Towards Transactional Memory Semantics for C++

Tatiana Shpeisman
Intel Corporation
Santa Clara, CA
tatiana.shpeisman@intel.com

Ali-Reza Adl-Tabatabai
Intel Corporation
Santa Clara, CA
ali-reza.adl-tabatabai@intel.com

Robert Geva
Intel Corporation
Santa Clara, CA
robert.geva@intel.com

Yang Ni
Intel Corporation
Santa Clara, CA
yang.ni@intel.com

Adam Welc
Intel Corporation
Santa Clara, CA
adam.welc@intel.com

ABSTRACT

Transactional memory (TM) eliminates many problems associated with lock-based synchronization. Over recent years, much progress has been made in software and hardware implementation techniques for TM. However, before transactional memory can be integrated into mainstream programming languages, we must precisely define its meaning in the context of these languages. In particular, TM semantics should address the advanced features present in the existing software TM implementations, such as interactions between transactions and locks, explicit user-level abort and support for legacy code.

In this paper, we address these topics from both theoretical and practical points of view. We give precise formulations of several popular TM semantics for the domain of sequentially consistent executions and show that some of these semantics are equivalent for C++ programs that do not contain other forms of synchronization. We show that lock-based semantics, such as Single Global Lock Atomicity (SLA) or Disjoint Lock Atomicity (DLA), do not actually guarantee atomicity for race-free programs and propose a new semantics, Race-Free Atomicity (RFA) that gives such a guarantee. We compare these semantics from the programmer and implementation points of view and explain why supporting non-atomic transactions is useful. Finally, we propose a new set of language constructs that let programmers explicitly specify whether transactions should be atomic and describe how these constructs interact with user-level abort and legacy code.

Categories and Subject Descriptors

D.1.3 [**Programming techniques**]: Concurrent Programming—*Parallel Programming*; D.3.3 [**Programming languages**]: Language Constructs and Features—*Concurrent programming structures; Frameworks*

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.

General Terms

Design, Languages, Theory

Keywords

Atomicity, C++, Serializability, Synchronization, Transactional Memory

1. INTRODUCTION

Transactional memory (TM) promises to provide a simpler synchronization abstraction than locks. From a programmer's point of view, a transaction executes atomically; that is, as a single indivisible operation whose steps do not interleave with actions of other threads. TM implementations enable concurrency by executing transactions in parallel whenever such execution does not violate the illusion of atomicity. Transactional memory has been a topic of active research for several years and is now on the verge of becoming a practical programming technology. Multiple software TM implementations have been available for a while [18, 16, 25, 17, 24, 12, 31], and Sun has recently announced hardware TM support in an upcoming machine [11]. Before, however, transactions can be adopted as a practical programming technique by a wide community of programmers, it is necessary to define their precise meaning in the context of mainstream programming languages such as C/C++ and Java.

Despite years of transaction research in the context of database systems, integrating transactions into programming languages has turned out to be a non-trivial task. The challenges of defining transactional memory semantics come from the significant differences between the way shared data are accessed in database systems and in multi-threaded programs. In the database world, all data accesses are transactional. In the context of programming languages, shared data may be accessed both inside and outside of transactions. In this case, most software TM (STM) implementations do not provide an illusion of atomicity, as they do not detect data conflicts between transactional and non-transactional memory accesses.

In this paper we investigate TM semantics in the context of C++. C++ gives well-defined behavior only to race-free programs [6, 8]. A programmer is expected to synchronize concurrent accesses to shared data by using locks or special $atomic < T >$ variables (which behave similar to Java

Initially `x = 0`

	Thread 1	Thread 2
1:	`atomic {`	
2:	`t1 = x;`	
3:		`x = 1;`
4:	`t2 = x;`	
5:	`}`	

Can t1 != t2 ?

Figure 1: Under SLA semantics, this C++ program can have arbitrary behavior because it contains a data race on `x`. In particular, Thread 1 is allowed to see `t1 != t2`.

Initially `atomic<int> v = 0, atomic<int> w = 0`

	Thread 1	Thread 2
1:	`atomic {`	
3:	`v = 1;`	
5:		`while (v != 1) {}`
6:		w = 1;
7:	`while (w != 1) {}`	
8:	`}`	

Can this program terminate?

Figure 2: SLA does not guarantee atomicity in race-free programs.

volatile variables). In turn, the system guarantees *sequential consistency* — a program appears to execute as a simple interleaving of actions of all threads in which each read of a shared variable sees the last value written to that variable. [1]

Many researchers have suggested that transactions should have Single Lock Atomicity (SLA) semantics [20, 33, 5]. Under SLA, transactions behave as if they were protected by a single program-wide mutual exclusion lock. In race-free C++ programs, critical sections protected by the same lock appear to execute sequentially with respect to each other. Similarly, the actions of one transaction appear to complete either before or after all actions of another transaction. Thus, SLA effectively guarantees that transactions are serializable. SLA is an attractive choice for C++ TM semantics because of its simplicity and practicality. Because the C++ memory model gives undefined semantics to programs with data races, TM implementations do not have to provide *strong atomicity* [4]; that is, they do not have to isolate transactions from concurrent non-transactional accesses that form data races with transactional memory accesses. For example, SLA semantics does not require TM implementations to track conflicts between transactional and non-transactional accesses to `x` in the program shown in Figure 1; the behavior of this program is undefined because it contains a data race on `x`. Recent work has demonstrated that SLA can be efficiently implemented in an STM without imposing any restrictions on the programming actions performed inside transactions [28].

