Dynamic Separation for Transactional Memory

Martín Abadi[†⋆] Andrew Birrell[†] Tim Harris[‡] Johnson Hsieh[†] Michael Isard[†]

Microsoft Research, Silicon Valley[†] University of California, Santa Cruz[⋆] Microsoft Research, Cambridge[‡]

abadi@microsoft.com birrell@microsoft.com tharris@microsoft.com johnsonhsieh1@hotmail.com misard@microsoft.com

Abstract

Implementations of language constructs over transactional memory have typically provided unexpected semantics, required the re-compilation of non-transacted code, or assumed new hardware. We introduce an alternative approach founded on a contract between the programmer and the language implementation in which strong semantics are provided to programs that are "correctly synchronized" in their use of the language, even if the underlying TM implementation provides weaker guarantees.

Our approach is based on the dynamic separation of objects that can be updated in transactions, objects that can be updated outside transactions, and read-only objects that are accessible everywhere. We introduce explicit operations that, at run-time, identify transitions between these modes of access. Dynamic separation is more flexible than earlier notions of static separation, while still permitting an extremely wide range of hardware-based and software-based implementations. We define what it means for a program to obey the dynamic-separation discipline, and we show how a runtime checking tool—analogous to a data-race detector—can test this property. We also describe our design and implementation of a system with dynamic separation, and examine the use of dynamic separation in an asynchronous IO library.

1. Introduction

Recently there has been much work on implementing language-level atomic blocks over software transactional memory (STM). This approach provides an alternative to using locks and condition variables for shared-memory concurrency. Typical STM implementations allow threads to execute non-conflicting transactions in parallel, enabling data structures which scale with performance similar to that of complicated fine-grained locking, with the programming simplicity of coarse-grained locking.

Much recent transactional-memory research has focused on the language constructs that are exposed to the programmer [9, 10, 13, 21, 22] and the semantics that an implementation of these constructs must obey [16, 1]. The interaction between program fragments running inside transactions and those running concurrently in non-transacted code has been found to be particularly subtle [4, 22, 16, 1]. In this paper we address one aspect of this interaction that arises when an object is accessible by both transacted and non-transacted code. We are particularly motivated by the goal of portability: to facilitate early adoption of transactional memory it is important that programs run efficiently, with identical semantics, over a range of current STM implementations, and that they continue to run, with the same semantics and without unnecessary overheads, once hardware transactional memory becomes widely available.

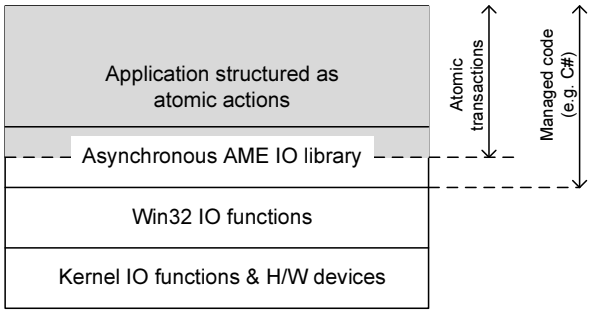

Figure 1. System design: an AME application is structured as atomic actions which execute over an asynchronous IO library. The IO library interfaces between the atomic actions used by the application and the ordinary direct execution of the underlying layers.

We introduce a new technique called "dynamic separation" (DS) whereby each object has a "protection mode" indicating whether it can be accessed inside or outside a transaction, and the programmer explicitly indicates when this mode should be changed. We show that this is more flexible for the programmer than previous notions of static separation [10, 1, 16], and permits more flexibility for the language implementor than weaker notions like "violation-freedom" [1]. We implement DS within the AME programming model [13]. In this setting, where all code runs inside a transaction by default and non-transacted code is explicitly delimited by the programmer, there is a natural symmetry between the DS constructs and familiar existing notions such as "pinning" a managed object before passing it to native code.

Figure 1 illustrates the structure of an application built using AME [13]. Rather than using threading with occa-

```
// Atomic action A1
if (status == Idle) {
  <populate buffer b>

  // IO library
  status = Requested;
  unprotected {
    // Unprotected code U1
    // unprotect(b); // DS1
    pin(b);
    unsafe {
      syscall(&b[0]);
    }
    unpin(b);
    // protect(b);   // DS2
  }
  status = Complete;
}
```

```
// Atomic action A2
blockuntil(status == Complete);
<use results from buffer>


// Atomic action A3
if (status == Idle) {

  <populate buffer b>
  status = Requested;
  ...
}
```

Figure 2. Example (pseudo-code).

sional `atomic` blocks, an application is structured as a set of atomic actions. AME provides constructs for delimiting the boundary between one atomic action and the next, for blocking an atomic action until it is ready to run, and for executing an asynchronous call as a new atomic action. We call these atomic actions "protected" code and, inverting the normal way of programming with transactions, an AME program may contain explicit `unprotected` blocks which execute directly, finishing the atomic action before them, and starting a new atomic action after them. The atomic actions in the application interact with the external world through a layer built using `unprotected` blocks that serve as an interface with external libraries and resources. We review the constructs involved in Section 1.1.

We aim to run programs with "strong semantics" meaning that (i) atomic actions appear genuinely atomic, without interleaving other atomic actions or ordinary code ("strong atomicity" [4]), and (ii) operations appear to execute in the same order as specified in the source code, without impact from compiler transformations or relaxed memory models. This semantics is notoriously difficult in settings like that of Figure 1: existing solutions either require hardware extensions, or they require the implementation of non-atomic code to be aware of the transactional machinery (a non-starter when DMA transfers from devices are considered in modern IO libraries).

