Design and Implementation of Transactional Constructs for C/C++

Yang Ni Adam Welc Ali-Reza Adl-Tabatabai Moshe Bach Sion Berkowits
James Cownie Robert Geva Sergey Kozhukow Ravi Narayanaswamy Jeffrey Olivier
Serguei Preis Bratin Saha Ady Tal Xinmin Tian

Intel Corporation

{yang.ni,adam.welc,ali-reza.adl-tabatabai,moshe.bach,
sion.bar-kovetz,james.h.cownie,robert.geva,sergey.s.kozhukow,
ravi.narayanaswamy,jeffrey.v.olivier,serguei.v.preis,bratin.saha,ady.tal,xinmin.tian}@intel.com

Abstract

This paper presents a software transactional memory system that introduces first-class C++ language constructs for transactional programming. We describe new C++ language extensions, a production-quality optimizing C++ compiler that translates and optimizes these extensions, and a high-performance STM runtime library. The transactional language constructs support C++ language features including classes, inheritance, virtual functions, exception handling, and templates. The compiler automatically instruments the program for transactional execution and optimizes TM overheads. The runtime library implements multiple execution modes and implements a novel STM algorithm that supports both optimistic and pessimistic concurrency control. The runtime switches a transaction's execution mode dynamically to improve performance and to handle calls to precompiled functions and I/O libraries. We present experimental results on 8 cores (two quad-core CPUs) running a set of 20 non-trivial parallel programs. Our measurements show that our system scales well as the numbers of cores increases and that our compiler and runtime optimizations improve scalability.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—Concurrent programming structures

General Terms Design, Languages, Performance

Keywords Transactional memory, C/C++

1. Introduction

Transactional Memory (TM) has received significant attention recently as a simpler concurrency control mechanism compared to locks. Locks have several pitfalls: Coarse-grained locking doesn't scale to a large numbers of cores, while fine-grained locking risks introducing bugs and complicating composition of software modules. By providing automatic fine-grained concurrency control, TM avoids the problems associated with locks and allows safe and scalable composition of software modules.

Recent work has extended various languages with new block constructs for expressing transactions. Much of this work has focused on managed languages such as Java [20], C# [22], Haskell [21], and Caml [39], and on C [46, 3, 35, 15, 28]. Only quite recently have TM language proposals for C++ started emerging [10]. Needless to say, C and C++ are important languages because of their prevalence in systems code and performance-critical applications.

Some prior work has supported transactional programming in C and C++ by providing an API rather than language constructs [13, 31, 11, 16]. API approaches have allowed rapid prototyping of STM algorithms and analysis of their performance. But such APIs do not deliver completely on the main goal of TM, which is to simplify concurrent programming. Manually adding calls to TM API functions imposes a significant burden on the programmer and is error prone.

In this paper, we present a complete software transactional memory (STM) system that adds first-class language constructs for transactional programming to C++. The system consists of new C++ language extensions, a compiler that translates and optimizes these extensions, and a high-performance STM runtime library. Compared to prior work on C, the new language constructs support C++ language features such as classes, inheritance, virtual functions, exception handling, and templates. We also provide constructs that allow the programmer to create efficient transactional versions of existing libraries. This paper makes the following contributions:

- We introduce new language extensions to support transactional memory in C++. Unlike prior work that focuses on C, our language extensions support C++ classes, virtual functions, inheritance, templates, and exception handling. The language constructs allow the programmer to call existing precompiled code inside transactions, including code that may perform unrestricted I/O. The system also provides constructs for the expert library developer to develop versions of their libraries optimized for execution inside transactions. (Section 2)
- We extend an existing, high-performance production C/C++ compiler to support our new transactional language constructs. We describe novel code generation techniques and compiler optimizations for these new language constructs. (Section 3)
- We present a novel STM runtime library that implements both optimistic and pessimistic concurrency control. The runtime also implements a serial execution mode to support calls to legacy binaries and to support unrestricted I/O operations inside transactions. The runtime can switch between these execution modes dynamically to optimize performance. We present a novel TM library API that supports this flexible concurrency control model while allowing compiler optimizations. (Section 4)
- We present a thorough experimental evaluation of our STM system on a large set of parallel programs ported to use our language extensions. Our measurements demonstrate that our system scales well across these programs and that our optimizations are important for performance. Our measurements also show that optimistic concurrency control performs better than pessimistic concurrency control though our pessimistic concurrency control algorithm performs competitively in many cases. (Section 5)

We have released an earlier version of our system, including the compiler and runtime described in this paper at the Intel WhatIf web site[1]. Early adopters can download these tools and experiment with transactional programming in C++.

2. Transactional C/C++

This section describes our language extensions to support transactional memory in C++. Prior systems support basic mechanisms for C [46, 3]. We refine and extend these mechanisms to provide first-class language constructs (rather than compiler pragmas) and to add full support for C++ classes, virtual functions, inheritance, templates, and exception handling.

[1] The URL for the release of our system is http://whatif.intel.com.

2.1 Atomic blocks

The `__tm_atomic` statement defines a basic *atomic block*, similarly to the `atomic` construct previously defined in the literature [2, 22] and introduced as a C language pragma in [46]:

```
__tm_atomic {
  // block of arbitrary C/C++ statements
}
```

The TM system executes atomic blocks as transactions and isolates concurrently executing transactions from each other with the net effect that all the operations in one transaction appear to complete either before or after all the operations in any other transaction.

Within each atomic block, the compiler instruments each shared-memory access so that its execution is delegated to the TM runtime. The TM runtime tracks all transactional accesses and detects conflicting accesses among concurrently executing transactions. Two transactions conflict if they both access the same memory location at the same time and at least one of them writes to that location. On a potential conflict, the TM runtime transparently rolls back the side effects of one of the conflicting transactions and re-executes it until it succeeds without conflicts.

Atomic blocks can contain arbitrary code of all regular C/C++ statements, including direct and indirect function calls, and virtual function calls. This includes calls to precompiled libraries (i.e., code that has not been compiled by the transactional compiler) and those that perform arbitrary I/O operations. Since the TM runtime cannot track the accesses inside precompiled code or roll back I/O operations, calling into such code inside a transaction causes the runtime to serialize execution of the transaction with respect to other transactions – no other transactions are allowed to be in-flight concurrently with the serial one. Section 4 describes the details of this serial execution mode.

2.2 Abort statements

The `__tm_abort` statement (a *user abort*) allows the programmer to roll back an atomic block explicitly. This statement must appear in the lexical scope of an atomic block. It rolls back all side effects of the atomic block that statically encloses it and transfers control to the statement immediately following the block. It ends the transaction if the enclosing atomic block is the outermost atomic block.

Because the runtime cannot log the side effects of precompiled functions, the `__tm_abort` statement can execute only if the innermost atomic block containing it has not called a precompiled function (unless that function is a `tm_pure` function as described in Section 2.4.2). A runtime error will occur if an atomic block executes a user abort after it has called a precompiled function:

```
__tm_atomic {
  print(‘‘HelloWorld!’’);
```

```
    __tm_abort; // error (runtime failure)
  }
```

This does not preclude calling a precompiled function in an atomic block that is at an outer dynamic nesting level relative to the block containing the abort statement:

```
  __tm_atomic {
    print(``HelloWorld!'');
    __tm_atomic {
      __tm_abort; // OK
    }
  }
```

Nested atomic blocks have closed nesting semantics [37], which means that the side effects of a nested transaction commit become visible only when the dynamically outermost transaction commits. Abort statements allow a programmer to roll back a transaction partially by aborting the innermost nested atomic block. The compiler and runtime, therefore, may not flatten those nested atomic blocks that contain user abort statements.

2.3 Single lock semantics

Atomic blocks provide single lock atomicity (SLA) semantics [34]: A program behaves as if a single global lock guards each atomic block. This guarantees that programs that are race free under a single global lock will execute correctly under transactional execution. These semantics support the privatization and race free publication patterns [34]. A trivial implementation of atomic blocks can use a single global lock to implement isolation (though such an implementation must still perform undo logging to support abort statements). In fact the serial execution mode (which supports calling precompiled binaries) falls back to using a single global lock to implement atomic blocks.

Consistent with the emerging C/C++ memory model specification [8], SLA semantics provides no guarantees for programs containing data races. In the presence of data races between transactions and non-transactional code, code executing outside a transaction may see speculative or intermediate values produced by a transaction, and it may violate the isolation of a transaction by writing to memory locations accessed inside a transaction.

To support SLA semantics correctly, the STM implementation must guarantee several important safety properties, namely privatization safety, granular safety, and observable consistency[34]. These safety properties ensure that code that is race free under a single global lock remains race free under transactional execution; that is, they ensure that the STM implementation does not introduce a data race into a program that is otherwise race free under a single lock. Most prior STM systems did not properly maintain these properties and thus cannot be used to implement single global lock semantics for C/C++ atomic blocks.