In one respect, however, SLA weakens the promise of TM: When transactions contain *nested synchronization* (that is, locks or C++ atomic variables), SLA allows behaviors that could not occur if transactions executed as indivisible actions. Consider the example in Figure 2. Thread 1's transaction communicates a value to non-transactional code in Thread 2 and then waits for the response. The communication is performed via C++ atomic variables, which are synchronization constructs and do not contribute to data races. This program terminates only if non-transactional accesses in Thread 2 are allowed to violate the atomicity of Thread 1's transaction. [2] SLA and other lock-based semantics (e.g., DLA, ALA and ELA [26]) allow this behavior as locks in general do not guarantee atomicity for race-free programs [13].

SLA and other lock-based semantics also provide a poor basis for defining the semantics of user-level abort. A user-level abort (such as the `__tm_abort` keyword in [28]) allows a programmer to roll back a transaction explicitly. Critical sections protected by locks cannot be rolled back, so lock-based analogies provide no guidance with respect to the semantics of a user-level abort.

Another complication for C++ TM semantics comes from the requirement to support *legacy code*; that is, uninstrumented code that has not been compiled for transactional execution. Supporting uninstrumented code allows transactions to perform unrestricted I/O and to interact with existing precompiled libraries that are available only in binary form. But the requirement of supporting legacy code conflicts with the requirement to support user-level aborts and with the desire to provide atomicity. Without instrumentation, STM systems cannot support user-level abort of legacy code. Furthermore, without instrumentation, STM systems cannot detect conflicts between legacy code and transactions. STM systems can still provide SLA semantics in this case by serializing the execution of all transactions, but STM systems cannot practically guarantee atomicity for race-free programs that contain lock-based synchronization hidden in legacy code. [3]

We believe that the best way to resolve these conflicting requirements and at the same time to provide the programmer with transactions that are guaranteed to be atomic is to provide two sets of language constructs for transactional memory. The first construct guarantees that a transaction is atomic and can be rolled back via a user-level abort at any time, but it disallows the transaction to call legacy code. The second construct guarantees that a transaction is serializable and allows the transaction to call legacy code, but it places restrictions on user-level aborts inside the transaction.

In this paper, we propose a practical TM semantics for C++ that follows these design principles. In particular, we make the following contributions:

- We introduce a framework for describing and reasoning about transactional properties and the semantics of

[1] C++ also provides low-level atomic variables that support more complicated behaviors; these constructs, however, are intended for expert programmers with good understanding of weak memory models.

[2] Some might argue that this behavior violates isolation rather than atomicity. Indeed, there is a mismatch in terminology used by the programming languages and database communities. We side with the programming language community and use the term atomicity to describe the situation where all actions inside a transaction do not interleave with the actions of other threads.

[3] STM systems could provide atomicity in this case by stopping all non-transactional threads. It might also be possible to provide atomicity in this case by using binary translation or hardware support for unbounded transactional memory. We do not consider such approaches practical for C++.

transactions in the context of programming languages such as C++ whose semantics assign well-defined behaviors only to race-free programs. (Section 2.)

- We investigate the properties of SLA and compare it to a weaker semantics, Disjoint Lock Atomicity (DLA) [26]. We prove that these semantics are equivalent for C++ programs without nested synchronization. Furthermore, both DLA and SLA provide atomicity for race-free programs without nested synchronization. We further argue that DLA makes reasoning for the programmer more complicated without providing significant additional implementation flexibility. (Section 3.)

- We define a new TM semantics called Race-Free Atomicity (RFA) that guarantees atomicity for all race-free programs, including those with nested synchronization. Furthermore, we prove that for programs without nested synchronization SLA and RFA semantics are equivalent. (Section 4.)

- We discuss the semantics of user-level abort and show that completely ignoring the effects of aborted transactions imposes requirements that effectively eliminate the implementation flexibility of semantics weaker than strong atomicity. (Section 5.)

- We propose a new set of language-level transactional constructs that explicitly differentiate between transactions that provide atomicity and transactions that allow unrestricted use of legacy code. We also describe how these constructs interact with user-level abort and nested synchronization. (Section 6.)

2. FRAMEWORK

In this section, we describe a framework for reasoning about TM semantics. Our framework considers only sequentially consistent executions. This restriction is consistent with C++ memory model that assigns sequentially consistent behavior to correctly synchronized or race-free programs [4].

We use a simple framework based on the C++ memory model without low-level atomics [6]. A thread execution consists of a set of actions together with a partial order on these actions defined by the *sequenced-before* relation [8]. (C++ sequenced-before relation is analogous to the Java program order relation [23], except that it leaves the order of argument evaluation undefined.)

Thread actions may include the following: reads and writes to shared memory (*data accesses*), reads and writes to synchronization variables (that is, C++ atomic variables), lock acquire and release operations, and special start and commit transaction operations that denote the scope of a transaction. Given a transaction T, we use the notation T_S and T_C to denote its start and commit operations. Each transaction start operation should have a matching transaction commit operation. A transaction start operation is sequenced before all other actions executed by the transaction. All actions executed by a transaction are sequenced before its commit operation. At this point, we assume that transactions do not contain user-level abort. For simplicity, we also do not allow nested transactions. (We can easily extend our framework to support closed nested transactions, as such transactions do not affect the inter-thread semantics.)

We call lock operations and operations on synchronization variables *synchronization actions*. We call actions executed within the scope of a transaction (that is, the actions that are ordered in between of start and commit operations of some transaction by the sequenced-before relation) *transactional actions*. We call other actions *non-transactional actions*. We use the term *nested synchronization* to denote transactional actions that are synchronization actions.

A program execution consists of a set of thread executions together with a total *execution order* on actions of all threads, which satisfy the following properties: (1) Each thread execution is consistent with the C++ single-threaded semantics. (2) The execution order is consistent with the sequenced-before orders of each thread. (3) As required by sequential consistency, each read of a memory location sees the value written by the most recent write to that memory location in the execution order or the initial value of the memory location if the memory location was not written by any thread before the read.