Figure 2 shows an example to illustrate the problems we have in mind. An atomic action `A1` sets up a buffer of data to be output. The IO library finishes the atomic action by entering an `unprotected` block. The buffer is then "pinned" in memory—that is, the garbage collector is prevented from relocating or reclaiming it, letting the object be accessed within the system call. Concurrently, a second atomic action `A2` blocks until a status flag shows that the request is complete whereupon it accesses results from the buffer. A third atomic action `A3` conflicts with `A1`; it changes a flag that `A1`'s control flow depends on, and it tries to use the buffer `b` for its own purpose.

In this example we must make sure that accesses in `U1` see the preceding updates made from `A1` and that, once `A2` is allowed to run, it sees any updates made to `b` by the system call. We must also consider concurrency introduced by the implementation of atomic actions over STM. For example, suppose that `A3` executes concurrently with `A1` and tries to prepare an alternative request using the buffer. With some STMs `A3` may continue to make updates to `b` before the conflict with `A1` is detected. This could corrupt the data while it is being accessed by the system call.

The basic idea of DS is to extend the programming model with explicit operations for indicating, at run-time, which data should be accessed by atomic actions, which data should be accessed directly, and which data is immutable and can be shared between those access modes. The only changes needed to use DS in our example are to uncomment the lines DS1 and DS2: the buffer is marked "unprotected" during the time that it is accessed directly. We introduce these `protect`/`unprotect`/`share` operations and their semantics in Section 2, and discuss how we apply this notion to C# in Section 3.

DS can be seen as a contract between the programmer and the language implementor. So long as the program is correct in its use of DS—accessing only protected data in atomic actions, and accessing only unprotected data outside atomic actions, then the implementation must run the program with strong semantics. Crucially, the question of whether or not the program uses DS correctly is based on its hypothetical execution under strong semantics.

This form of contract benefits programmers by insulating them from implementation details like exactly which kind of TM implementation is used: correct programs run on all correct implementations. This is convenient in the short term, since it simplifies the task of learning to use transactional constructs; it may prove essential for long-term adoption of transactional memory, since without such a contract to support portability, code re-use and maintenance becomes very problematic. Such a contract also benefits the language implementor by providing a clear definition of which program transformations and implementation techniques are correct.

We have implemented AME with DS as an extension to the Bartok-STM system [11]. We discuss the implementation in detail in Section 4. However, the issues raised by the example of Figure 2 serve to illustrate the approach: we constrain the STM to make updates only to protected objects. This prevents it from trampling over data that has been passed to non-transacted code. There is an analogy here with using "pinning" to enable objects to be passed to native code: while pinned, an object will not be relocated in memory or reclaimed by the GC. Similarly, while unprotected, an object will not be modified by the TM implementation. In each case this isolates the non-GC or non-transacted code from the details of the GC or TM implementation.

We discuss how the programming language's memory model can incorporate DS (Section 4.2). We focus on the .NET 2.0 memory model, but briefly discuss aspects of the

Java memory model which allows a larger set of program transformations.

We sketch alternative implementations for different TMs (Section 4.3). In particular, our design for the semantics of DS avoids the overhead of tracking objects' protection status when building on an HTM with native support for strong atomicity. So long as it uses DS correctly, the same source program can run over this wide range of systems.

To help programmers use DS correctly, we support a debugging mode that adds run-time checks that all of a program's memory accesses are correct (Section 5).

As a case study, we examine DS in the context of applications built over the AME asynchronous IO library (Section 6). We show where DS operations are used in the library and how their implementation adds an imperceptible overhead in the applications. It adds less than 1% in a CPU-bound workload with comparable synchronization patterns. In contrast, if we require transacted and non-transacted applications to access statically-disjoint data, marshaling between these copies adds an order-of-magnitude slowdown.

We discuss related work in Section 7, contrasting DS with other disciplines such as notions of violation-freedom [1] or single global lock atomicity [15].

Finally, in Section 8, we conclude by discussing how DS could apply to languages like C# or Java augmented with atomic blocks.

In a companion paper [2], we develop a formal model of DS. We compare DS to other programming disciplines, formally, and study two models of implementations. In particular, we prove the correctness of a model based on our Bartok-STM approach.

1.1 Background: Automatic Mutual Exclusion

In this section we review the AME programming model [13, 1].

AME distinguishes "protected" code, which executes within transactions, from ordinary "unprotected" code. Importantly, the default is protected code. Unprotected code is supported primarily in order to permit interactions with legacy code, much as systems like the Java Virtual Machine (JVM) support native methods to interface between application code and system calls. Design choices that simplify protected code, or allow faster implementations of it, are often preferable even if they make writing unprotected code more complicated.

Running an AME program consists of executing a set of asynchronous method calls. The implementation guarantees that the program execution is equivalent to executing each of these calls (or their fragments, defined below) in some serial order. An asynchronous call is created by the invocation:

```
async MethodName(<method arguments>);
```

The caller continues immediately and, in the conceptual serialization of the program, the asynchronous callee will be executed after the caller has completed. AME achieves concurrency by executing asynchronous calls in transactions, overlapping the execution of multiple calls, with roll-backs when conflicts occur. If a transaction initiates other asynchronous method calls, their execution is deferred until the initiating transaction commits, and they are discarded if the initiating transaction aborts.

An asynchronous call may also make any number of calls to the system method `blockuntil(<predicate>)`. From the programmer's perspective, an asynchronous method can only complete if all these predicates evaluate to true. This behavior is like that of `retry` in some systems [10].

An asynchronous call may also invoke the system method `yield()`. A `yield` call breaks a method into multiple atomic fragments, implemented by committing one transaction and starting a new one. These atomic fragments are delimited dynamically by the calls of `yield`, not statically scoped like explicit atomic blocks. AME thus avoids some of the pitfalls of pure event-based programming models (in particular, "stack ripping"). With this addition, the overall execution of a program is a serialization of its atomic fragments.