2.4 Function annotations

To optimize code generation and allow the user to optimize calls to functions that do not require instrumentation, our system introduces function annotations (similar to [46]). Any given function can be called from both inside and outside of an atomic block. To support both transactional and non-transactional execution of functions efficiently, the compiler generates two versions of each function, one version with instrumentation for transactional execution and one version without. While just-in-time compilers can decide whether and when to duplicate code at run time [2], static C/C++ compilers must make this decision at compile time. To avoid unnecessary code duplication, we introduce an annotation to denote functions (including class member functions) that can be called from inside transactions. All annotations are specified using the `__declspec` keyword on Windows and the `__attribute__` keyword on Linux. The examples that follow use the Windows notation.

2.4.1 `tm_callable`

A `tm_callable` function is one that the programmer intends to call from inside a transaction and would like compiled for efficient transactional execution. The compiler generates two versions of the code for such functions, one with instrumentation and one without. The compiler uses a mangled name for the transactional clone of a `tm_callable` function. The name mangling simply adds a transactional suffix to the function name and was designed to work correctly with the template name mangling for `tm_callable` function templates. If a `tm_callable` function calls an unannotated function, the compiler generates code that triggers serial execution unless it knows that the called function does not require instrumentation or it has automatically generated an instrumented version of the called function.

2.4.2 `tm_pure`

A `tm_pure` function is one that the programmer asserts can execute safely inside a transaction without requiring transactional instrumentation and without switching to serial execution. The programmer takes full responsibility for the behavior of `tm_pure` functions. The main intent of `tm_pure` is to provide the programmer a way to optimize calls to precompiled library functions (such as math functions) that are known to be pure. The programmer can safely annotate a function as `tm_pure` if it does not access any static or non-local memory or if it is pure from the perspective of higher-level program logic (e.g., it accesses only immutable global variables). The compiler can not validate the purity of all functions; however, to help flag potential errors, it will issue a warning if it compiles the definition of a `tm_pure` function and detects that it would have added transactional instrumentation to that function.

2.4.3 `tm_unknown`

The `tm_unknown` attribute annotates a function whose TM properties are unknown (e.g., it is unclear to the programmer whether the function is going to be called from inside a transaction, or the function is in a library and cannot be recompiled by the TM compiler.) An unannotated function is implicitly a `tm_unknown` function. The compiler may decide to create an instrumented version of a `tm_unknown` function. This annotation allows the programmer to override class-level annotations described in Section 2.5.

2.5 Class annotations

The `tm_callable` annotation is allowed on class declarations including C++ template classes. All member functions of a `tm_callable` class, both virtual and non-virtual, are implicitly `tm_callable`. This is equivalent to annotating each member function of the class as `tm_callable`. This eases C++ programming as it allows the programmer to annotate once at the class level rather than annotating each member function. In the following class declaration, for example, both `foo()` and `bar()` are implicitly `tm_callable`:

```
__declspec(tm_callable) class C {
  void foo();
  void bar();
};
```

Derived classes inherit the class-level `tm_callable` annotation. In case of multiple inheritance, a derived class inherits the `tm_callable` annotation if at least one of its base classes is annotated with `tm_callable`.

Function-level annotations override the class-level annotation. In the following class declaration, for example, function `foo()` gets the class-level `tm_callable` annotation while function `bar()`'s class-level annotation gets overridden with `tm_unknown`:

```
__declspec(tm_callable) class C {
  void foo();
  __declspec(tm_unknown) void bar();
};
```

2.6 Virtual function overriding and inheritance

Virtual function overriding and inheritance introduce additional subtleties. A virtual function can legally override another virtual function if and only if the two have compatible TM annotations. Table 1 shows the compatibility rules. In general, function overriding is legal if and only if the virtual function in the derived class has the same annotation as in the base class, or the function in the base class is `tm_unknown`

In the case of multiple inheritance, a function in the derived class may override virtual functions in the base classes only if the rules specified in Table 1 hold separately for every pair of functions in the base and derived class. Consider a more complicated example that combines these rules:

	derived class		
base class	`tm_callable`	`tm_pure`	`tm_unknown`
`tm_callable`	yes	no	no
`tm_pure`	no	yes	no
`tm_unknown`	yes	yes	yes

Table 1. Compatibility rules for virtual function annotations

```
__declspec(tm_callable) class A {
  __declspec(tm_unknown) virtual void foo();
};
class B {
  __declspec(tm_callable) virtual void foo();
};

class C: A, B {
  void foo();
};
class D: A, B {
  __declspec(tm_pure) void foo(); // Error!
};
```

In this example, class `C` inherits the `tm_callable` attribute from class `A`, so `C::foo()` is implicitly `tm_callable`. According to the rules in Table 1, a `tm_callable` function (`C::foo()`) may override both `tm_unknown` (`A::foo()`) and `tm_callable` (`B::foo()`). At the same time, it is an error to annotate `D:foo()` with `tm_pure`, because it is incompatible with `tm_callable` (`B::foo()`).

2.7 Templates

The function annotations can be used on template functions, and the the `tm_callable` annotation can be used on template classes. The following example shows how a function annotation can be used with function templates:

```
template <class T>
__declspec(tm_callable) T max(T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}
```

2.8 Exception handling

Uncaught exceptions that propagate out of an atomic block cause the atomic block to commit its side effects. The alternative strategy provides failure atomicity by rolling back the atomic block's side effects when an exception propagates out of the atomic block. Our justifications for committing the transaction are the following: (1) Committing is consistent with a single lock semantics model as lock-based critical sections do not roll back side effects on an uncaught exception; (2) it is impossible to roll back the state of an atomic block if it has executed any `tm_unknown` functions in serial mode without instrumentation; (3) rolling back the transac-

```
void  addCommitAction(void (*)(void*),void*)
void  addUndoAction(void (*)(void*),void*)
```

Figure 1. API for adding user actions for `tm_wrap` functions

tion could result in an inconsistent state for the thrown exception object (whose state may also be rolled back); and (4) we believe that it is better to provide a separate explicit mechanism for roll back than to overload exceptions with roll back – the programmer can always catch exceptions and explicitly roll back side effects using the `__tm_abort` statement.

2.9 Support for writing transactional libraries

The system provides an additional annotation along with an API that together allow the programmer to create optimized transactional versions of libraries. Section 4.5 describes how we use these features to create a transactional version of the memory management library. The system also provides an additional statement, `__tm_waiver`, supporting escape actions that allow programmers to reduce transactional instrumentation overhead whenever it is considered safe to do so. These features are intended for experts (e.g., library developers) as programmers who use these features must understand some of the details of how TM systems are implemented.

2.9.1 Function wrappers

The `tm_wrap` annotation declares a *transactional wrapper* function. Calls inside transactions to a "wrapped" function are redirected to the user-specified wrapper function that escapes the transaction. Transactions that execute in serial mode may or may not execute wrapper functions as such transactions might execute uninstrumented code. The following example shows how to declare a transactional wrapper `fooTxn()` for some function `foo()`:

```
__declspec(tm_wrap(foo)) void fooTxn();
```

After seeing this declaration, the compiler translates every in-transaction call to `foo()` into a call to `fooTxn()`, which executes without TM instrumentation.

Like `tm_pure` functions, the wrapper function is executed without transactional instrumentation so the programmer takes responsibility for their correct behavior. Inside a wrapper function, the programmer must be careful not to access memory that has been accessed transactionally as such an access may see an inconsistent or speculative value; that is, the programmer must segregate the data accessed inside the wrapper function from data that is accessed transactionally by any thread including the thread making the call. The programmer must also not execute any atomic blocks inside the wrapper function.

2.9.2 Undo and commit actions

Inside wrapper functions, the programmer must register the proper undo and commit actions to roll back or finalize the effects of the wrapper function on an abort or commit, respectively. A commit action executes when the transaction commits. An undo action executes when the transaction rolls back due to a user abort or a conflict.

The TM library exports an API (Figure 1) that the programmer can use to register commit and undo actions inside of wrapper functions. `addCommitAction` adds an entry to the commit action log. `addUndoAction` adds an entry to the undo action log. In both functions, the first parameter is a pointer to the function that implements the action, and the second parameter is an argument that is passed to the action when it executes.

Serial mode transactions containing no `__tm_abort` statements do not roll back because they are guaranteed to commit. The undo actions for such transactions are ignored and their commit actions may execute immediately without actually being added to the commit action log.