Given a program execution E, we denote by $actions(E)$ the set of actions in that execution, and by $<_E$ the execution order of those actions. Given an action A, we denote the thread that executed A as $thread(A)$. Similarly, given a transaction T, we denote the thread that executed T as $thread(T)$. For brevity, we sometimes use the term execution to refer to a program execution.

We use a standard definition for conflicting data accesses, namely, two data accesses executed by different threads conflict if they access the same memory location, and at least one of them is a write. We also extend the definition of conflict to synchronization actions. Two synchronization actions executed by different threads conflict if either (1) they access the same synchronization variable, and at least one of them is a write, or (2) they operate on the same lock. Two transactions conflict if they execute conflicting actions.

A *data race* occurs when an execution contains two conflicting data accesses that are adjacent to each other in the execution order; that is, when two conflicting access may execute concurrently. An execution is race-free if it contains no data races. Boehm and Adve [6] show that this definition of a data race (type-1 data race) results in the same notion of program race-freedom as a Java-style definition of a data race via the happens-before relationship (type-2 data race). A program execution may contain a type-2 data race even if it contains no type-1 data race. (For example, two conflicting data accesses separated by a third non-conflicting data access form a type-2 data race but not a type-1 data race.) In this case, however, a program also has another execution where the conflicting data accesses are adjacent, and thus form a type-1 data race.

By itself, our framework places no restrictions on the interleaving of transactional actions with actions of other threads. These restrictions are imposed by a TM semantics, which we define simply as a subset of program executions. Given a semantics (a set of executions) S, we call an execution E *legal* under semantics S if it belongs to S. A well-formed TM semantics should not impose restrictions not related to the placement of transactional actions in the execution order. In particular, it should contain all the executions in which transactional actions do not interleave with actions of

[4]with the exception of programs that contain low-level atomic variables

other threads. It should also contain no executions with a data race between two transactional actions. In the rest of this paper we consider only TM semantics that satisfy these properties.

C++ assigns well-defined behavior only to race-free programs. It is therefore important to define what programs should be considered race-free. We say that a program P is race-free under semantics S if all executions of P legal under S are race-free. Otherwise, a program is racy. Note that in our framework, the definition of race-freedom makes no sense without a semantics, as a racy execution counts toward determination of the program race-freedom only if it is legal under the given semantics.

TM semantics thus defines both a set of executions that a programmer can use to reason about the program behavior and a set of execution that a programmer can use to reason about the program race-freedom. A program that is race-free under a particular semantics should behave according to one of its executions legal under that semantics. A program that is racy under a particular semantics has undefined behavior.

We say that two executions E and E' of a program P are equivalent if (1) $actions(E) = actions(E')$, (2) for any two actions A and B such that $A <_E B$, either $A <_{E'} B$ or A and B are non-conflicting actions executed by different threads, and (3) for any two actions A and B such that $A <_{E'} B$, either $A <_E B$ or A and B are non-conflicting actions executed by different threads. It is easy to see that equivalent executions of a race-free program result in the same visible behavior. Each read of a variable sees the same value. This value is determined by the last write to the variable in the execution order, which is the same because the equivalence preserves the order of conflicting actions. The final state of the memory is also the same, as for each memory location this state is determined by the last write to that location, which is again the same because equivalence does not reorder writes to the same memory location.

In the rest of this paper we use the following lemma to reason about the equivalency of program executions:

LEMMA 1. *Let E be a race-free execution of the form (P, F, A, S), where P, F and S are arbitrary sequences of actions (P and S might be empty), A is a single action such that no action in F is executed by the same thread that executed A and either 1) A is not a synchronization action or 2) F contains no synchronization actions that conflict with A. Then, execution $E' = (P, A, F, S)$ is equivalent to E.*

PROOF. Let $F = (F_1 \dots F_n)$. We first observe that F_n and A cannot conflict. If F_n and A are data accesses then they cannot conflict because E is race-free. If F_n and A are synchronization actions then they cannot conflict by the preposition of the lemma. Finally, if only one of F_n and A is a synchronization action, they cannot conflict by definition of conflicting actions.

Let $E'' = (P, F_1 \dots F_{n-1}, A, F_n, S)$. E'' and E are equivalent because they differ only by the order of F_n and A, which do not conflict. We now prove the lemma by induction on length of F. If $n == 1$ then $E'' == E'$, so E' is equivalent to E. Assume that the lemma holds for any length of F that is less than n. Then E' is equivalent to E'', and thus is equivalent to E. □

Two TM semantics S_1 and S_2 are equivalent for a class of programs C if: (1) Any program in C that is race-free under S_1 is race-free under S_2; (2) Any program in C that is race-free under S_2 is race-free under S_1; (3) For any race-free program P in C, any execution of P that is legal under S_1 is either legal under S_2 or is equivalent to an execution legal under S_2. (4) For any race-free program P in C, any execution of P that is legal under S_2 is either legal under S_1 or is equivalent to an execution legal under S_1.

We now formally define what we mean by atomicity and serializability:

- In a given execution, a transaction is *atomic* if its actions do not interleave with actions of other threads. Formally, a transaction T is *atomic* in execution E if any action A such that $T_S <_E A <_E T_C$ is executed by the same thread that executed T ($thread(T) = thread(A)$). An execution is atomic if all transactions in that execution are atomic.

- In a given execution, a transaction is *serializable* if its actions do not interleave with actions of other transactions. Formally, transaction T is serializable in execution E if any action A such that $T_S <_E A <_E T_C$ is either executed by the same thread that executed T or is a non-transactional action. An execution is serializable if all transactions in that execution are serializable. An atomic execution is serializable but the reverse is not true.

We say that a semantics provides atomicity for a given program if any execution of that program legal under the semantics is either atomic or is equivalent to an atomic execution. Similarly, a semantics provides serializability for a given program if any execution of that program legal under the semantics is either serializable or is equivalent to a serializable execution.