To allow the use of legacy non-transacted code, AME provides block-structured `unprotected` sections. These must use existing mechanisms for synchronization. AME terminates the current atomic fragment before the code, and starts a new one after.

Many aspects of AME appear in other models. Transactional Coherence and Consistency (TCC) provides a programming model where all execution is divided into transactional units [8]. By construction this avoids the interactions between transacted and non-transacted code that we are concerned with here. Smaragdakis et al.'s Transactions with Isolation and Cooperation (TIC) model [21] provides non-block-structured atomic actions, several constructs for expressing roll-back operations, and different forms of transactional nesting. We focus on AME because it provides a smaller number of core constructs and because of the existing AME calculus [1]. DS readily applies to TIC and could be valuable in the scenarios Smaragdakis et al. have considered.

2. Dynamic Separation

In overview, DS works as follows:

- We distinguish dynamically between transacted ("protected") data, non-transacted ("unprotected") data, and read-only data. By default, data allocated inside a transaction is created in "protected" mode and data allocated outside a transaction is created in "unprotected" mode.
- We provide explicit operations to move data between these modes.
- We require that programs access data in the correct mode: read-only data may be read anywhere but not updated, protected data may be accessed freely inside transac-

tions, and unprotected data may be accessed freely outside transactions.

If the program obeys the dynamic-separation discipline, the language implementation is required to run it with strong semantics even if the underlying STM provides weaker guarantees.

This section develops this approach, exposing and resolving some subtleties in the details. We follow three goals which we use to motivate many of the design decisions:

- **The "fundamental property".** We want DS to be defined in terms of a program's execution under strong semantics and, if DS is used correctly, to require an implementation to run the program with strong semantics. This provides portability across TM implementations. This is analogous to the "fundamental property" of Saraswat et al. [19] in defining programming language memory models.
- **Compatibility with non-transacted memory accesses.** We want to avoid memory-access barriers outside atomic actions. Accordingly, we wish to assume only "weak atomicity" [4]. This makes transactions "pay-to-use". However, even with optimizations to remove redundant barriers (such as in [20]), we do not want to instrument the memory accesses within system calls like that in Figure 2, and we cannot instrument memory accesses made by devices.
- **TM implementation flexibility and parallelism.** We want to support a wide range of TM implementations—for example STMs which make in-place updates (e.g. [18, 11]), STMs which defer updates until transactions commit (e.g. [5]) as well as HTMs. We want to avoid introducing contention between non-conflicting transactions and to avoid adding costs to implementations with strong native guarantees (e.g. we want to avoid HTMs needing to track whether or not an object is protected).

2.1 Examples

Our first example uses DS in a simple sequential setting:

```
b = new Buffer();
<populate buffer b>

unprotected {
  unprotect(b);
  <use b directly>
  protect(b);
}

<use buffer>
```

It is clear that this program is correctly synchronized: the buffer is created in protected mode in the first atomic action, it is unprotected before the direct accesses to it, and it is then re-protected. Note that DS restricts where data is actually accessed by a program, not how the data is reachable through references. This lets a protected data structure be used as a conduit between pieces of unprotected code or vice-versa.

Our second example is modeled on the "privatization" idioms that have been studied in STM (e.g. [22, 15, 1]):

```
// Initially b_shared=true, b_shared protected, b protected

// Atomic action A1           // Atomic action A2
b_shared = false;             if (b_shared) {
                                <use b atomically>
// Unprotected code U1        }
unprotected {
  unprotect(b);
  <use b directly>
}
```

This example is correctly synchronized and illustrates how we define the criteria for using DS based on execution under strong semantics: If `A1` executes first then `A2` will not access `b`, whereas if `A2` executes first then it will run before `A1` sets `b_shared` to `false` and `U1` unprotects `b`. The language implementation must prevent (e.g.) `A2` accessing `b` concurrently with `U1`.

Similarly, the following example is correctly synchronized:

```
// Initially x=y=0, x protected, y protected, z unprotected

// Atomic action A1           // Atomic action A2
x = 10;                       if (x != y) {
y = 10;                         z = 42;
                              }
// Unprotected code U1
unprotected {
  <read z>
}
```

Even though the source code of `A2` contains a store to `z` this is never executed under strong semantics and so store `z=42` should never be seen. This must be true even when the example is run over an STM that does not provide this guarantee natively; the extensions we make to support DS must prevent `A2`'s implementation from trampling on `z`.

2.2 Semantics

The semantics of DS requires several subtle design choices. For example, what if `protect` is called on a location that is already protected? What if `unprotect` is called on a location that, under strong semantics, is currently being manipulated by protected code; should it block until the protected code is finished (providing a new synchronization mechanism for programmers to use), or is this an error? Can DS operations be called anywhere? Are the rules the same for all three operations? What happens if a transaction attempts to access unprotected data: should it abort with an exception, be rolled-back and re-executed, or continue regardless? What if code tries to write to read-only data?

Our goal of supporting DS over many different TM implementations provides a methodical way of selecting between different options. Conversely, other decisions would be possible if we restricted attention to particular TM implementations. Many design choices follow from considering two extreme kinds of TM:

HTM with strong atomicity. If the underlying TM provides strong atomicity then we want to use it directly without

imposing overheads (e.g. of tracking per-object protection states). This means we avoid design choices that require this information to be available at run time: we cannot require DS operations to block or fail if called on the wrong kind of data. Similarly, we cannot require data accesses to block or fail if made on the wrong kind of data.