2.9.3 Escape actions

Certain data accesses inside of transactions, such as accesses to private or read-only data, do not need to be instrumented and yet the compiler may not always be able to detect such accesses automatically. The `__tm_waiver` statement allows the programmers to convey such application-level knowledge to the compiler and avoid unnecessary instrumentation-related overhead. The `__tm_waiver` statement defines a block of code that will not be instrumented by the compiler and can be used by programmers to optimize a transactional application. Code regions defined by the `__tm_waiver` statement in effect bypass the transactional concurrency control mechanisms and as such should be used with caution.

3. TM compiler

This section describes the compiler support required for translating and optimizing the transactional language constructs. The compiler translates the transactional language constructs into code instrumented with calls to an STM runtime. The instrumentation is amenable to many classical optimizations, such as redundancy elimination, dead code elimination, and memory optimizations. The interface between the compiler and runtime is designed to support compiler optimizations and at the same time to support multiple STM algorithms.

3.1 Compiler-runtime ABI

Prior work [46] tightly coupled the compiler and generated code to the STM algorithms to maximize compiler optimization opportunities. The compiler in [46], for example, inlined the STM fast paths into the program binary and exposed the operations that constitute the underlying STM algorithm to compiler optimizations. This binds both the generated code and the compiler to one particular STM algorithm and precludes changing the STM runtime or its algorithms without changing the compiler and the generated code.

```
TxnDesc*   getTransaction()
int        begin(TxnDesc*,int)
void       commit(TxnDesc*)
int        beginInner(TxnDesc*,int)
void       commitInner(TxnDesc*)
void       userAbort(TxnDesc*)
void       switchToSerialMode(TxnDesc*)
void       write<Type>(TxnDesc*,Type*,Type)
Type       read<Type>(TxnDesc*,Type*)
void       memcpy(TxnDesc*,void*,void*,size_t)
void       logValue<Type>(TxnDesc*,Type*)
void       logBulk(TxnDesc*,void*,size_t)
void       writeAW<Type>(TxnDesc*,Type*,Type)
Type       readAR<Type>(TxnDesc*,Type*)
Type       readAW<Type>(TxnDesc*,Type*)
Type       readFW<Type>(TxnDesc*,Type*)
```

Figure 2. Compiler-runtime ABI

In contrast, the compiler-runtime ABI in this work decouples the compiler and generated code from the runtime. This approach sacrifices some compiler optimization opportunities but significantly increases the flexibility of the runtime: It allows the runtime to switch STM algorithms dynamically, and it allows the runtime library developer to replace the library in the field if necessary.

We believe that this is the right trade off to make. At high thread counts, the STM algorithms will likely influence end-to-end application performance more than compiler optimizations that are designed to reduce single-thread overheads of the STM. For low threads counts, a single global lock mode executing uninstrumented code may yield even better performance than an STM. Moreover, research on STM algorithms will likely continue into the future, and it is premature to commit to a single STM algorithm as the best; therefore, it is best to keep the generated code flexible so that new STM algorithms can be linked dynamically.

Although we limit our discussion to STM in this paper, support for hardware acceleration also requires a flexible runtime supporting dynamic switching between TM algorithms. Prior work [41, 43, 12] has developed various algorithms for accelerating TM performance using different hardware acceleration techniques. Our system can dynamically switch between different algorithms to take advantage of hardware acceleration.

Figure 2 shows the ABI between the compiler and runtime. Each transaction has a descriptor structure (`TxnDesc`) that holds the transaction meta-data. The descriptor is kept in thread local storage (TLS). The ABI functions all take an explicit argument for the descriptor structure to avoid redundant TLS accesses.

The `begin` and `commit` functions start and end a transaction, respectively. The `begin` function acts like a `setjmp` in that it may return multiple times. On the initial begin call, the return code directs the generated code to take the instrumented or uninstrumented code path. The runtime can direct execution to the uninstrumented code if it decides to start a transaction in serial mode. On a conflict or `__tm_abort`, the runtime executes a longjmp back to the begin function and returns a code that directs the generated code to re-execute or abort the transaction. The second argument of the `begin` function passes flags communicating information about the generated code to the runtime. This information includes whether the compiler has generated instrumented or uninstrumented code, whether the atomic block has an abort statement, and whether it will (or might) call precompiled code (i.e., a `tm_unknown` function). The runtime uses this information to select the most appropriate execution mode.

When generating instrumented transactional code, the compiler knows if any atomic block it encounters is nested. The compiler flattens nested atomic blocks unless they contain an abort statement, in which case it generates calls to the `beginInner` and `commitInner` functions. Abort statements are translated to the `userAbort` function.

The `switchToSerialMode` function switches the transaction to the serial execution mode. As described later, the compiler inserts a call to this function before the first call to a `tm_unknown` function. If the compiler detects that all paths through the atomic block call a `tm_unknown` function then it will communicate this fact to the runtime in the `begin` function; the runtime may then start the transaction in serial mode.

The compiler translates each transactional memory access into a call to the `read` and `write` functions (also known as *barriers*). There is a read and write function for each primitive data type. These frequently-called functions use register parameter calling conventions to reduce their overhead. As an optimization, the interface provides a function implementing memory copying inside of transactions (`memcpy`).

Writes to local variables and thread-local variables must be logged in case of an abort but don't need to be tracked for conflict detection as other transaction can't access them. The compiler explicitly saves and restores live scalar locals that are modified by the atomic block on transaction begin and abort, respectively. No logging is necessary for such variables when they are written inside the atomic block. For a transactional write to a non-scalar local or a thread-local variable, the compiler generates a call to the `logValue` ABI function followed by the write operation instead of generating a call to the `write` function. The `logValue` function logs the old value of an address without tracking conflicts to that address. The `logBulk` function similarly logs a memory range and can be used to log aggregates.

3.2 Barrier optimizations

Our design strategy of decoupling the compiler and generated code from the STM runtime precludes many optimizations on the read and write barriers. The compiler cannot in-

```
     read(x)
      /   \
     /     \
read(x)  read(y)
     \     /
      \   /
     write(x)
        |
     read(y)
```

(a) unoptimized

```
       read(x)
        /   \
       /     \
readAR(x) read(y)
       \     /
        \   /
      writeAR(x)
          |
      readAR(y)
```

(b) after barrier optimization

```
read(x)
   |
writeAR(x)
   |
read(y)
```

(c) after redundancy elimination

```
readFW(x)
   |
writeAW(x)
   |
read(y)
```

(d) after read-for-write optimization

Figure 3. Memory access optimizations

line barrier fast path code sequences into the generated code, for example, as such code sequences depend on the STM algorithm. Similarly, redundancy elimination can't eliminate barriers that may appear redundant as such optimizations also depend on the STM (e.g., a read barrier that is dominated by a write barrier to the same location is redundant in an in-place-update STM but not in a write-buffering STM).

To enable barrier optimization, the ABI provides specialized read and write barriers that encode the interesting redundancy patterns between read and write operations. These barriers allow the compiler to communicate the results of redundancy analysis to the STM runtime. Section 4.2.3 describes how the runtime eliminates redundant STM operations in these specialized read and write barriers for the different STM algorithms.

The `readAW` and `writeAW` barriers are used instead of regular read and write barriers where a read or write access to a given location is dominated by another write to the same location executed within the same transaction. (`readAR` stands for "read-after-read", `readAW` stands for "read-after-write", and `readFW` stands for "read-for-write", etc.) The `readAR` barrier is used instead of a regular read barrier where a read access to a given location is dominated by another read from the same location executed within the same transaction. The compiler uses the `readFW` barrier instead of a regular read barrier when it detects that a read operation from a given location will always be followed by a write to the same location. To use this barrier, the compiler first transforms the write operation into an internal `writeAR` operation and then transforms it into a `writeAW` operation after it replaces the read that dominates the write with a `readFW` operation.

Figure 3 shows an example of memory access optimizations using the specialized read and write barriers. This figure shows a control flow graph at different optimization stages.

3.3 Function calls

A function's annotation determines whether the compiler generates a transactional clone of that function. The clone contains calls into the runtime ABI and has a mangled version of the function's name. The compiler clones all `tm_callable` functions and those `tm_unknown` functions that it detects might be called from inside a transaction. A simple inter-procedural analysis detects the set of functions reachable from inside a transaction.

The function annotations also determine the code generation strategy for direct function calls. Within transactional code, the compiler generates calls to the mangled names of cloned functions and to the `tm_wrap` functions for wrapped functions; otherwise, it generates calls to the original uninstrumented function. For `tm_unknown` functions that it didn't clone, it inserts a call to the `switchToSerialMode` ABI function before the call. Calls to `tm_pure` functions go to the uninstrumented version without switching to serial mode as the programmer has asserted that these functions are safe to execute inside of transactions without instrumentation.