Our framework allows us to give precise definitions to many popular TM semantics. For example, we can describe strong atomicity as a semantics that contains only atomic executions (and, consequently, guarantees atomicity for all programs) and SLA as a semantics that contains only serializable executions (and thus guarantees serializability for all programs). Note that these semantics differ not only by the set of their legal executions but also by their definition of the race-freedom. Figure 1, for example, shows a program that is racy under SLA but is race-free under strong atomicity. In this program a data race occurs only if the write to `x` interleaves with the actions of the transaction in Thread 1 that reads `x`. Under strong atomicity such interleaving is impossible, so this program is race-free.

Note that our framework is designed to reason about properties of TM semantics rather than implementations. Our work is thus complementary to the work on correctness criteria for TM implementations [15] that already assumes a particular TM semantics (namely, strong atomicity).

3. LOCK-BASED SEMANTICS

Many proposed TM semantics express the behavior of transactions via an analogy with locks. Under Single Lock Atomicity (SLA) [20], for example, transactions execute as if protected by a single program-wide lock. Under Disjoint Lock Atomicity (DLA) [26], transactions execute as if protected by some minimal set of locks such that two transactions share a common lock if and only if they conflict.

This definition matches an intuition that transactions without conflicting accesses to shared memory can be executed in parallel. Weaker memory models with more complex lock analogies have also been proposed [26]. At the same time, lock-based semantics have been criticized for reducing the transactional semantics to that of lock-based synchronization [22, 32].

We believe that the problem lies with lock-based formulations of the semantics rather than semantics themselves. Both SLA and DLA capture useful properties of transactional executions that can and should be expressed without referring to locks.

In this section, we use our framework to give precise definitions to SLA and DLA and to analyze their properties. While past work has discussed the relative benefits of SLA and DLA in the context of Java [19, 26], in this paper, we compare SLA and DLA in the context of C++, which defines semantics only for race-free programs. We do not consider the semantics weaker than DLA (e.g., ALA and ELA [26]) because these semantics have been proposed solely to eliminate STM implementation constraints specific to Java. To avoid confusion, we keep the original names of SLA and DLA even though our definitions no longer refer to locks.

In our framework, SLA can be formulated simply as a semantics that contains only serializable executions. This definition captures the essence of SLA — if transactions were protected by the same lock their actions would not interleave in the total order of actions used to explain the program behavior. DLA imposes weaker restrictions: Conflicting transactions appear to execute serially with respect to each other, and the actions of non-conflicting transactions are allowed to interleave. More formally, DLA can be defined as a semantics that contains all executions E that satisfy the following property: if A and B are conflicting transactional actions in E such that A executes within a transaction T, B executes within a transaction T', and $A <_E B$, then $T_C <_E T'_S$. Note, that DLA is a weaker semantics than SLA; that is, any execution legal under SLA is also legal under DLA but the reverse is not true.

For C++ programs without nested synchronization, choosing between SLA and DLA is not necessary because in this domain these semantics are equivalent. Moreover, both SLA and DLA provide atomicity for race-free programs without nested synchronization. The following theorems formalize these claims.

THEOREM 1. *DLA provides atomicity for race-free programs without nested synchronization.*

PROOF. Consider a program without nested synchronization that is race-free under DLA. Let E be a non-atomic execution of this program that is legal under DLA. Because the program is race-free under DLA, the execution E is race-free. Let k be the length of the longest atomic prefix in E. That is, A_{k+1} is the first action that interleaves with some transaction T. We shall show that E has an equivalent execution with an atomic prefix of length $k+1$. It then follows by induction that E also has a full equivalent atomic execution.

Let $E = (P, T_P, A_{k+1}, S)$, where P and S are some sequences of actions (possibly empty) and T_P is the sequence of actions in the transaction T that execute before A_{k+1}. Consider execution $E' = (P, A_{k+1}, T_P, S)$ that permutes A_{k+1} and T_P. Execution E' has an atomic prefix of length $k+1$. By Lemma 1 it is also equivalent to execution E because in programs without nested synchronization T_P cannot contain synchronization actions. □

THEOREM 2. *SLA and DLA are equivalent for C++ programs without nested synchronization.*

PROOF. The proof directly follows from Claims 2.1- 2.4 listed below and from the definitions of program equivalence and program race-freedom. □

CLAIM 2.1. *Any execution legal under SLA is legal under DLA.*

PROOF. This follows directly from the definitions of SLA and DLA. □

CLAIM 2.2. *For programs without nested synchronization that are race-free under DLA, any execution legal under DLA is either legal under SLA or is equivalent to an execution legal under SLA.*

PROOF. This follows from Theorem 1, as any atomic execution is legal under SLA. □

CLAIM 2.3. *A program that is race-free under DLA is race-free under SLA.*

PROOF. Consider a program that is race-free under DLA. Any execution of this program that is legal under SLA is also legal under DLA (Claim 2.1) and thus is race-free. Consequently, the program is race-free under SLA. □

CLAIM 2.4. *A program that contains no nested synchronization and is race-free under SLA is race-free under DLA.*

PROOF. We prove this property by contradiction. Assume that there exists a program that contains no nested synchronization, is race-free under SLA and is not race-free under DLA. This program should have a racy execution E that is legal under DLA and is not legal under SLA. Let A_k and A_{k+1} be the first actions in E that form a data race. Let P be the prefix of E of length $k-1$. That is, $E = (P, A_k, A_{k+1}, S)$. Prefix P is race-free as A_k and A_{k+1} are the first actions that form a data race. By Claim 2.2 P has an equivalent prefix P' that is legal under SLA. Let us add actions A_k and A_{k+1} to P'. Since A_k or A_{k+1} are conflicting data accesses, neither is a transaction start action. Thus, the resulting partial execution $E'_P = (P', A_k, A_{k+1})$ is legal under SLA. Let us complete E'_P in any way allowed by SLA. We now constructed a racy execution that is legal under SLA. This contradicts our assumption that the program is race-free under SLA. □

THEOREM 3. *SLA provides atomicity for race-free programs without nested synchronization.*

PROOF. This theorem follows directly from Claim 2.1 and Theorem 1. □

For race-free programs with nested synchronization, neither SLA nor DLA guarantee atomicity (as the example in Figure 2 demonstrates). Furthermore, in the presence of nested synchronization, DLA is strictly weaker than SLA and does not guarantee even serializability. Figure 3 shows an example of a race-free program with nested synchronization that may behave differently under SLA and DLA. [5]

[5]This figure shows a modified version of publication by empty transaction from [26].