STM with in-place updates and optimistic concurrency control. This set of STM design choices lets us examine whether or not DS operations can be used in transactions. A permissive design is simple under strong semantics: a transaction may protect and then access a location, or it may unprotect a location that it will no longer access.

However, this permissive design is problematic for STMs that use in-place updates and optimistic concurrency control. The problem occurs when a transaction protects data, experiences an undetected conflict, and then proceeds to update the data in-place. For example, consider the following variant of the privatization idiom:

```
// Initially b_shared=true, b_shared protected, b unprotected

// Atomic action A1      // Atomic action A2
b_shared = false; // 3   if (!b_shared) { // 1
unprotected {              protect(b); // 2
  <update b>;  // 5       <update b>; // 4
}                          unprotect(b);
                         }
```

If we were to allow DS operations within atomic actions then this example would be correctly synchronized (either `A1` runs first, in which case `A2` does not access `b`, or `A2` runs first and `A1` sees `A2`'s updates). However, with optimistic concurrency control, the steps could execute in the order shown: `A2` is doomed to roll back but, with lazy detection, the conflict has not yet been identified and the memory updates at 4 and 5 will race. It is insufficient to validate `A2` as part of step 2 because the conflict does not occur until step 3.

Following our goal of implementation flexibility, we therefore decide that DS operations can be invoked only in unprotected code. Again, one could take other decisions if interest were restricted to particular TM implementations.

3. Dynamic Separation in C#

In this section we consider how to apply dynamic separation to AME in C#. There are three general questions.

First, at what granularity do we associate protection status with data? We chose to dynamically associate a protection mode with each C# object. We considered alternatives: per-class settings would hinder code re-use (e.g., all `Hashtable` objects would have to be protected or all unprotected), and per-field settings would require repeated DS operations (e.g., on each element of an array, introducing similar asymptotic costs to marshaling the data by copying). We do not associate a protection mode with variables because they remain thread-local. We chose to statically declare the protection mode of static fields rather than letting them change dynamically. Our reasoning is that static fields often represent read-only state that's accessed by many threads in different protection modes: the field and the data reachable from it remain read-only (we discuss static initializers below). This is an engineering choice and could readily be revisited.

The second design question is how to express the DS operations themselves. Rather than adding explicit keywords we make the operations virtual methods on the `Object` superclass. By default these change the protection mode of the object itself. This lets the programmer override the methods to provide class-specific functionality (e.g. to change the protection mode of a whole object graph).

The final question is exactly which operations constitute "accesses" to data for the purpose of defining correct synchronization. Following our approach in Section 2.2 our design is motivated by considering a range of STM implementations and where problems or overheads would be incurred. This led us to the general principle that we only consider accesses to the normal fields of objects (or, in the case of arrays, their elements). Applying this principle to different constructs in turn:

Method calls. We do not place restrictions on method calls themselves. Protected code can call a method on an unprotected object and vice-versa.

Array lengths, type information. As with method definitions these are immutable data maintained by the runtime system rather than fields of the object concerned.

Delegates. C# delegates are roughly similar to function pointers in C++; a delegate encapsulates a reference to a particular static method or an instance method on an object. A delegate itself is a C# object with fields that represent the target. We treat delegates as C# objects, initializing their protection mode in the usual way and requiring correctly synchronized programs to access delegates in the correct mode.

Boxed values. C# provides mechanisms for "boxing" a primitive value to create a heap object that can be referred to by an ordinary object reference. As with delegates, we treat boxed values as ordinary objects.

Indirect data accesses. C# allows the creation of references to individual fields, array elements, and the like. Creating a reference does not access the underlying data, so we allow references to be created and passed freely, but require that the target's protection mode be correct when attempting an access.

Access from native code. C# provides for calls into native code by a mechanism called `p/Invoke`: a method signature is given, but the code is imported from a native library. Rules define defaults for marshaling parameters (e.g. between different string representations). There are two cases to consider for DS: (i) data accessed during marshaling, (ii) data accessed from native code. Regarding (i), native calls can occur only in unprotected code, so we treat the marshaling code as any other: a correctly synchronized program must ensure that the data being marshaled is unprotected.

Regarding (ii), if native code accesses an object's fields directly then it must "pin" the object before the call is made. Native code accessing other objects is seriously incorrect; the GC could move the objects in memory. We chose to require correctly synchronized programs to ensure all pinned objects are unprotected. This lets us express the correctness criteria in terms of C#; consequently we can check it without instrumenting the native code.

Locks, volatile fields. Lock operations and all accesses to volatile fields are not permitted in protected code, so these are always unprotected and need no special treatment.

Static initializers. Each C# class contains code to initialize its `static` fields. The runtime system executes this code in unprotected mode upon the first access to the class. We permit this code to initialize read-only fields; this is consistent with the language's existing loop-hole for initializing `readonly` fields. Static initializers must use protected code when touching protected statics.

Finalizers. An object's `Finalize` method is called automatically after the object has become unreachable. We always run finalizers in unprotected mode. This supports their intended use for clean-up work with native code. We chose against selecting dynamically based on an object's protection mode at time of death; this would require an HTM-based implementation to track the mode.

4. Implementing Dynamic Separation

In this section, we discuss implementations of DS. We focus in detail on our implementation using Bartok-STM because we believe this is the most challenging setting in which to implement DS correctly (Section 4.1). We initially assume a sequentially consistent memory model; we revisit this in Section 4.2. We then sketch alternative implementations using different TMs (Section 4.3).

4.1 Implementation Using Bartok-STM

Bartok-STM [11] uses weak atomicity with in-place updates and optimistic concurrency control. This combination of features has been found to perform well [18] and also to be particularly troublesome in terms of problems like privatization [1, 22].