Indirect function calls are slightly more complicated. Function pointer types do not have annotations, so it is unclear to the compiler whether the target of an indirect call has a clone or whether it requires serial execution because it is precompiled and not a `tm_pure` function. All function pointers point to the original uninstrumented code so that indirect calls outside of transactions are not affected. Similar to [46], the compiler generates a marker at the beginning of every uninstrumented function that has a clone; Figure 4 illustrates this marker. Inside transactions, an indirect function call first

```
ORG_FUNC_ENTRY:
    jmp RENAMED_ORG_FUNC_ENTRY
    mov eax, TM_INDIRECTION_MAGIC
    jmp CLONED_FUNC_ENTRY
RENAMED_ORG_FUNC_ENTRY:
    // original (un-instrumented)
    // function code starts here
CLONED_FUNC_ENTRY:
    // cloned (instrumented)
    // function code starts here
```

Figure 4. Un-instrumented function's prologue

checks whether the TM_INDIRECTION_MAGIC value exists at a fixed offset from the target address in the function pointer. If it does, then a cloned version of the function exists and the call jumps to the clone's address (CLONED_FUNC_ENTRY). Otherwise, the call first switches to serial execution mode before calling the original address in the function pointer. This scheme adds an extra level of indirection to indirect function calls inside of transactions.

This technique for implementing indirect calls unfortunately treats precompiled tm_pure functions as tm_unknown functions: Inside transactions indirect calls to precompiled tm_pure functions trigger serial execution. To avoid this for tm_pure function that it recompiles, the compiler generates a prologue similar to the one presented in Figure 4 but with both jump targets pointing to the same un-instrumented version of the code.

Generating calls to virtual member functions requires special care. Because of the virtual function overriding and inheritance rules defined in Section 2.6, the compiler knows that all functions that override a tm_callable or tm_pure virtual function will have the same annotation. So calls to tm_callable virtual functions can indirectly call the clone of the target function via the address in the target function's prologue, and calls to tm_pure virtual functions can simply call the target function without any checks. Calls to tm_unknown virtual functions, however, must dynamically check whether a clone of the target function exists using the same technique as for indirect calls.

Inlining transactionally annotated functions also requires special care when such functions are inlined into atomic blocks or into transactional clones. The compiler's intermediate representation includes special code markers that allows inlining of tm_pure functions. These markers ensure that the compiler omits transactional instrumentation for the inlined body of a tm_pure function. The compiler automatically promotes the inlined body of a tm_unknown or unannotated function to tm_callable if it inlines the function into another transactional clone or atomic block.

4. TM runtime

In this section we describe runtime support for multi-mode execution, discuss in detail our STM algorithms, introduce our contention management strategy, and present a technique enabling safe in-transaction explicit memory allocation and deallocation.

4.1 Execution modes

The STM runtime supports four execution modes: (1) optimistic, (2) pessimistic, (3) obstinate, and (4) serial. The first two execution modes use an STM algorithm that implements in-place updates (eager versioning) with strict two-phase locking[17] for writes. The algorithm implements both optimistic and pessimistic concurrency control for reads (the optimistic and pessimistic modes, respectively) and allows the runtime to choose dynamically between these two modes on a per-transaction basis. The STM system can switch between these two modes mid-way through a transaction. It allows optimistic and pessimistic transactions to execute concurrently and to read the same data. This is the first STM algorithm that can support both forms of concurrency control at the same time, while preserving important safety properties such as privatization safety [45, 34]. We refer to this new STM algorithm as the *unified STM algorithm*.

A transaction running in serial mode never conflicts with another transaction – regular transactions are forbidden to run concurrently with a serial transaction. A transaction running in obstinate mode always wins all conflicts with other transactions – regular transactions are allowed to run concurrently with the obstinate one, but the obstinate transaction has the highest conflict resolution priority of all transactions in the system. The serial mode provides a mechanism for executing precompiled code and unrestricted I/O operations, while the obstinate mode provides an efficient execution mode for long running transactions that are likely to have conflicts and are expensive to roll back.

To allow dynamic switching between modes, each transaction descriptor contains a pointer to a function dispatch table containing functions that implement the mode-specific compiler-runtime ABI functions. Each of the four execution modes defines its own function dispatch table. The ABI functions are implemented as indirect calls through the mode pointer to the actual functions implemented for the mode. To switch modes a transaction simply points its descriptor's dispatch table pointer to the desired mode's dispatch table. Although the extra indirection adds overhead to each read and write barrier, it allows the runtime to select the most efficient execution mode for each transaction. Section 4.4 discusses mode switching in more detail.

A serial transaction containing no user abort statements does not require instrumentation because it is guaranteed to commit and there are no other concurrent transactions with which it can conflict. When starting such a transaction, the runtime selects execution of un-instrumented code if the compiler has indicated that it has generated an un-instrumented version of an atomic block. But in case the compiler did not generate an uninstrumented version of an atomic block, (e.g., to minimize code bloat) the runtime has a serial mode dispatch table with trivial read and write barriers that simply access memory. In addition, the runtime has a fifth *serial atomic* dispatch table to support serial mode execution of atomic blocks that contain user abort statements. To support roll back, write barriers in this dispatch table log old values on writes.

4.2 Unified STM algorithm

The pessimistic mode algorithm uses a bit-vector to represent the set of visible readers who have locked a memory location for shared reading. Pessimistic reader-writer locks automatically provide observable consistency and, in the ab-

	Optimistic TxnRec		Pessimistic TxnRec		
State	Encoding	Upper bit values	Encoding	Upper bit values	Meaning
Shared	x..x1	version number	0..001	all zero	no pessimistic readers
			x..x01	bit-vector of readers	read locked
			x..x11	bit-vector of readers	read locked with pending upgrade
Exclusive	x..x0	owner TxnDesc	0..000	all zero	write locked by optimistic
			x..x00	owner bit mask	write locked by pessimistic

Figure 5. Transaction record encoding

sence of optimistic readers, also automatically provide privatization safety [34, 33]. The optimistic mode algorithm uses a timestamp-based algorithm that incrementally updates a transaction's timestamp by validating the transaction each time it reads a value that has a more recent time stamp. Like other timestamp-based algorithms [46, 13], it keeps a consistent read set thus maintaining observable consistency. To implement privatization safety in the presence of optimistic readers, all transactions must quiesce [46, 13, 33] on commit.

The quiescence algorithm maintains a global *quiescence list* of timestamps for all in-flight transactions. Every pessimistic transaction has an infinite timestamp. Every optimistic transaction sets its timestamp at the beginning of its execution to the value of the global timestamp, and updates it every time it successfully validates its read set. Before committing, every transaction must wait for all transactions whose timestamp in the list is smaller than its own timestamp. After a transaction commits or aborts it sets its corresponding timestamp to infinity.

A pointer-sized transaction record (or TxnRec) tracks the transactional state of aligned memory blocks accessed inside transactions. The memory block size is a parameter of our system that we set to be the same as the cache line size but can be any power-of-two size. A fixed-sized transaction record table contains all of the transaction records. Each table entry has two TxnRecs, one for optimistic and the other for pessimistic concurrency control. A hash function maps memory addresses to entries in this table.

A TxnRec table entry can be in either the *shared* state, indicating that multiple transactions can read the data that maps to that entry, or the *exclusive* state, indicating that a single owning transaction can read or write the data that maps to it. Figure 5 summarize the pessimistic and optimistic TxnRec bit encodings. In the exclusive state, the optimistic TxnRec contains a pointer to the descriptor of the owning transaction while the pessimistic TxnRec contains a bit mask uniquely identifying the exclusive owner. In the shared state, the optimistic TxnRec contains a timestamp value while the pessimistic TxnRec contains a bit-vector representing the visible readers who have read-locked the data that maps to the TxnRec. The least-significant bit of the optimistic TxnRec distinguishes between the shared and exclusive states: Timestamps are odd values and transaction descriptor pointers point to word-aligned structures. Similarly for the least-significant bit of the pessimistic TxnRec.

4.2.1 Write barrier algorithm

On a write, a transaction first acquires exclusive ownership of the pessimistic TxnRec regardless of whether it is running in optimistic or pessimistic mode. Once it has acquired exclusive ownership of the pessimistic TxnRec, it then changes the optimistic TxnRec to the exclusive state. A transaction can change the value of the optimistic TxnRec only when holding exclusive ownership of the pessimistic TxnRec. To release exclusive write ownership, a transaction releases the optimistic TxnRec (by storing a new timestamp into it) before releasing the pessimistic TxnRec. The pessimistic TxnRec thus acts as a write-lock for the entire table entry. This implies that a transaction always has exclusive ownership of the pessimistic TxnRec if it has exclusive ownership of the optimistic TxnRec.

