A good discussion of why open-nested transactions might be useful can be found in [4]. Open nesting significantly complicates the semantics of transactions, however, because open-nested transactions violate the all-or-nothing guarantee: An aborted transaction may have effects that are visible to other threads because it may include open-nested transactions that commit. Sim-

ilarly, a committed transaction may not appear to execute atomically because it may include open-nested transactions that commit, and make their effects visible to other threads, before the parent transaction commits (or aborts). Designing open nesting semantics that enables the desired functionality without giving up many of the benefits of transactions is an active area of research, and there is not yet agreement about what the right trade-offs are.

We also recommend against allowing parallelism within a transaction in an initial implementation; each transaction should be executed by a single thread. There are three reasons for this recommendation. First, although many C++ programs are concurrent, concurrency is not native to the language, and creating a new thread is a relatively heavyweight operation. C++ programs tend not to create threads liberally, but instead create enough threads to provide the desired parallelism and then coordinate their execution with locks and other synchronization mechanisms. Transactional memory in C++ serves that same purpose. Second, research on transactional memory thus far has been primarily for single-threaded transactions. Well developed proposals for efficient transactional memory implementations that support parallelism within transactions do not yet exist, and the cost of supporting such seems likely to be high, at least in the near term. Third, similar to open nesting, parallelism within transactions introduces many semantic issues, particularly with respect to the nesting model. We believe that eventually it may be important to support parallelism within transactions, but not yet, at least not for C++.

4. Exceptions

Exceptions present a challenge in the integration of transactional memory into C++. On one hand, exceptions are intended to handle errors that cannot be handled locally when they are discovered. The design principle in C++ is that exceptions should be rare, that they indicate failure, and that the purpose of exception handlers is to restore invariants; this principle is embodied in such notions as *exception safety* [44]. This is similar to the original purpose of transactions: aborting a transaction discards its effects, restoring any invariants that hold outside of transactions; manual restoration of invariants seems redundant. Thus, predictability suggests that transactions terminated by exceptions should abort.

On the other hand, an exception in C++ may throw arbitrary objects, which carry information from the point of the throw to the exception handler. In particular, these objects may contain pointers or references to objects allocated or modified in a transaction to a point outside the transaction. If a transaction aborts when it is terminated by an exception, then the information in the exception may be lost. Indeed, if all the effects of a transaction are discarded, then referenced objects that were constructed within the transaction would no longer exist, and the references would be dangling, violating the expectations of C++ programmers. This problem is avoided if transactions terminated by exceptions commit.

Thus, aborting transactions is a natural fit to the *use* of exceptions in C++, but causes problems for the exception *mechanism*. We find the disadvantages induced by either committing or aborting transactions terminated by exceptions to be significant enough that we believe more experience is needed before settling on a solution. And if we choose to abort such transactions, we must still decide which effects are discarded.

Because of the challenges involved in determining how exceptions should interact with transactions and (in some cases) of implementing the desired behavior, it may be desirable to simply disallow exceptions within transactions for an initial implementation. For many applications, this restriction is of little consequence, especially if violations of this rule can be detected and reported, preferably at compile time. Thus, we should not wait to make a final determination on this issue before releasing an initial implementation and encouraging experimentation with it. In the rest of this section, we discuss some alternatives for integrating exception support in the future.

4.1 Exceptions Commit

This option is the simplest: Committing transactions when an exception leaves the `transaction` statement leaves the exception mechanism unchanged in both semantics and implementation. Ringenburg and Grossman [39] favor for this approach for their AtomCaml language. However, different approaches are appropriate for different contexts. Committing a transaction that throws exception seems counter to the intended use of exceptions in C++ as a mechanism to deal with failure and restore invariants. This intention is not merely a de facto standard; it is an explicit design rule of C++ [23, 44]. Since an exception signals failure of some assumption, it may occur when the transaction has made the state inconsistent, relying on the guarantee that no other threads will be able to see that inconsistency.

If the programmer must explicitly restore any invariants that may have been violated when an exception was raised, additional code will be required to do the restoration, and perhaps also to record the effects of a transaction in *normal* execution in order to facilitate the restoration. This seems to be a poor choice when the transactional memory implementation can perform the task automatically and transparently. Worse, other transactions will be able to observe the intermediate state between the commit of the transaction that throws the exception and the commit of the transaction that restores the invariant. This severely undermines the power of transactions for simplifying programs by enforcing invariants.