Background, Bartok-STM design. The STM associates meta-data with each heap object and, within transactions, adds operations to "open" each object before it is accessed—`OpenForRead` on objects about to be read, and `OpenForUpdate` on objects about to be updated. The meta-data, called an object's "STM word", records a version number indicating how many times the object has been opened for update. This number is logged in `OpenForRead` and rechecked during transaction validation: a concurrent change indicates a conflict. The STM word also contains a flag indicating whether the object is currently "owned" by a transaction, i.e., open for update. This flag is used to enforce mutual exclusion between writers. An invalid transaction may continue to execute as a "zombie" before a conflict is detected. The runtime system sandboxes failures such as null reference exceptions if they occur in this state. The runtime system also guarantees that zombie transactions will be detected and rolled back.

Representing protected objects dynamically. Our basic approach is to modify the STM word to include a flag in place of one bit of the version number. If the flag is set then the object is protected. If the flag is clear then the object is either unprotected or read-only (as we show, this implementation need not distinguish between these cases, although our checking tool in Section 5 must). The flag is initialized along with the rest of the object's header when an object is allocated and then modified only by the implementations of `protect`/`unprotect`/`share`.

Correctness argument. Our companion paper [2] contains a correctness theorem in the context of the AME calculus. Here we include a brief informal sketch of the main points.

The modified STM implementation maintains an invariant that transactions only update objects whose protection flags are set. This means that zombie transactions will not trample on read-only or unprotected objects. So, if the program is correctly synchronized, such transactions' updates will not be seen by non-transacted code.

We maintain this invariant at run-time by (i) modifying the function `OpenForUpdate` so that it only provides access to protected objects, (ii) ensuring that `unprotect` and `share` (which revoke write access from protected code) block until there is no concurrent transaction with the object open for update, and (iii) our restriction that DS operations only occur in unprotected code rather than during the execution of a (possibly invalid) transaction.

Our treatment of objects that are read (but not updated) is more subtle: we do not need to check whether or not they are protected. The reason is that we only aim to guarantee strong semantics to correctly synchronized programs: if a program is correctly synchronized, and a transaction running in it is still valid, then it will only read from protected and read-only objects. Conversely, if the transaction is not valid, then the invalidity will be detected in the normal way. In either case, we meet the requirement to run correctly synchronized programs with strong semantics.

Pseudo-code. Figure 3 shows `DSOpenForUpdate` in pseudo-code. We use a `DS` prefix on functions provided by the new implementation with DS, and an `STM` prefix on the underlying functions provided by the existing STM.

The new function starts by opening the object for update (leaving the protection bit unchanged). Then, before the thread can update the object, it examines the protection bit. If the object is protected then the transaction proceeds as usual. If the object is unprotected then the transaction is validated. If it is valid then the program is not correctly synchronized: it is about to access an unprotected object transactionally so

```
void DSOpenForUpdate(tm_mgr tx, object obj) {
  STMOpenForUpdate(tx, obj);
  if (!IsProtected(GetSTMWord(obj))) {
    if (STMIsValid(tx)) {
      // Valid and choosing to access an unprotected object
      throw new DynamicProtectionError(); // Fail (uncatchable)
    } else {
      // Choice to access object may be based on invalid state
      STMAbort(tx); // Roll-back and re-execute
} } }
```

Figure 3. Production implementation of open-for-update supporting DS.

```
void DSUnprotect(tm_mgr tx, object obj) {
  while (true) {
    w = GetSTMWord(obj);
    if (!IsProtected(w) {
      break; // Already unprotected/readonly: done
    } else if (IsOwned(w)) {
      continue; // Wait until object not open for update
    } else {
      new_w = CreateSTMWord(w.GetVersion(),
                            NOT_PROTECTED, NOT_OWNED);
      if (CASSTMWord(obj, w, new_w)) {
        break; // Installed new STM word; done
} } } }
```

Figure 4. Production implementation of `DSUnprotect`. The implementation of `DSShareReadOnly` is identical.

the program fails with an error. If the transaction is invalid then the transaction is aborted and re-executed.

We extend the STM interface with operations that correspond to `protect`/`unprotect`/`share`. We show `unprotect` in pseudo-code in Figure 4. This implementation is a loop which repeats until either (i) it observes that the object is already unprotected (either before the call, or by a concurrent `unprotect`), or (ii) it succeeds in making the object unprotected. The second case must wait until the object is not owned by any transaction (`IsOwned` returns false) to preserve the invariant that protected code updates only protected objects. (Even in a correctly synchronized program, a zombie transaction may still have a previously-protected object open for update: we must wait for such transactions to drain from the system.)

The implementation of `share` is identical to that of `unprotect` because the run time does not need to distinguish between unprotected and read-only objects. The implementation of `protect` is symmetric to that of `unprotect` with the negation removed on `!IsProtected`, the STM word being created with a `PROTECTED` flag rather than `NOT_PROTECTED`, and the test of `IsOwned` being redundant.

4.2 Optimizations and Memory-Access Re-ordering

We have initially assumed a simple execution model ignoring transformations made by a compiler or by a processor with a relaxed memory model. Languages differ in terms of exactly which transformations are valid and so a desirable property for synchronization constructs is that they can be used over many such models and, if used correctly, can abstract the low-level details. Our design and implementation of DS has been guided by the .NET 2.0 memory model (exposed by the corresponding version of C#) [17]. However, we have considered several transformations allowed by other models to help gain confidence that DS is more broadly applicable.

We first consider whether DS operations may be reordered with other operations. Our principle is that we prohibit re-orderings that would transform a correctly synchronized program to an incorrectly synchronized one (reasoning, as usual, just using the strong semantics). For example, consider:

```
unprotected {
  unprotect(o1); // U1
  r1 = o1.x;  // R1
}
```

If `R1` could be moved before `U1` then `R1` may attempt to read from a protected object. The same problem occurs for a write and, in the case of `protect`, with reads and writes before the `protect` call. This suggests a rule such as “Reads and writes cannot move before a DS operation on the same location”.