Figure 6 shows the write barrier algorithm for both optimistic and pessimistic modes.(We show only the barrier algorithms for accessing an integer. The algorithms for other data types are similar.) The write barrier first acquires ex-

```
writeInt(txn, addr, val) {
  acquireLock(txn,addr);
  logUndoInt(txn, addr);
  *addr = val;
}
acquireLock(txn,addr) {
  txnRecPtr = getTxnRecPtr(addr);
  txnRec = txnRecPtr->pessimistic;
  if (tnxRecPtr->optimistic == txn)
    return;  /* already have ownership */
  if (isReadOrWriteLocked(txnRec) ||
      !CAS(&txnRecPtr->pessimistic,txnRec,
           txn->ownerBitMask)
    { acquireLockSlow(txn,addr); }
  logWrite(txn,txnRecPtr);
  if (txnRecPtr->optimistic > txn->localTimeStamp)
    { validate(txn); }
  /* lock the optimistic TxnRec */
  txnRecPtr->optimistic = txn;
}
```

Figure 6. Write barrier algorithm

```
validate(txnDesc) {
  ts = globalTimeStamp;
  for (txnRecPtr in txnDesc->readSet) {
    txnRec = txnRecPtr->optimistic;
    if (isWriteLocked(txnRec)) {
      if (txnRec != txnDesc)
        txnAbort(txnDesc):
    } else {
      if (txnRec > txnDesc->localTimeStamp)
        txnAbort(txnDesc);
    }
 }
 txnDesc->localTimeStamp = ts;
 updateQuiescenceList(ts);
}
```

Figure 7. Validation algorithm

clusive ownership on the blocks containing the accessed data (`acquireLock`) and then logs the old value in the undo log (`logUndoInt`) before performing the write. The `acquireLock` function checks for redundant lock acquisition requests and then attempts to acquire exclusive ownership of the pessimistic TxnRec if it detects no data access contention. If exclusive ownership of the pessimistic TxnRec cannot be immediately acquired, execution falls into the slow path. The `acquireLockSlow` function (not shown) handles the slow case of handling conflicts with other transactions and upgrading read locks to write locks. Section 4.3 describes contention management in more detail.

After acquiring ownership of the pessimistic TxnRec, a transaction executing the write barrier logs a pointer to the TxnRec into the write set (for later unlocking) and puts the optimistic TxnRec into the exclusive state. Then, if this transaction is optimistic, it validates its read set if the optimistic TxnRec has a later timestamp than the transaction's current time stamp. (Since the transaction has acquired ownership of the pessimistic TxnRec, the optimistic TxnRec must be holding a time stamp). The validation procedure (Figure 7) checks that for each optimistic TxnRec in the read set either the transaction has exclusive ownership of that TxnRec or the timestamp of the TxnRec is not greater than that of the current transaction's. This ensures that the transaction sees a consistent view of memory. The validation procedure also updates the local time stamp of the current transaction to reflect that it is consistent with respect to the current global time stamp and updates the transaction's timestamp in the global quiescence list.

4.2.2 Read barrier algorithm

The optimistic and pessimistic execution modes use different read barriers, and each mode uses its respective TxnRec. Figure 8 shows the optimistic mode read barrier. The read barrier executes a straight line fast path for the case in which the transaction already owns the optimistic TxnRec, or the case in which the TxnRec is not owned exclusively by anyone and has an earlier timestamp than the current transaction. It delegates all other cases to a slow path (`readSlowOptimisticInt`). The fast path logs a pointer to the TxnRec into the read set (for later validation) in the case where the TxnRec is not exclusively owned by anyone.

```
int readOptimisticInt(txnDesc,addr) {
  val = *addr;
  txnRecPtr = getTxnRecPtr(addr);
  txnRec = txnRecPtr->optimistic;
  if (txnRec == txnDesc) return val;
  if (isWriteLocked(txnRec) ||
      txnDesc->localTimeStamp < txnRec )
    return readSlowOptimisticInt(txnDesc,addr);
  logRead(txnDesc, txnRecPtr);
  return val;
}

int readSlowOptimisticInt(txnDesc,addr) {
  txnRecPtr = getTxnRecPtr(addr);
  do {
    txnRec = txnRecPtr->optimistic;
    val = *addr;
  }while(!validateAndLog(txnDesc,txnRecPtr,txnRec));
  return val;
}

int validateAndLog(txnDesc,txnRecPtr,txnRec) {
  if (isWriteLocked(txnRec) ||
      !checkReadConsistency(txnDesc,txnRecPtr,txnRec))
  {
    contentionOnRead(txnDesc, txnRecPtr);
    return 0;
  }
  logRead(txnDesc,txnRecPtr);
  return 1;
}

int checkReadConsistency(txnDesc,txnRecPtr,txnRec){
  if (txnRec > txnDesc->localTimeStamp)
    validate(txnDesc);
  return *txnRecPtr == txnRec;
}
```

Figure 8. Optimistic read barrier algorithm

The optimistic mode read barrier slow path loops reading both the TxnRec and the data until it sees that a TxnRec is not locked by another transaction. The `validateAndLog` function verifies that a TxnRec is not owned by another transaction and also post-validates the TxnRec using the function `checkReadConsistency`. Post-validation ensures that the transaction's read set is consistent with the timestamp stored in the optimistic TxnRec and that the value of the timestamp has not changed in the meantime. On contention, control passes to the `contentionOnRead` function in the contention manager.

```
int readPessimisticInt(txnDesc,addr) {
  txnRecPtr = getTxnRecPtr(addr);
  txnRec = txnRecPtr->pessimistic;
  if (!isLockedByMe(txnDesc,txnRec)) {
    return readSlowPessimisticInt(txnDesc,txnRecPtr);
  }
  return *addr;
}

int readSlowPessimisticInt(txnDesc,txnRecPtr) {
  txnRec = txnRecPtr->pessimistic;
  while (isWriteLockedOrUpgradeRequested(txnRec) ||
         !CAS(&txnRecPtr->pessimistic,txnRec,
              txnRec ^ txnDesc->ownerBitMask)) {
    contentionOnRead(txnDesc,txnRecPtr);
    txnRec = txnRecPtr->pessimistic;
  }
  logRead(txnDesc,txnRecPtr);
  return *addr;
}
```

Figure 9. Pessimistic read barrier algorithm

Figure 9 shows the pessimistic mode read barrier. The read barrier fast path completes successfully if the transaction already owns a read or write lock on the TxnRec; otherwise the execution falls into the slow path. The slow path loops until it can acquire a read lock on the TxnRec and then logs a pointer to the TxnRec into the read set (for later unlocking). Similarly to the optimistic read barrier, control passes to the `contentionOnRead` function on contention. The read barrier gives priority to any transaction who has requested an upgrade from a read lock to a write lock.

4.2.3 Optimized barriers

Section 3.2 described how the compiler communicates results of data flow analysis to the runtime via specialized barriers. This section describes how the runtime system optimizes these specialized barriers.

The `readAW` (read-after-write) and `writeAW` (write-after-write) functions contain only a simple load or store since the transaction already holds an exclusive lock for that location in both the pessimistic and optimistic modes. The `readAR` (read-after-read) functions contain a simple load for pessimistic transactions since the transaction already holds a read lock for the read location. For optimistic transactions, these functions do not update the read set since the location has already been added to the read set, and, if the current transaction's timestamp is smaller than the one stored in the optimistic TxnRec, the current transaction immediately aborts because the location may have been updated by other transactions. The `readFW` (read-for-write) function allows the runtime to take a write lock immediately, thus avoiding unnecessary read logging and avoiding the need to promote a read lock later (pessimistic read concurrency) or to perform additional validations (optimistic read concurrency).

4.2.4 Transaction commit and abort

A committing outermost pessimistic transaction first releases all its read locks. A committing outermost optimistic transaction first validates its read set, aborting if validation fails. As an optimization, this validation is performed only if the write set is not empty and the timestamp of the transaction is less than the current global time stamp (indicating that other transactions have committed since the last time the transaction validated). The commit algorithm then releases write locks and quiesces on all other in-flight optimistic transactions (to ensure privatization safety). Finally, the commit algorithm executes all the commit actions registered for a given transaction. The compiler flattens nested transactions that contain no abort statements. Nested transactions that cannot be flattened perform no actions on commit.

An aborting transaction, pessimistic or optimistic, reverts all the updates performed within the scope of the transaction and executes all the undo actions registered for a given transaction. An outermost transaction releases all its locks and consults the contention manager with respect to actions that may have to be taken before it is re-executed (e.g., exponential back-off).