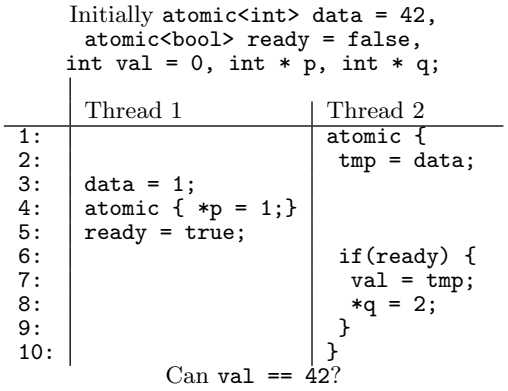

Figure 3: DLA does not guarantee serializability in race-free programs.

This program is race-free under both SLA and DLA semantics because it performs all non-transactional accesses using C++ atomic variables. Thread 2 can see $val == 42$ only if the transaction in Thread 1 interleaves with the execution of the transaction in Thread 2. Under SLA, transactions cannot interleave, so the result $val == 42$ is illegal. Under DLA, the interleaving shown in Figure 3 is legal if the transactions in Threads 1 and 2 do not conflict. This depends on whether pointers p and q alias, which in the general case is undecidable. DLA thus not only allows non-serializable executions, but also prevents clear reasoning about program behavior.

DLA also does not provide significant advantages over SLA from the point of view of the implementation. In the absence of nested synchronization, DLA cannot provide any advantages over SLA because in this case these semantics are equivalent. In the presence of nested synchronization, the advantages of DLA are marginal because both SLA and DLA effectively require serialization of transactions on a lock release or synchronization variable write operation. (Appendix A explains this point in more detail.) We thus believe that DLA is not a practical choice for TM semantics in the context of C++.

4. RACE-FREE ATOMICITY

Atomicity is a fundamental property that helps programmers reason about concurrent programs. In general, locks do not guarantee atomicity. However, previous work [13] presented convincing evidence that most of the time programmers use locks to express atomicity rather than mutual exclusion. Several researchers have recently focused on finding atomicity violations due to incorrect use of locks in programs [21, 14]. TM alleviates this problem by providing a high-level abstraction that allows programmers to express the intent of atomicity explicitly.

Strong atomicity guarantees atomicity for all programs, (including programs considered racy by SLA) but is not practical for C++. Implementing strong atomicity in software introduces overheads such as software instrumentation on non-transactional code [30]. This violates the "pay-as-you-go" principle of not imposing overhead on code that does not use transactions, an important practical consideration for C++. Although optimization techniques for strong atomicity have been demonstrated for Java [30, 29, 7] and C# [2], where readily available type information and dynamic compilation or patching allow for aggressive elimination of software instrumentation, the viability of strong atomicity has yet to be demonstrated for C++. Hardware support for strong atomicity has been proposed [27, 3], but such hardware is not available. We believe that strong atomicity is likely to remain unattainable for C++.

Providing strong atomicity for C++ is not only impractical but is also overkill because it prohibits behaviors than the standard C++ memory model would seem to allow (i.e., programs that would appear racy under lock-based semantics in C++ and, consequently, would be allowed arbitrary behaviors, would be considered race-free under strong atomicity). On the other hand, weaker semantics such as SLA do not provide atomicity for race-free programs. SLA does guarantee atomicity for race-free programs without nested synchronization, but we do not consider prohibiting locks inside transactions a good design choice. Such a restriction would prevent reuse or incremental refactoring of existing lock-based code and is likely to slow down the adoption of TM.

Another option is to treat conflicts between transactional and non-transactional synchronization actions as data races, as implied by the work on TM semantics that does not provide special treatment for synchronization variables and locks [1, 10]. We consider this design choice also unacceptable, as it allows for a possibility of creating a data race by adding transactions to a correctly synchronized program. For example, under such an interpretation of a data race, the program in Figure 2 is racy and may have an arbitrary behavior, even though without a transaction in Thread 1 this program is race-free.

We thus propose Race-Free Atomicity (RFA), a new semantics that guarantees atomicity for all race-free programs without imposing unnecessary restrictions on STM implementations. RFA is stronger than SLA because it provides atomicity for race-free programs even in presence of nested synchronization. It is weaker than strong atomicity in that it accepts fewer executions as race-free but, unlike strong atomicity, can be implemented in practice. Recent work [34] has suggested that TM implementations could provide atomicity in the presence of locks by holding the locks acquired inside a transaction until that transaction either commits or aborts. RFA captures this intuition by prohibiting transactions from interleaving with conflicting synchronization actions of other threads. More formally, we define RFA as a semantics that contains all executions E that satisfy the following property: for any transaction T and action A such that $T_S <_E A <_E T_C$, A is executed by the same thread that executed T ($thread(A) == thread(T)$), or is a non-transactional data access, or is a non-transactional synchronization action that does not conflict with T.

The following theorems state the properties of RFA and its relationship to SLA.

THEOREM 4. *RFA provides atomicity for race-free programs.*

PROOF. We shall show that any race-free execution that is legal under RFA is either atomic or has an equivalent atomic execution. Consider a non-atomic race-free execution E that is legal under RFA. Let k be the length of the longest atomic prefix in E. That is, A_{k+1} is the first action that interleaves

with some transaction T. We shall show that E has an equivalent execution with an atomic prefix of length $k+1$. It then follows by induction that E also has a full equivalent atomic execution.