A similar problem can occur if an `unprotect` operation is followed by a write that publishes the reference in shared memory, and then by a read in another thread. (This can occur with double-check locking which is a correct idiom in C#.) This suggests a further rule “Writes cannot move before a DS operation”.

Considering these two examples shows that many reorderings of memory accesses and DS operations must be prohibited. Therefore, for simplicity, we propose the stronger rule that “Reads and writes cannot move past DS operations”. We enforce this conservatively by having the compiler treat DS operations as having unknown side effects. This is an engineering choice that may be revisited as we consider more sophisticated optimizations.

The .NET 2.0 memory model prohibits transformations that are valid in other languages. For example, it does not permit reads to be introduced by hoisting them above an operation on which they are control-flow dependent. This property does not hold in more relaxed contexts, including the Java memory model. Such models may be attractive from a performance viewpoint—either to exploit processors with relaxed memory models, or to enable compiler transformations.

Re-ordering across dependencies is problematic: it can cause the implementation to access locations that it would not touch under the strong semantics. Consider the following “racy publication” idiom:

```
// Initially: x_shared=false, x=0,
//            x_shared protected, x unprotected

// U1                       // A2
unprotected {               r1 = -1;  r2 = x;
  x = 42;                   if (x_shared) {
  protect(&x);                r1 = r2;
}                           }
// A1
x_shared = true;
```

This is not correctly synchronized at a source level, but equivalent code may result if a compiler hoists A2's read of x to before its read of x_shared. Our implementation of DS supports such transformations.

The resulting additional reads are problematic for two reasons. First, they allow executions that are not possible for the source program under strong semantics—for example, A2 may read from x before U1's write and still commit. Second, such reads can cause a transaction to access data that is always unprotected (e.g. if the read is hoisted above a conditional that is always false)—the implementation would deadlock if the transaction waited for the object to become protected.

We deal with these reads by: (i) Making DS operations conflict with transactions read the object in question. (ii) Not checking the object's protection mode in OpenForRead or at commit-time (we must still, of course, check protection modes in OpenForUpdate). Taken together, these steps mean that data remains in the same protection mode from when it is read until when the transaction commits.

This is clearly correct if the data remains protected. To see why it is also correct for unprotected data, notice that either (i) the access would have been performed under strong semantics (hence the program is not correctly synchronized), or (ii) the access would not have been performed under strong semantics (as in "racy publication", so the program is not using the value read). In both (i) and (ii) the transaction may commit.

4.3 Alternative Implementations

We have considered implementations of DS over various different TMs. At a minimum the TM must guarantee that incorrectly synchronized programs remain type-safe. This is naturally true of designs that keep the TM's meta-data separate from the object's normal contents and use the same object layout for transacted and non-transacted data.

DS can be implemented over such TMs so long as the granularity at which the TM logs updates or undo operations is the same (or finer) than the granularity at which the language exposes DS. This ensures that the TM will not make accesses that "spill over" from the data apparently being accessed by the source code (this is similar to GLU problems [20]).

The implementation of DS is simpler when using an STM with deferred updates rather than in-place ones: the writes of zombie transactions are automatically contained. Therefore, the implementations of the DS operations do not need to wait if the object is open-for-update by a transaction that is still running. However, for unprotected code to correctly see a transaction's updates to an object, DS operations on the object must still wait for a transaction that has chosen to commit updates to the object but not yet written them back.

The implementation is simpler still when using a TM with strong atomicity: DS operations have no semantic runtime effect, although they may need to serve as memory fences depending on the language's memory model and the processor it is implemented over.

DS can also be implemented over some TMs that use different object formats for protected and unprotected data, so long as it is still type-safe for unprotected code to read from objects in their protected representation. This is true in Ennals' design [6] where the start of a protected object's representation is the same as the corresponding unprotected object, or in Fraser's design [7] if the indirection-header is overlaid on the STM word at the start of an ordinary object. In such implementations protect/unprotect would marshal between the different representations. However, supporting read-only mode requires a common object format for reads to be served from. We examine such marshaling formally in our companion paper [2].

5. Dynamically Checking Correct Usage

We extended Bartok with a debug mode that provides dynamic checks of whether or not a program run is correctly synchronized. Our goal is to report errors without any false positives, without missing error reports, and with all execution before the error being correct under strong semantics.

This works much like dynamic race detectors. In short, we introduce explicit tests on data accesses to check whether or not they are made in the correct mode. We constrain the compiler not to re-order, add, or remove memory accesses. We distinguish between three targets of memory accesses: (i) stack-allocated data, (ii) static fields, whose protection mode is given by attributes in the source code, (iii) objects, whose protection mode is set dynamically,

The first two cases are straightforward. No checks are needed on access to the stack. Checks on statics are handled during compilation. The compiler generates two versions of each method, one for protected and another for unprotected code, and so we compile correct-mode accesses as normal and incorrect-mode accesses to code that will report an error when executed.

Object accesses are handled by checking protection information in the object's STM word. Unlike the production implementation we must distinguish between unprotected data and read-only data (this lets us report errors where unprotected code attempts to update putatively read-only data). We do this by reserving a further bit from the STM word and providing operations IsProtected, IsUnprotected, and IsReadOnly to distinguish the three states. (We still have 27 bits of version number space and mechanisms to recover from overflow [11].)

We must distinguish four sources of memory accesses:

Atomic actions. We report an error if either (i) a valid transaction opens an unprotected or read-only object for writing, or (ii) a valid transaction sees an unprotected object in its read set during a successful validation.