4.2 Exceptions Abort on Exit

If a transaction terminated by an exception is to be aborted, the most natural approach is to abort it when the exception leaves the `transaction` statement. However, it still leaves

the thorny issue of what to do about the object thrown by the exception in the default case when the transaction is aborted.

The object thrown is not itself a problem: the exception mechanism already requires distinct memory allocation, so we can simply make these writes not part of the transaction; they occur immediately after the transaction aborts, and just before the object is thrown. In this sense, this solution is the one described above for `longjmp`, where aborting did not cause any problems because only an integer is passed from the point of the `longjmp` to the point of the `setjmp`. The problem occurs if the exception object contains pointers or references to other objects, particularly ones allocated or modified in the transaction.

The simplest possibility is to forbid throwing objects with pointers or references. This is not as restrictive as it might seem at first: although C++ exceptions may throw arbitrary objects, in practice, most exception objects are chosen from a conventional class hierarchy. The most likely case of an exception object with pointers is a `std::string`, as in:

```
std::string x( "hello" );
throw x;
```

The normal course of action is to copy construct `x` into the exception memory. This copy construction may involve a call to `new`, because strings are generally indirect objects. Worse, the string may be indirected to a reference-counted object: copying the object involves incrementing the reference count, while aborting the transaction involves rolling it back, resulting in an incorrectly built object in the exception memory.

Note that the compiler cannot always detect violations of a rule forbidding throwing objects containing pointers (because exceptions are a dynamic construct). Therefore, if we chose this rule, we would make no guarantees about pointers or references in exception objects thrown out of a transaction. A disadvantage of this approach relates to our desire for incremental adoption and predictability: it would be easy to call some legacy code that potentially throws a complex object; the error would not be caught by the compiler and the behavior would be unpredictable.

Another possibility is to copy the objects referenced by the exception object. If we do only a shallow copy, then all the problems remain behind one level of indirection, though solving the problem for one level of indirection may be sufficient to address the vast majority of uses in practice. On the other hand, a deep copy may be prohibitively expensive, especially if the copied objects must reside in the exception memory. Thus, we do not find this possibility very attractive.

A third possibility is to allow pointers and references to other objects, and to discard all the effects on objects that were not allocated within the transaction. However, objects allocated within the transaction would still remain, and so there would be no dangling references. This is essentially the approach proposed for STM Haskell [15], and we believe it worth considering. One possible shortcoming is that examining the exception object may be confusing because it may not be self-consistent: objects allocated within the transaction will reflect the effects of the transaction, while objects not allocated within the transaction will not.

4.3 Exceptions Abort on Throw

Another approach is to exploit the idea that exception handlers and transaction aborts are both "fixing mistakes", and rely exclusively on the transaction mechanism. Specifically, abort the transaction at the point of throw when the handler is determined not to be within the same transaction; no exception is thrown at all. Automatically retrying the transaction in this scenario is probably inadvisable, so we must specify where control resumes after the transaction aborts. We can do this by adding a clause to the `transaction` statement:

`transaction` *statement* [`exception` *statement*]

This approach will lead to smaller transaction bulk and better performance. However, it makes it difficult to determine cause: the only information one can get from a throw-aborted transaction is that it aborted via throw. This approach may be fine as long as one follows the "exceptions are indications of program failure" design rule of the C++ standard, but again, experience is necessary to assess this.

A slightly more powerful variant is to allow passing *some* information, whose type is restricted by the syntax, to the `exception` clause. For example, we might allow the programmer to bind one integer variable, which is accessible inside the `transaction` statement and the `exception` clause, and will not be rolled back if the transaction aborts. The programmer can use this variable to pass simple information out of an aborted transaction to the `exception` clause, while the exception mechanism can avoid the complication of dealing with arbitrary thrown objects.

4.4 Overriding defaults

One consideration in evaluating the tradeoffs discussed above is how easy it is to override the default behavior. For example, if the default behavior is for exceptions to abort on exit, programmers can easily cause the transaction to commit when appropriate, as in:

```
{
  SomeType throwthis = NULL;
  transaction {
    try {
      // code that might throw an exception
    } catch (SomeType e) {
      throwthis = e;
    }
  }
  if (throwthis)
    throw throwthis;
}
```

This idiom provides a reasonable emulation of the alternative semantics, but is not seamless. For example, catching the exception inside the transaction and throwing it outside would lose useful information such as stack traces, which would interfere with debugging.