Let $E = (P, T_P, A_{k+1}, S)$, where P and S are some sequences of actions (possibly empty) and T_P is the sequence of actions in T that execute before A_{k+1}. Since RFA disallows interleaving of transactions with conflicting synchronization actions, A_{k+1} is not a synchronization action conflicting with T_P. We thus can apply Lemma 1 to obtain an equivalent execution $E' = (P, A_{k+1}, T_P, S)$ that has an atomic prefix of length $k+1$. □

THEOREM 5. *RFA and SLA are equivalent for C++ programs without nested synchronization.*

PROOF. The proof of this theorem directly follows from Claims 5.1- 5.4 listed below and the definitions of program equivalence and program race-freedom. □

CLAIM 5.1. *Any execution legal under RFA is legal under SLA.*

PROOF. This follows directly from the definitions of RFA and SLA. □

CLAIM 5.2. *For programs without nested synchronization that are race-free under SLA, any execution legal under SLA is either legal under RFA or is equivalent to an execution legal under RFA.*

PROOF. This follows from Theorem 3, as any atomic execution is legal under SLA. □

CLAIM 5.3. *A program that is race-free under SLA is race-free under RFA.*

PROOF. Consider a program that is race-free under SLA. Any execution of this program that is legal under RFA is also legal under SLA (Claim 5.1) and thus is race-free. Consequently, the program is race-free under RFA. □

CLAIM 5.4. *A program that contains no nested synchronization and is race-free under RFA is race-free under SLA.*

PROOF. The proof is similar to that of Claim 2.4. Assume that there exists a program that contains no nested synchronization, is race-free under RFA but is not race-free under SLA. Let E be a racy execution of this program that is legal under SLA and is not legal under RFA. Let E be of form $E = (P, A_k, A_{k+1}, S)$, where A_k and A_{k+1} are the first actions that form a data race. Prefix P is race-free and by Claim 5.2 has an equivalent prefix P' that is legal under RFA. Let us add actions A_k and A_{k+1} to the prefix P'. Since A_k and A_{k+1} are conflicting data accesses, neither of them can be a synchronization action or a transaction start action. Thus the resulting partial execution $E'_P = (P', A_k, A_{k+1})$ is also legal under RFA. Completing this prefix in any way allowed by RFA constructs a racy execution that is legal under RFA, and, thus, proves that our assumption was invalid. □

Although RFA is weaker than strong atomicity, it still cannot be practically implemented in an STM when transactions contain nested synchronization hidden inside legacy code. Section 6 describes our proposal for TM semantics that deals with this limitation.

Initially `int x = 0`

	Thread 1	Thread 2
1:	`atomic {`	
2:	`x++;`	
3:		`t = x;`
4:	`abort;`	
5:	`}`	

Figure 4: Is this program race-free?

Initially `int x = 0`

	Thread 1	Thread 2
1:	`atomic {`	
2:	`t1 = x;`	
3:		`x = 1;`
4:	`t2 = x;`	
5:	`if (t1 != t2)`	
5:	`abort;`	
6:	`}`	

Figure 5: Can this transaction abort?

5. USER-LEVEL ABORT

Many TM systems provide some variant of a user-level abort construct such as abort or retry. Intuitively, aborted transactions should behave as if they never executed. We considered interpreting such an intuition literally and assigning user-level abort an *invisible abort semantics*, that is, modeling user-level abort in our framework by excluding the actions of aborted transactions from program executions. Such definition of user-level abort, however, would preclude the actions of aborted transactions from participating in determination of program race-freedom, and would require TM implementations to provide strong atomicity for programs that may contain user-level abort.

Figure 4 shows a simple example that illustrates the consequences of invisible abort semantics. Under strong atomicity, this program clearly is race-free, but under RFA and weaker semantics, this program appears to have a data race on `x`. Under invisible abort semantics, the write to `x` in Thread 1's aborted transaction is not a part of any program execution, and thus cannot form a data race with the read of `x`, which implies this program is race-free under all our semantics. Moreover, it implies the result `t == 1` is illegal, because the write that could produce this result is not a part of a program execution. Invisible abort semantics thus precludes in-place update STM implementations that do not provide strong atomicity, as such implementations cannot prevent non-transactional reads from observing results of aborted transactions.

The example in Figure 5 shows that invisible abort also has consequences for write-buffering STMs. It also illustrates how invisible abort complicates reasoning about program behavior. Under invisible abort semantics, the result `t1 != t2` is illegal because it leads to circular reasoning. Assume that `t1 != t2`. This can happen only if this program has a data race on `x`. In this case, Thread 1's transaction aborts, so the reads of `x` are not a part of a program execution, and they could not cause a data race. This program is therefore race-free, but this implies that `t1 != t2` is impossible. The only way to avoid this circular argument is to prohibit an execution where `t1 != t2`, that is, effectively guarantee strong atomicity for Thread 1's transaction.

Requiring TM implementations to provide strong atomicity for programs with user-level abort is impractical even if user-level abort is rare. Without whole-program analysis, it is not possible to statically determine if non-transactional code may be executed concurrently with a transaction containing user-level abort. Consequently, such a requirement would cause any software TM implementation that does not support whole-program analysis to impose the overhead of software instrumentation on all non-transactional code.