Unprotected managed code. We must check the object's protection mode atomically with the data access: otherwise,

in an incorrectly synchronized program, a concurrent thread may protect the data and access it transactionally, letting us see a non-committed transaction's write without reporting an error. We deal with this difficulty in a similar way to Shpeisman et al.'s runtime support for strong atomicity [20]: we expand each unprotected access into a series of steps that accesses the STM word along with the data location. In effect we treat the access as a small transaction.

Runtime system (RTS) code. The STM, GC, and other pieces of the RTS are implemented in C# and compiled along with the application. The RTS performs its own concurrency control and the data it accesses is disjoint from the application data (e.g., an object's header rather than its payload fields), except for times when it accesses entire objects (e.g., during garbage collection with all threads stopped).

We must not report errors from accesses made by RTS code. We therefore introduce a new source-code attribute `RTSRoot` to identify entry points to the RTS. Such methods are compiled without access-mode checks along, recursively, with any code they call. The RTS does not call into application code and so the resulting duplication is limited to a small number of system classes (e.g., `System.UIntPtr` whose instances represent pointer-sized integers).

Native code. In correctly synchronized programs any objects being passed to native code must have been pinned in unprotected code. We test that (i) an object is unprotected when it is pinned, (ii) an object being protected is not pinned.

6. Evaluation

We have used the implementation described in Sections 4 to study the effectiveness of DS. We discuss its use in programs in Section 6.1 and performance in Section 6.2.

6.1 Case Study: a Library for File and Network IO

We experimented with the implementation of the AME IO library and web-proxy and file access applications built over it. These have concurrent activity combined with asynchronous file and network access. We wrote two versions of the web-proxy application: one in terms of classic threads and locks, the other in the AME model. The latter is substantially more straightforward, since most of the required synchronization is automatic.

The methods that the library provides for initiating its IO operations each take a similar form. Each method takes an object `ObjA` (allocated by the application in protected mode) that describes the requested operation. The method call returns a new object `ObjB` (also in protected mode) that corresponds to the in-progress operation. The application can use a field of `ObjB` with `blockuntil` to wait for completion of the operation, and may then access the results of the operation that have been stored into the original request object `ObjA`.

For example, an application can perform a read from a file by executing the following, from within protected code:

```
ioInfo = StartAsyncRead(readBlock);
yield();
blockuntil(ioInfo.IsCompleted);
// access the status code and data from fields of "readBlock"
```

The application could, if it wished, place the call of `blockuntil` and subsequent code into a separate asynchronous method call (and consequently omit the call of `yield`).

For the purposes of the present paper, the interesting code lies inside the IO library, which at its core must use unprotected code to access the underlying IO calls provided by the operating system. This works as follows.

Running inside a transaction, the call of `StartAsyncRead` places the request (`readBlock`) onto an internal (global) queue:

```
ioInfo = new Action(readBlock, ...);
if (queue.head == null) {
  queue.head = ioInfo;
} else {
  queue.tail.next = ioInfo;
}
queue.tail = ioInfo;
```

When this transaction commits, the updated contents of the queue are visible to a permanently executing thread (`TSend`) inside the IO library, which extracts the request, then uses unprotected code to hand it to the operating system:

```
do {
  blockuntil(queue.head != null);
  ioInfo = queue.head;
  queue.head = queue.head.next;
  buffer = ioInfo.buffer;
  unprotected {
    unprotect(buffer);
    osReq = new OSReq(ioInfo, buffer, ...);
    ... call the operating system ...
  }
}
```

We use the Windows "completion port" mechanism to receive notification when the actual IO finishes. A second permanently executing thread (`TReceive`) in the IO library uses unprotected code to receive this notification, then updates the `IsCompleted` field of the request (thus allowing the application to complete its call of `blockuntil`):

```
do {
  unprotected {
    ... wait in the OS for an event completion ...
    ... assign the OS request object to "osReq" ...
    protect(osReq.buffer);
    ioInfo = osReq.ioInfo;
  }
  ioInfo.IsCompleted = true;
}
```

Note the protection mode of the various objects involved. The original `readBlock` object is accessed only from within protected code, as is the `ioInfo` object. The actual data array that will be presented to the underlying operating system is `readBlock.buffer`. This array was allocated (protected) by the application, and after completion will be read (protected) by the application. But `TSend` unprotects it before handing the array to the operating system, and `TReceive` protects it again before setting `ioInfo.IsCompleted`.

We have a few conclusions from this exercise:

- The basic ideas of DS work as intended.
- The read-only mode is particularly useful for data managed by core libraries that are used by protected and unprotected code, for example the "empty string" object, and tables for localization and suchlike.
- The dynamic transition between protection modes does indeed allow efficient transfer of data in and out of protected code.
- It is very convenient to be allowed to have an unprotected object (our data buffer during the actual underlying IO operation) carried within an object that remains protected (the `readBlock` request descriptor). Similarly, a protected object `ioInfo` is carried within unprotected data to be sent through the completion port.

6.2 Performance

The performance of a program using DS depends on several factors: the immediate cost of the DS operations, the overhead that supporting them adds to the TM, and any costs incurred in structuring the program to use DS. We focus initially on the Bartok-STM implementation and then discuss alternatives.

Using Bartok-STM, the fast-path of the DS operations is a single read then compare-and-swap (CAS) on the object's STM word. If the CAS fails then the slow path distinguishes the different cases as in the pseudo-code of Figure 4. DS operations only block if the object is open-for-update by a transaction (which, in a correctly synchronized program, must be a zombie transaction). This delay is the same as for a non-transactional access in an implementation of strong atomicity following Shpeisman et al.'s design [20].