Similarly, given explicit abort (see Section 5.2), we can catch an exception and explicitly abort the transaction. Thus the choice between these options is mostly concerned with common-case performance and the semantics most likely to be appropriate. In contrast, if *throwing* an exception aborts the transaction, we do not have such flexibility.

5. Variants and Idioms

Beyond the basic design for integrating transactional memory into C++, there are additional features that may be useful to provide, and some idioms that provide useful functionality. We discuss some of these in this section.

5.1 Programmer-directed contention control

In the basic design discussed thus far, the system is responsible for arbitrating conflicts between transactions. If a transaction aborts due to conflict with another transaction (or an asynchronous signal), it is simply retried. To avoid livelock, the system must have some mechanism to facilitate progress, for example by managing contention so that each transaction eventually executes without conflict. An alternative (or additional) possibility is to provide the programmer a hook for detecting and dealing with transactions that abort for some reason. A simple such hook is a clause for the `transaction` statement, such as:

`transaction` *statement* [`failure` *statement*]

The programmer can use the `failure` clause to implement some simple functionality that restricts the number of retries, or tries an alternative transaction upon failure, etc.

Note, however, that the existence of such a hook need not imply that the system will always execute the statement in the `failure` clause when the transaction is aborted: Because the effects of the aborted transaction are discarded, the system may retry the transaction before ceding control to the programmer. Similarly, we should not be tempted to overspecify the circumstances under which the `failure` clause may be invoked: different implementations have different reasons for failing transactions. The main point is to give control to the programmer eventually in case the transaction (repeatedly) fails to complete.

The `failure` clause should not be conflated with the `exception` clause from the "abort on throw" option for exceptions; the desired behavior in these cases are likely to be quite different.

5.2 Explicit abort

It can be useful to provide a way for the programmer to explicitly abort a transaction. For example, such a mechanism enables a convenient style for programming a task that should be done only if a certain condition is met, and determining that condition involves doing much of the task. Rather than testing the condition and then redoing much of the work if the task needs to be done, the programmer can simply start doing the work in a transaction and then abort the transaction if the condition is not met.

If exceptions abort transactions, they could be used for this purpose. However, regardless of how we eventually decide to handle exceptions, it may still be better to provide a separate mechanism for explicitly aborting transactions, especially if exceptions abort on throw: the behavior desired when a transaction is explicitly aborted is likely to be quite different from that desired when a truly unexpected event occurs. For exceptions that abort on exit, the need for a separate mechanism is less clear because we can easily distinguish the cases by using a special kind of exception for an explicit abort. However, if explicit abort is intended as an alternate means of ordinary control flow (like `break`), then by the C++ design principle that exceptions indicate program failure, an exception should not be used for this purpose.

Therefore, we believe that we will eventually want to support explicit abort. The behavior of explicit abort may depend on the nesting model implemented. For example, with flat nesting, the obvious behavior is to undo the effects of the entire transaction, and resume execution after the aborted transactional statement. With closed nesting, it may be desirable to abort only the current (nested) transaction and resume execution of the parent transaction. To avoid breaking existing code when introducing closed nesting, and to allow the programmer to express both kinds of aborts, the programmer should express "abort to the top" and "abort this (nested) transaction" differently, for example with `abortAll` and `abortInner` keywords.

We note that it is not necessary to explicitly support an indication of whether a transaction was aborted, because this can easily be achieved using an idiom like the following one:

```
{
  int aborted = true;
  transaction {
    aborted = false;
    ...
  }
  if (aborted) {
    ...
  }
}
```

5.3 Passing an argument to a `transaction` statement

We also considered allowing an "argument" to be passed to a `transaction` statement, such as

`transaction (x)` *statement*

Effects on this variable would *not* be discarded if the transaction aborts, so it can be used to implement much of the

functionality described above. This feature would be especially helpful in a system intended primarily for exploration of various options, and in that case, we may want the argument to be a `void` pointer, so that we can easily adapt the system. However, such an open-ended mechanism is probably not appropriate as the final point in a language design.