We thus believe that user-level abort should instead be given *visible abort* semantics, where aborted transactions are part of the program execution and contribute to determination of the program race-freedom. In this case, user-level abort is simply an undo action that rolls back the side effects of the aborted transaction. That is, it restores the state of the memory locations written by the transaction to the values they had at the time the transaction wrote them and releases the locks acquired by the transaction. Under such an interpretation of user-level abort, the programs in Figures 4 and 5 are racy (under RFA and weaker models) and, thus, can be handled in a straightforward way by an STM. Note that for atomic executions of race-free programs, such restricted interpretation of user-level abort still guarantees that an aborted transaction behaves as if it never executed, as the side effects of a transaction are rolled back before they can be observed by an action of another thread. [6]

6. OUR PROPOSAL FOR C++ TM SEMANTICS AND LANGUAGE CONSTRUCTS

By itself, neither SLA nor RFA is a perfect candidate for C++ TM semantics. SLA does not guarantee atomicity for all race-free programs. RFA guarantees atomicity but cannot be practically implemented without restricting what actions can be executed transactionally. In particular, STM implementations cannot practically provide RFA in the presence of nested synchronization hidden in legacy code. We therefore make a radical departure from the previous proposals for C++ language-level constructs [9, 28] and propose a dual TM semantics with two language-level constructs: *atomic statements* and *critical statements* (which we denote using the `tm_critical` keyword). Furthermore, we define these constructs so that they interoperate.

In race-free programs, atomic statements appear to execute as single indivisible actions; that is, they behave according to RFA semantics. Atomic statements, however, cannot contain arbitrary actions, such as calls to legacy code. Critical statements appear to execute in some serial order with respect to other critical and atomic statements; that is, they behave according to SLA semantics. Critical statements may contain arbitrary code. However, critical statements that contain nested synchronization and I/O might appear to execute non-atomically. More formally, we define our dual semantics as a semantics that contains all serializable executions E such that for any transaction T in E that corresponds to an atomic statement, the actions of T do not interleave with conflicting synchronization actions of other threads in the execution order of E.

The operations inside an atomic or critical statement can be rolled back explicitly via a user-level abort statement. An abort statement undoes the writes executed by the atomic or critical statement that statically encloses it and transfers control to the statement immediately following that statement. In particular, it restores the state of the memory locations written by the transaction to the values they had at the point the transaction initially wrote to those locations.

[6]We leave to future work a more rigorous treatment of abort semantics and the proof of this statement.

Initially `atomic<int> v = 0, int x = 0`

	Thread 1	Thread 2
1:	`tm_critical {`	
2:	`x = 1;`	
3:	`v = 1;`	
4:		`if (v == 1) {`
5:		`t1 = x;`
6:	`tm_abort;`	
7:	`}`	
8:		`t2 = x;`
9:		`}`

Can `t1 != t2`?

Figure 6: User abort and irrevocable actions.

Some actions cannot be undone via user-level abort. Examples of such actions include legacy code, non-transactional I/O and, in general, any action that exposes the internal state of a transaction to another thread or the external environment. Figure 6 illustrates the last scenario. In this example, the critical statement in Thread 1 publishes the value in `x` by setting a value in the synchronization variable `v`. Thread 2 then observes this value in `v` (synchronizing with Thread 1) and then reads the value in `x`. If Thread 1 now aborts, it will undo its writes to `v` and `x`, and create a race with the second read of `x` in Thread 2. Moreover, the program is left in an inconsistent state because Thread 2 observed a speculatively written value from Thread 1. Clearly Thread 1 should not be allowed to abort after it violates atomicity.

We call the actions that cannot be undone via user-level abort *irrevocable actions*. In our semantics, atomic statements cannot contain irrevocable actions. They thus have no restrictions on placement of user-level abort. Critical statements may contain irrevocable actions. However, an attempt to execute an abort statement after executing an irrevocable action will result in in a run-time failure. (An alternative design choice is to simply prohibit user-level abort in critical statements. However, this would also prevent programmers from aborting critical statements that dynamically executed no irrevocable actions.)

Our design leaves some room for the choice of *unsafe actions*, that is, the actions that should be prohibited inside atomic statements. Conceptually, an action is unsafe if it is irrevocable or cannot be practically implemented to execute atomically. Calls to legacy libraries are obviously unsafe, as they are both irrevocable and cannot be executed atomically in an STM. We believe that operations on C++ atomic variables should also be declared unsafe. Executing these operations atomically would require software instrumentation on non-transactional accesses to atomic variables. This might be unacceptable given that C++ atomic variables are intended as a mechanism for implementing concurrent algorithms that is more efficient than locks. [7] In an ultimate push for simplicity, we could also prohibit locks inside atomic

[7]The examples in this paper use C++ atomic variables for the sake of brevity. The same concepts can be illustrated by using locks instead of C++ atomics.

statements. This would allow one to describe the behavior of both critical and atomic statements by SLA semantics, as without nested synchronization RFA and SLA are equivalent. Such restriction, however, would limit composition of locks and transactions, and is unnecessary, as STM implementations can provide atomicity for locks [34].

There exist several options for enforcing safety restrictions on actions of atomic statements. First, we can prohibit unsafe code within atomic statements statically via rules that the compiler can check automatically: an atomic statement cannot contain in its lexical scope any unsafe code or any calls to functions that may contain unsafe code. The latter part of this rule requires interprocedural analysis, which we do not believe is a reasonable requirement for a C++ language construct. Alternatively, we can prohibit unsafe code within atomic statements dynamically. In this approach, an attempt to execute unsafe code inside an atomic statement would cause the atomic statement to abort and throw a runtime exception. This still guarantees atomicity but increases the burden on the programmer to be careful in avoiding unsafe code inside atomic statements. The third, and the most practical option, is to introduce a function annotation that explicitly marks a function as safe (e.g., `tm_safe`). The compiler then statically enforces that neither atomic statements nor `tm_safe` functions contain calls to functions that are not `tm_safe`. To allow indirect function calls inside atomic statements and `tm_safe` functions, the `tm_safe` annotation can also be used as a qualifier on a function pointer type.