4.2.5 Quiescence optimizations

The runtime performs two optimizations to reduce or avoid the cost of quiescence. The *lazy start* optimization delays declaring a transaction as optimistic for as long as possible, reducing the time interval during which other transactions need to wait for it during quiescence. Instead of setting the transaction's timestamp at transaction start, this optimization sets the timestamp to infinity until the first optimistic read barrier, at which point it sets the timestamp to the value of the global timestamp. Read after write and read for write barrier functions as well as all the write barrier functions do not set the timestamp as these barriers hold a lock for the memory access. As we show in Section 5, this technique significantly improves the performance of workloads in which all reads are read for writes or read after writes.

In the *filtering* optimization, quiescence uses a mask to skip over transactions that have not performed optimistic reads. Each transaction is assigned a *stability flag* byte in a global mask that indicates whether the transaction has performed any optimistic reads (not including read after writes or read for writes). Eight stability flags make a 64-bit integer, which can be checked in one instruction. Instead of going through all transactions one by one, quiescence can now quickly check eight transactions a time. If a 64-bit chunk of the mask in nonzero, it can check individual bytes using binary search and quiesce only on transactions with nonzero stability flag bytes. This optimization reduces quiescence cost for programs in which most transactions don't perform optimistic reads.

4.3 Contention management

The STM runtime's contention management framework allows multiple contention handling policies to be plugged in via a contention management interface. The current system provides a default policy that uses exponential back-off and a policy that uses a variant of the polka policy [49]. The `contentionOnRead()` and `contentionOnWrite()` functions handle the case in which a transaction encounters contention when attempting a data access operation, whereas the `contentionOnAbort()` function handles the case in which a transaction has aborted and is about to re-execute. Each policy implements a dispatch table containing pointers to these functions, and each transaction holds a pointer to a dispatch table for the contention management policy.

4.4 Mode switching

A transaction starts in optimistic mode. The obstinate mode guarantees forward progress of long transactions, so a contention manager may transition an optimistic transaction to obstinate mode if the transaction re-executes too many times due to conflict. If the transition to obstinate mode fails because of another obstinate transaction, then a contention manager may simply transition the transaction to pessimistic mode. The transition can occur in-flight or on re-execution of the transaction.

To become pessimistic in-flight, an optimistic transaction validates its read set and acquires pessimistic read locks for all the transaction records in its read set. If this fails, the transaction aborts and restarts in pessimistic mode. We do not support in-flight transitions in the opposite direction – a pessimistic transaction can become optimistic, but only on re-execution.

An obstinate transaction is simply a unique pessimistic transaction that has the highest conflict resolution priority in the system. An optimistic transaction can reach obstinacy only after becoming pessimistic. The system allows at most one obstinate transaction at at time, implemented using an *obstinacy token*. A transaction must acquire this token in order to become obstinate.

A serial transaction runs exclusively with respect to all other transactions – no other transaction is allowed to run while a serial transaction is running. This is different from an obstinate transaction, which can coexist with other transactions. A transaction can transition to the serial mode either in-flight or on re-execution. To transition in-flight, a transaction first becomes obstinate, which ensures that no other obstinate or serial transaction is present in the system. It then transitions to serial mode and waits until all transactions complete. No new transactions can start while a serial transaction exists.

4.5 Transactional memory management

In order for an STM system to be practical, it must provide a safe and efficient mechanism supporting memory allocation and de-allocation inside of transactions. Failing to provide adequate memory management support may lead to serious performance and safety problems, such as memory exhaustion or dangling references.

Unlike previous work on transactional memory management [27], our system builds on top of the default platform memory allocator. This avoids requiring the developer to rebuild the entire application with a custom transactional memory allocator. We use the `tm_wrap` annotation described in Section 2.9 to implement wrappers for all memory allocation functions. The wrappers also register commit and undo actions that need to be executed by the memory allocator on transaction termination.

Similar to the solution presented in [27], we associate a ticket number with each allocation and de-allocation site. Non-transactional code has an implicit ticket number equal to 1. A non-nested transaction starts with a ticket number equal to 2, increments it on the start of every nested transaction, and decrements it on every commit. Unlike in [27] our ticket numbers are thread-local and do not have to be unique across threads. Ticket numbers are used to determine if a free operation can be executed immediately or should be deferred until a future transaction commit.

In addition to allocating the requested memory, the allocation function wrapper creates an *allocation record* and stores it in a thread-local table. The allocation record contains the address of an allocated memory fragment and the current ticket number. The function wrapper also registers commit and undo actions to be executed on commit or abort of the outermost transaction. The commit action removes the allocation record from the table. The undo action removes the allocation record and de-allocates the given memory fragment.

The de-allocation function wrapper determines when to de-allocate a memory chunk based on the chunk's ticket number, which it looks up in the allocation record table using the chunk's memory address.[2] The wrapper function de-allocates memory immediately if the ticket number is greater than or equal to the current ticket number. Otherwise, it registers a commit action to be executed on commit of the innermost transaction where the ticket number of the resuming transaction meets the conditions for de-allocation. The commit action simply deallocates the given memory fragment, and deletes its associated allocation record if one exists.

5. Experimental results

To evaluate our system, we experimented with a wide range of workloads, including STAMP[9], SPLASH2[51], and PARSEC[4]. Our results show that the optimistic mode outperforms the pessimistic mode, but the pessimistic algorithm performs competitively in many cases, especially at low con-

[2] If no record exists, the allocation site is non-transactional and assigned a ticket number of 1

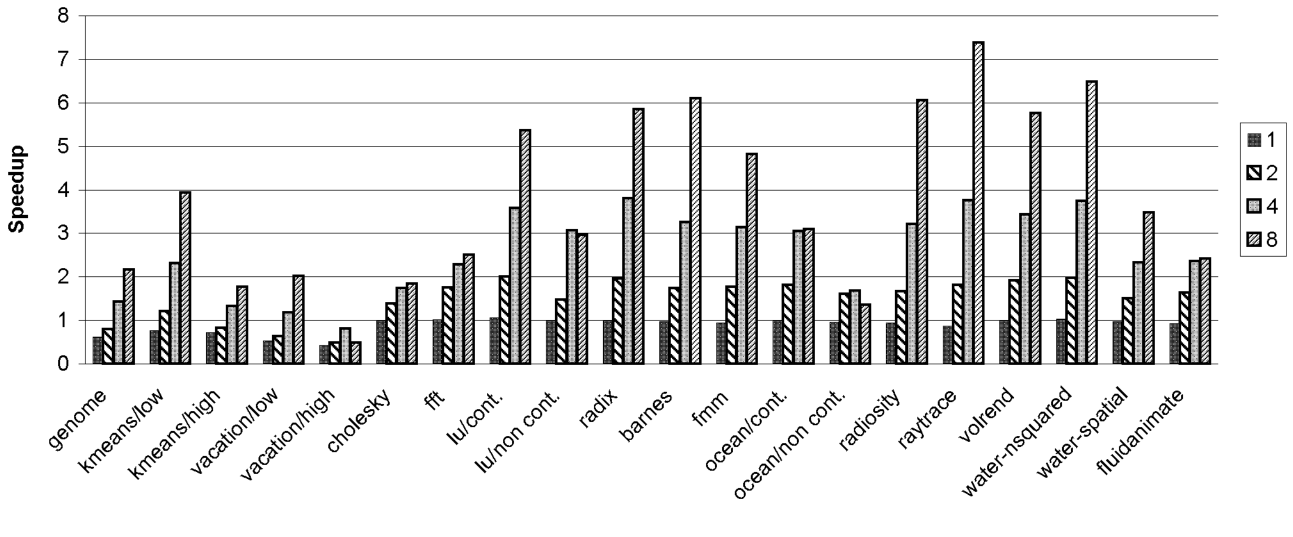


Figure 10. Scalability of STM. The numbers are reported as speedups over the single-thread execution of the same programs using coarse-grained locks, running in 1, 2, 4, and 8 threads.

tention levels. In addition, we found compiler optimizations and runtime quiescence optimizations very effective for reducing the overhead of STM and improving its scalability.

5.1 Workloads and experimental environment

We ported and ran 20 programs from the benchmark suites of STAMP[9], SPLASH2[51], and PARSEC[4] for our experiments. STAMP is a TM benchmark suite developed by Stanford. We used version 0.9.4, which contains three programs, all of which feature relatively long transactions compared to traditional parallel workloads. SPLASH2 is a benchmark suite of classical parallel programs, including numerical analysis kernels and scientific and graphics applications. According to our measurements, most of the SPLASH2 programs spend less than 1% execution time in critical sections (and some of them 5% with large input). Some of the SPLASH2 programs, however, use parallel programming patterns that are interesting from a TM perspective; for example, barnes uses double-checked locking to avoid locking overhead when loading the contents of an octtree data structure, and radiosity does privatization to take tasks off a shared task queue. We also used one program – fluidanimate – from the PARSEC suite in our evaluation. This program simulates fluids and features extremely short atomic regions (one increment operation each) in extremely large numbers (more than 10 million).