6. Conclusion

We have summarized the main issues that must be addressed for a pragmatic initial integration of transactional memory into C and C++, aimed at supporting incremental development and adoption. Our goal has been to explore the design choices for integrating transactions into these languages in a way that will be natural for programmers who use the prevalent coding styles in C and C++.

To support transactions in C and for C-like data types in C++, a word-based transactional memory implementation is most appropriate, and this naturally leads to a `transaction` statement as the primary syntax for expressing transactions. For C, the trickiest issue is how to handle I/O in transactions, if at all, while C++ provides additional challenges, particularly related to exceptions.

This paper narrows the design space on a number of issues, but leaves open reasonable alternatives for others. Future work includes implementing and experimenting with some of these alternatives, allowing use and experience to guide us towards successful integration of transactional memory into the C and C++ standards, as well as successful use of these features by programmers.

Acknowledgments

We are grateful to Stephen Clamage, Peter Damron, Maurice Herlihy, Chris Quenelle, and Douglas Walls for helpful discussions.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, 2006.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.

[3] C. Blundell, E. Lewis, and M. Martin. Deconstructing transactional semantics:the subtleties of atomicity. In *Proc. Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[5] C. Cole and M. Herlihy. Snapshots and software transactional memory. *Science of Computer Programming*, 58(3):310–324, 2005.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, Oct. 2006.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. 20th Intl. Symp. on Distributed Computing*, September 2006.

[8] D. Dice and N. Shavit. What really makes transactions faster? In *TRANSACT Workshop*, June 2006. http://research.sun.com/scalable/pubs/TRANSACT2006-TL.pdf.

[9] K. Fraser. *Practical Lock-Freedom*. PhD thesis, Cambridge University Technical Report UCAM-CL-TR-579, Cambridge, England, Feb. 2004.

[10] K. Fraser and T. Harris. Concurrent programming without locks. Submitted for publication.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[12] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proc. 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 258–264, 2005.

[13] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. 31st Annual International Symposium on Computer Architecture*, 2004.

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

[15] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proc. 10th ACM SIGPLAN Symposisum on Principles and Practice of Parallel Programming*, 2005.

[16] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, May 2003.

[17] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proc. Conference on Object-Oriented Programming, Systems, Languages and Applications*, Oct. 2006.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for supporting dynamic-sized data structures. In *Proc. 22th Annual ACM Symposium on*

Principles of Distributed Computing, pages 92–101, 2003.

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

[21] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: A scalable transactional memory allocator. In *ISMM '06: Proc. 2006 International Symposium on Memory Management*, pages 74–83, 2006.

[22] International Organization for Standardization. *ISO/IEC 9899:1999 Programming Languages – C*. 1999.

[23] International Organization for Standardization. *ISO/IEC 14882:2003(E) Programming Languages – C++, Second Edition*. 2003.

[24] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, 1994.

[25] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2006.

[26] Y. Lev and J. Maessen. Towards a safer interaction with transactional memory by tracking object visibility. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, Oct. 2005.

[27] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, May 2004.

[28] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. In *Proc. of the 19th International Conference on Distributed Computing*, Sept. 2005.

[29] V. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proc. 19th Intl. Symp. on Distributed Computing*, September 2005.

[30] V. J. Marathe, W. N. S. III, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Oct. 2004.

[31] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006. Earlier, extended version available as TR 893, Computer Science Department, University of Rochester, Mar. 2006.

[32] J. F. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proc. 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, 2002.

[33] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.

[34] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[35] K. E. Moore, M. D. Hill, and D. A. Wood. Thread-level transactional memory. Technical Report CS-TR-2005-1524, Dept. of Computer Sciences, University of Wisconsin, Mar. 2005.

[36] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. 34th International Symposium on Microarchitecture*, pages 294–305, Dec. 2001.

[37] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 5–17. ACM Press, 2002.

[38] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005.

[39] M. F. Ringenburg and D. Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, New York, NY, USA, 2005. ACM Press.

[40] B. Saha, A.-R. Adl-Tabatabai, R. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programs (PPoPP)*, 2006.

[41] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[42] N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995.

[43] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10):99–116, 1997.

[44] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[45] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *Proc. 18th European Conference on Object-Oriented Programming*, pages 519–542, 2004.