7. CONCLUSIONS

Integrating transactional memory into mainstream programming languages such as C++ requires precise definition of its semantics in the context of these languages. These semantics should provide the essential database transaction properties that have for decades proven themselves useful in the database world. At the same time, these semantics must consider the unique interactions that arise from integrating transactions into an existing programming language, such as the interaction between transactions and existing synchronization primitives, irrevocable actions and legacy code.

In this paper we have addressed the semantics of TM in the context of the C++ language. We have given a precise formulation of several TM semantics and related them to the properties of atomicity and serializability. We have shown how several of these semantics are equivalent for C++ programs that perform no synchronization inside transactions. We have introduced a new semantics called RFA that guarantees atomicity for programs with no nested synchronization. Finally, we have proposed new C++ language constructs that allow the programmer to specify explicitly whether transactions should guarantee atomicity. With the language constructs and semantics presented in this paper, we now have a solid foundation for introducing first-class transactional memory constructs into C++.

8. REFERENCES

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

[2] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP 2009*.

[3] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strong-atomic hybrid transactional memory. In *ISCA 2008*.

[4] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2005.

[5] H. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *to HotPar 2009, to appear*.

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

[7] N. Bronson, C. Kozyrakis, and K. Olukotun. Feedback-directed barrier optimization in a strongly isolated STM. In *POPL 2009*.

[8] C++ Standards Committee, Pete Becker, ed. Working draft, standard for programming language C++. C++ standards committee paper WG21/N2857=PL22.16/09-0047, March 2009. `http://www.open-std.org/JTC1/ SC22/WG21/docs/papers/2009/n2857.pdf`.

[9] L. Crowl, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Integrating transactional memory into c++. In *The Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[10] L. Dalessandro and M. L. Scott. Strong isolation is a weak idea. In *The Fourth ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[11] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS 2009*.

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

[13] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL 2004*.

[14] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs.

[15] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP 2008*.

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

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

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

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

[20] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[21] S. Lu, T. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *ASPLOS 2006*.

[22] V. Luchangco. Brief announcement: Against lock-based semantics for transactional memory. In *SPAA 2008*.

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

[24] V. J. Marathe, W. N. Scherer, III, and M. L. Scott. Adaptive software transactional memory. In *International Symposium on Distributed Computing 2005*.

[25] V. J. Marathe, W. N. Scherer, III, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *LCR 2004: Languages, Compilers, and Run-time Support for Scalable Systems*.

[26] 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*.

[27] C. C. Minh, M. Trautman, 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*.

[28] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA 2008*.

[29] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA 2008*.

[30] 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*.

[31] M. Spear, M. Michael, and C. von Praun. Ringstm: Scalable transactions with a single atomic instruction. In *SPAA 2008*.

[32] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-based semantics for software transactional memory. In *OPODIS 2008*.

[33] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Brief announcement: Privatization techniques for software transactional memory. In *PODC 2007*.

[34] H. Volos, N. Goyal, and M. Swift. Pathological interactions of locks with transactional memory. In *The Third ACM SIGPLAN Workshop on Transactional Computing*, 2008.

APPENDIX

A. SLA VS. DLA

In Section 3 we claimed that DLA provides only marginal advantages over SLA from the implementation point of view. We substantiate this claim using the example in Figure 7. In this program, Thread 2 privatizes `data` by setting flag `shared` to zero and hands off the data to Thread 3 by setting atomic variable `v` to `1`. Thread 1 modifies `data` only if it is shared. The example is somewhat contrived - Thread 2 first hands off `data` to Thread 3 and then makes it private. However, technically it is correct under either SLA or DLA. Transactions in Threads 1 and 2 conflict so they cannot interleave even under DLA. Thread 3 reads `data` only if it executes after the start of Thread 2's transaction (otherwise, `v == 0` and `data` is not read). Thread 1's transaction writes `data` only if it executes before the transaction in Thread 2.

Initially `atomic<int> v == 0, int shared == 1 int data == 0`

	Thread 1	Thread 2	Thread 3
1:		atomic {	
2:		v = 1;	
3:			if (v == 1) {
4:			t1 = data;
5:	atomic {		
6:	if (shared)		
7:	data = 42;		
8:	}		
9:			t2 = data;
10:			}
11:		shared = 0;	
12:		}	

Can `t1 != t2`?

Figure 7: Transitive privatization via nested synchronization

Consequently, this program is race-free and the result `t1 != t2` is impossible. Thread 3 should either see all effects of the transaction in Thread 1 or none.

The read of `v` in Thread 3 effectively acts as an atomicity break point for Thread 2's transaction. More precisely, we say that a non-transactional lock acquire (or synchronization variable read) is *atomicity break point* for transaction T if it acquires a lock released by T (or reads the value written in T) and executes before the commit of T. Under either SLA or DLA, any transaction T' that conflicts with T and did not complete before T's atomicity break point P should appear to execute after T. (It cannot appear to execute before T because in that case its side effects should have been visible to non-transactional code that executes after P.) From the implementation point of view, this means that T' cannot commit between T's atomicity violation point P and T's linearization point L. (For write-buffering STMs, linearization point is validation point; for in-place update STMs it is commit point.) In in-place update STMs, T' also cannot abort between P and L, so it cannot execute between P and L at all. (If T' aborts between P and L the effects of its speculative execution might be visible to non-transactional code after P.)

Under DLA T' can commit between P and L if it does not conflict with T. However, in general it is impossible to determine if two transactions conflict till both of them have completed. Thus, DLA offers no practical advantages over SLA with respect to handling nested lock release operations and synchronization variable reads (except in limited situations, such as when T' is empty).

DLA gives more flexibility to STM implementations with respect to nested lock acquire and synchronization variable reads. For example it allows the execution in Figure 3 that is illegal under SLA. However, we believe this to be a marginal advantage as locks are more frequent than synchronization variables and lock acquire is unlikely to occur in a transaction without an accompanying lock release.