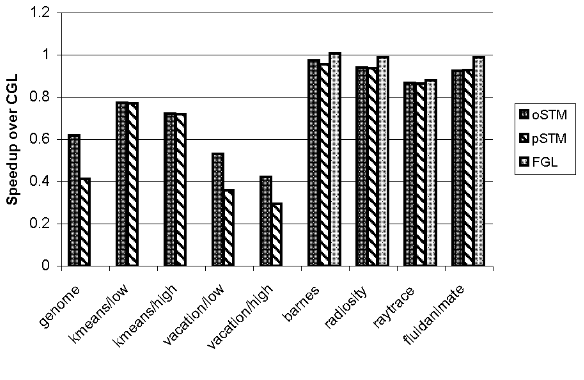
The STAMP programs were originally written using transactions. We ported STAMP to our language constructs and created a coarse-grained lock version of them by using a single global lock to guard every atomic block. The SPLASH2 programs were originally written using fine-grained locks. We created a transactional version of SPLASH2 by replacing the lock-based critical sections with atomic blocks and created a coarse-grained version by replacing all the locks with a single global lock. We did the same thing for fluidanimate to create transactional and coarse-grained lock versions. For some SPLASH2 programs, we also increased the input size. For barnes, raytrace, and radiosity, the programs finished in seconds when running in a single thread using the original input. With our input, they finished in minutes when running in a single thread.

The barnes program contains a data race in its double-checked locking pattern. This program breaks when translated to use transactions because of this data race. We rewrote barnes to remove this data race and to make it comply with the emerging C++ memory model [8]. Eliminating the data race not only fixed the problem but also improved its performance relative to locks.

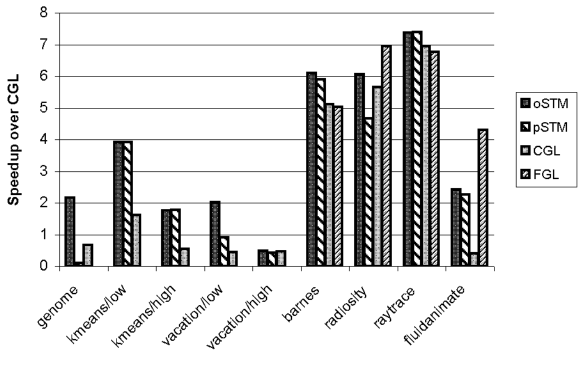
We ran our experiments on an 8-core system with 8GB of main memory running RedHat Enterprise Linux Version 4. The system had two sockets, each with a quad-core Intel Xeon X5355 (Clovertown) CPU at 2.66GHz. Each CPU had an 8MB L2 cache. Our STM was configured to use a table of 2^{20} transaction record entries and cache-line-granularity for conflict detection.

5.2 Scalability of STM

Figure 10 shows the scalability of our system, running the 20 programs from STAMP, SPLASH2, and PARSEC. The numbers are reported as speedup over single-thread execution of the same programs using coarse-grained locks, and they include all compiler and runtime optimizations. For each program, we show the speedups with 1, 2, 4, and 8 threads. Numbers greater than 1 reflect better performance



(a) Single Thread



(b) 8 Threads

Figure 11. Comparison of optimistic STM (oSTM), pessimistic STM (pSTM), coarse-grained locks (CGL) and fine-grained locks (FGL). Numbers are reported as speedup over single-thread execution of CGL. We don't have a version of STAMP using fine-grained locks.

than single-thread coarse-grained locks, while numbers less than 1 reflect worse performance.

Most of the programs scaled well using our STM. At 2 or more cores, the SPLASH2 programs and fluidanimate all perform better than the single-core coarse-grain lock configuration. Some of the SPLASH2 programs – cholesky, fft, and ocean – did not scale well due to lack of large input. These programs also don't scale using coarse-grained or fine-grained locks. At 4 or more cores, the STM configuration performed better than the single-thread coarse-grained lock configuration for all STAMP programs except vacation/high. We observed that vacation/high had many false conflicts due to cache-line granularity conflict detection, so finer-grain conflict detection should improve the performance of vacation/high. To achieve good scalability on kmeans and genome, we used `__tm_waiver` blocks for reads to shared read-only data.

Past work has considered optimistic STM superior to pessimistic mainly because the latter performs lock operations on transactional reads [40]. We measured the scalability of both, and compared them to the coarse-grained lock configuration running the same workloads. Figure 11 measures the performance of the optimistic and pessimistic modes of the runtime and compares it with the performance of fine-grained and coarse-grained locks. (Note that we don't have fine-grain versions of the STAMP programs. Also, in this figure and in the rest of this section, we limit our discussion of results for SPLASH2 to three programs – barnes, raytrace, and radiosity – which had non-trivial critical sections compared to the rest of the SPLASH2 programs.) Figure11(a) shows the single-thread speedup of the optimistic mode, pessimistic mode, and fine-grain lock configurations relative to coarse-grain locks. The results in Figure11(a) show that both STM modes imposed additional single-thread overhead compared to coarse-grained locks. This overhead was mostly caused by the compiler instrumentation for read and write barriers. STAMP programs are affected by this more than SPLASH2 or fluidanimate. The pessimistic mode incurred slightly more overhead than the optimistic mode due to the lock operations it does on reads. As the number of threads increases, STM starts overcoming this instrumentation overhead and starts beating coarse-grained locks. The results in Figure 11(b)) show that at 8 threads STM performs better than coarse-grained locks, and for SPLASH2 performs close to fine-grained locks. In most programs, pessimistic performs close or equal to optimistic, but in genome and vacation/low, pessimistic performs significantly worse because of the lock operations it does on reads.

5.3 Runtime quiescence optimizations

Figure 12 shows the overhead of quiescence (privatization safety) and the performance improvements from the quiescence optimizations. The numbers in this figure show speedups over a baseline STM that uses the quiescence algorithm described in Section 4.2 but without quiescence optimizations. To measure the overhead of quiescence (privatization safety), we measured the performance of our workloads with quiescence disabled. Because the majority of our benchmarks (all but radiosity) do not use the privatization pattern, it is legal to turn quiescence off for them. Disabling quiescence improved the performance of the optimistic algorithm for all the benchmarks except for vacation/high (as noted before, vacation/high suffers from false conflicts). Quiescence incurs a significant overhead on the STM, over 150% in the case of barnes and 40% or more in 5 other workloads. The filtering and lazy start quiescence optimizations both reduced the overhead of quiescence. Some workloads benefit mostly from the lazy start optimizations

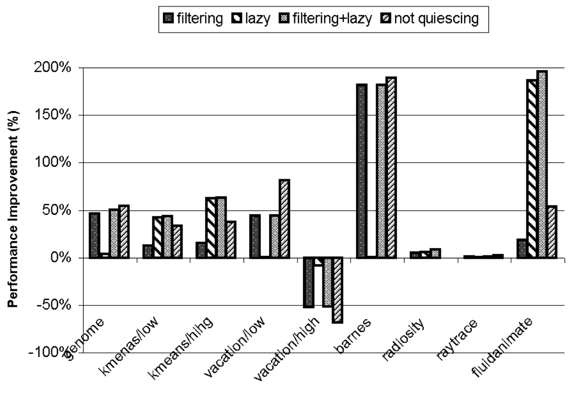


Figure 12. Optimizations for quiescence

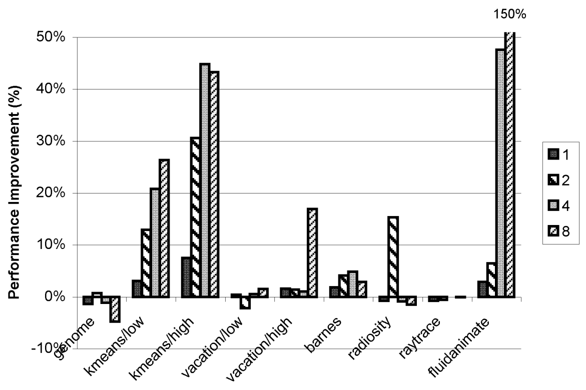


Figure 13. Compiler optimizations

while others benefit mostly from the filtering optimization, so a combination of both optimizations appears to be the best strategy. The combined optimizations gain back most of the overhead of quiescence and improve the performance of barnes and fluidanimate by almost 200%. For fluidanimiate, the optimized optimistic STM even outperformed the STM with quiescence disabled because the lazy start technique also reduced the number of read set validations in the read barriers. (Note that the numbers in Figures 11 and 10 include these two optimizations.)

5.4 Compiler optimizations

Figure 13 shows the performance improvements from using the compiler optimizations described in Section 3.2. This figure shows results for 1, 2, 4, and 8 threads as speedup over using no compiler optimizations for the same number of threads. Fluidanimate benefits the most from compiler optimizations, with a 150% improvement at 8 threads and 48% at 4 threads. Kmeans/low and kmeans/high also benefit significantly, with improvements of 26% and 43%, respectively, at 8 threads. These improvements were due mostly to the read-for-write barriers introduced by the compiler optimizations and described in Section 3.2. For fluidanimiate and kmeans, this optimization turns all read barriers into read-for-write barriers, which combined with the quiescence optimization avoids quiescence and validation costs.

6. Related work

Several STM systems have introduced the basic atomic block language construct into C [46, 10, 3, 35, 15, 28]. Some of these approaches introduce pragmas [46, 28] and OpenMP extensions [3, 35]. In contrast, our work aims to introduce first class C++ language constructs for TM that are orthogonal to the programming model and interact correctly with other C++ language features. Crowl et al. [10] explore some of the high-level alternatives to introducing TM language constructs into C++. Other systems have introduced the basic atomic block language construct into managed languages [20, 21, 39, 2, 22, 26]

Several of these past systems also provide an explicit construct to rollback a transaction in the form of either an abort [46, 10] or a retry statement [21, 2, 3, 35]. The commit and abort actions available in our system are reminiscent of the handlers used in the recent proposals for open-nested transactions [38] and for transactional boosting [23], in which handlers can be used to implement finalizing and compensating actions. In addition to certain subtle differences in their behavior, handlers used with open nesting differ from actions used in our system by being automatically executed as separate transactions.

Instead of first-class language constructs, some systems provide an STM library interface [25, 13, 24, 31, 14]. Some of these systems leverage C++ language features (such as multiple inheritance, templates, and operator overloading) to eliminate the syntactic clutter associated with STM APIs [11, 16]. In contrast, first-class language extensions provide syntactic convenience to the programmer, enable compiler optimizations for TM, and enable static analyses that provide static guarantees.

Different STM systems implement different types of concurrency control. Existing eager versioning (i.e., in-place update) STM systems support pessimistic concurrency control for writes and either optimistic [2, 22] or pessimistic [40] concurrency for reads. Lazy versioning (i.e., write-buffering) STM systems [13, 47, 20] can support optimistic concurrency control for write operations as well. Unlike the system described in this paper, none of the existing STM systems has allowed concurrent transactions that use different concurrency control mechanisms.

The idea of mode switching has appeared previously but only in the context of systems that utilize a mix of hardware and software transactional memory techniques [41, 12, 29, 43]. In the phased TM approach [29], all transactions in the system execute in the same mode – one failing hardware transaction causes all transactions in the system to execute in software. In the Hybrid TM [12] and hardware-accelerated STM [41] approaches, a transaction may execute either in a pure software mode or in a mode that uses hardware support for TM, and transactions using different modes can execute concurrently.

Memory models for STM systems and issues concerning transactional semantics in general have recently attracted significant attention. Blundell et al. [5] introduced the notions of weak and strong atomicity. Shpeisman et al. [42], Abadi et al. [1], and Moore and Grossman [36] have all further investigated these issues. Menon et al. [34, 33] and Grossman et al. [18] discuss several other TM memory model issues, along with implications that language memory models, such as the Java Memory Model [30] or the emerging memory model for C/C++ [8], may have for TM. Finally, issues concerning the desirable safety properties that STM systems should preserve when executing concurrent transactions, such as privatization or publication safety, have been explored by Spear et al. [45], Menon et al. [34, 33] and Abadi et al. [1], among others.

In order to guarantee safety of STM systems in a more general sense, certain key mechanisms must be designed and implemented carefully to work correctly in a transactional context. Correct handling of I/O and calls to legacy code from inside transactions was first proposed by Blundell et al. [6] for hardware transactions and then by Spear et al. [44] and Welc et al. [48] in the context of an STM. Solutions for safe and efficient memory management in STM systems have been addressed by Hudson et al. [27] (explicit memory allocation for unmanaged languages) and McGachey et al. [32] (automatic memory management through garbage collection).

STM systems must efficiently manage contention between concurrently executing transactions. Herlihy et al. [24] introduced the concept of contention managers separated from the rest of the STM system. In [49] and [50], Scherer and Scott further investigated and evaluated several different contention management policies. Examples of other topics investigated in this area include contention management policies with provable worst case properties (Guerraoui et al. [19]) or identification of performance pathologies that may result from using certain contention management policies (Bobba et al. [7]).

7. Conclusions

In this paper we presented a software transactional memory system that introduces first-class C++ language constructs for transactions. We described new C++ language extensions to support transactional memory. These constructs support C++ language features such as classes, inheritance, virtual functions, exception handling, and templates. We extended an existing, high-performance production C/C++ compiler to translate and optimize these new language extensions. We presented a novel STM runtime library implementing both optimistic and pessimistic concurrency control, as well as other important features such as support for calls to legacy binaries and unrestricted I/O. We also presented a thorough experimental evaluation of our STM system on a large set of TM workloads and demonstrated that our system performs well across these workloads.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL 2008*.

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

[3] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM transactional application programming interface. In *PACT 2007*.

[4] C. Bienia, S. Kumar, J. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report 811-08, Princeton University, 2008.

[5] C. Blundell, E. C. Lewis, and M. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), Nov. 2006.

[6] C. Blundell, E. C. Lewis, and M. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report CIS-06-09, University of Pennsylvania, Department of Comp. and Info. Science, 2006.

[7] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *ISCA 2007*.

[8] H. J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *PLDI 2008*.

[9] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA 2007*.

[10] L. Crowl, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Integrating transactional memory into C++. In *TRANSACT 2007*.

[11] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in C++. In *TRANSACT 2007*.

[12] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS 2006*.

[13] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC 2006*.

[14] R. Ennals. Software transactional memory should not be obstruction-free. http://www.cambridge.intel-research.net/rennals/notlockfree.pdf, 2005.

[15] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süßkraut, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT 2007*.

[16] J. Gottschlich and D. A. Connors. DracoSTM: A practical C++ approach to software transactional memory. In *LCSD 2007*.

[17] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[18] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *MSPC 2006*.

[19] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust contention management in software transactional memory. In *SCOOL 2005*.

[20] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 2003*.

[21] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP 2005*.

[22] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI 2006*.

[23] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP 2008*.

[24] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC 2003*.

[25] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA 2006*.

[26] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC 2006*.

[27] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: A scalable transactional memory allocator. In *ISMM 2006*.

[28] IBM. *IBM C/C++ for Transactional Memory, for AIX, V0.9, Language Extensions and User's Guide*. http://dl.alphaworks.ibm.com/technologies/xlcstm/xlcstm-whitepaper.pdf.

[29] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *TRANSACT 2007*.

[30] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL 2005*.

[31] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, I. William N. Scherer, and M. L. Scott. Lowering the overhead of software transactional memory. In *TRANSACT 2006*.

[32] P. McGachey, A.-R. Adl-Tabatabai, R. L. Hudson, V. Menon, B. Saha, and T. Shpeisman. Concurrent GC leveraging transactional memory. In *PPoPP 2008*.

[33] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for Java STM. In *SPAA 2008*.

[34] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Single global lock semantics in a weakly atomic STM. In *TRANSACT 2008*.

[35] M. Milovanović, R. Ferrer, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, and M. Valero. Multithreaded software transactional memory and OpenMP. In *MEDEA 2007*.

[36] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL 2008*.

[37] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and preliminary architecture sketches. In *SCOOL 2005*.

[38] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP 2007*.

[39] M. F. Ringenburg and D. Grossman. AtomCaml: first-class atomicity via rollback. In *ICFP 2005*.

[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 *PPoPP 2006*.

[41] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO 2006*.

[42] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and S. Bratin. Enforcing isolation and ordering in STM. In *PLDI 2007*.

[43] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstata, C. Heriot, I. William N. Scherer, and M. F. Spear. Hardware acceleration of software transactional memory. In *TRANSACT 2006*.

[44] M. Spear, M. Michael, and M. Scott. Inevitability mechanisms for software transactional memory. In *TRANSACT 2008*.

[45] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. Technical Report 915, University of Rochester, Computer Science Dept., 2007.

[46] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO 2007*.

[47] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for Java synchronization. In *ECOOP 2006*.

[48] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA 2008*.

[49] I. William N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC 2003*.

[50] I. William N. Scherer and M. L. Scott. Contention management in dynamic software transactional memory. In *CSJP 2004*.

[51] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA 1995*.