Supporting DS adds no overhead to the fast-path of the existing STM operations: the check of whether or not an object is protected is combined with an existing test of whether or not it is open for update (both look at low-order bits in the STM word).

These performance characteristics would change slightly for an STM with deferred updates: the DS operations would never need to wait for transactions to be rolled back, though they may still block while another transaction is committing. Again, these costs follow those of a non-transacted access in Shpeisman et al.'s design. With hardware support for strong atomicity the DS operations would be no-ops and, of course, no changes would be needed to the TM implementation.

A more subtle question is how performance is affected by structuring code to use DS rather than some other discipline. There are both positive and negative effects. Comparing with static separation, DS may allow marshaling code to be removed. Conversely, when comparing with violation-freedom or a single-global-lock discipline, it may be necessary to add DS operations and structure the program so they are called appropriately. This is a complex trade-off: the DS operations add a cost, but the underlying implementations of more permissive models limit scalability by introducing synchronization between non-conflicting transactions [15] and preclude the use of in-place updates which have been found to perform well [18].

We examined the performance of two applications over the AME IO library. The first of these, `FileTest`, is a micro-benchmark which copies a file on disk, structured using a loop that performs asynchronous IO requests. We build two versions: "dummy" in which the underlying IOs are not sent to the kernel, and "real" in which they are. The dummy version makes this CPU-bound, highlighting the overhead added by the DS operations. The second program, `WebProxy` is a caching web proxy which interacts with multiple concurrent clients and web servers, maintaining an on-disk cache of requested pages. We load the web proxy with 1..4 concurrent client requests. In each case we use sufficiently large files that the execution time is readily measurable. We use an otherwise-unloaded machine with dual 4-core processors and plentiful memory. The applications are quite simple, and our experiments can be interpreted mostly as a sanity check that our implementation does not introduce any unexpected overhead.

Figure 5 shows the results. We compare five different implementations. "Baseline" uses the underlying Bartok-STM with DS disabled. We normalise against its performance. "Baseline + DS" is our implementation of DS. "Run-time checking" is the implementation described in Section 5. The WebProxy performs and scales identically to a (much more complicated) alternative built using traditional synchronization.

As expected, the overhead of "Baseline + DS" over "Baseline" is low, even in the CPU-bound program. However, the "Baseline" is not a correct implementation because it may allow undetected conflicts between transacted and non-transacted accesses in correctly synchronized programs. To confirm that this did not distort results (for example, if such race conditions delayed the baseline execution), we built an alternative "Serialized baseline" which serializes transactions with a global lock. This correctly supports DS with the operations compiled as no-ops. We compare this with "Serialized baseline + DS" adding the normal DS implementation.

Finally, we studied an implementation of the AME IO library built to maintain static separation between transacted and non-transacted data. Prior to developing DS this was the only correct programmer-centric programming model we had identified for writing programs with Bartok-STM. Static separation requires data to be marshaled between access modes. Even with the IO-intensive AME applications we are using this was over a decimal order of magnitude slower in total execution time than "Baseline + DS".

7. Related Work

Adve and Hill pioneered the approach of requiring correctly synchronized programs to run with sequential consistency,

	FileTest (dummy)	FileTest (real)	WebProxy (1)	WebProxy (2)	WebProxy (3)	WebProxy (4)
Baseline	1.00	1.00	1.00	1.11	1.27	1.49
Baseline + DS	1.00	1.00	1.00	1.11	1.27	1.49
Serialized baseline	1.41	1.27	1.00	1.11	1.27	1.49
Serialized baseline + DS	1.42	1.27	1.00	1.11	1.27	1.49
Run-time checking	1.01	1.02	1.00	1.11	1.27	1.49

Figure 5. Performance of test applications—execution time, normalised against "baseline" and, for WebProxy, a 1-client workload.

and the use of a programmer-centric definition of which programs are correctly synchronized [3]. Saraswat et al. subsequently termed this the "fundamental property" of memory models [19]. Spear et al. [22] and Abadi et al. [1] concurrently identified the link between that work and languages implemented over TM with weak atomicity.

Hill subsequently argued that hardware should provide sequential consistency [12]. However, this is separate from the design of the language's memory model which must also consider program transformations made by the compiler.

Many papers have examined different programming disciplines which, as with DS, provide strong semantics to a class of programs built over transactional memory.

7.1 Strong Programming Disciplines

It may be desirable in principle to provide strong atomicity between transacted and non-transacted memory accesses, and provide constructs (such as `volatile` modifiers) to constrain re-ordering or prevent memory accesses from being eliminated. Shpeisman et al. showed how to build this over an STM that natively provides weak atomicity [20]. Lev and Maessen introduced the idea of compiling non-transactional memory accesses to include a run-time check of whether the data is visible to transactions [14]. If the data is visible then it is accessed using the TM. They track data's visibility at run-time, marking objects as transacted when they are made reachable via an existing transacted object.

Without hardware support, none of these approaches meet our goal of supporting implementations with weak atomicity (e.g. so that code from the kernel, other processes or DMA transfers from devices can access the data).

7.2 Violation-Freedom and Single-Global-Lock Atomicity

Violation freedom [1] formalizes the notion that the same data should not be accessed transactionally and non-transactionally at the same time. However, as we showed in earlier work [1], supporting violation-free programs with strong semantics is incompatible with our goal of implementation flexibility: we cannot use implementations with optimistic concurrency control, in-place updates and weak atomicity.

Menon et al. [15] defined a "single-global-lock atomicity" (SGLA) semantics for transactions in Java, by relating the behavior of a program using `atomic` blocks to one where those blocks are replaced by synchronized regions on a single process-wide lock. This, in turn, defines a notion of correct use of `atomic` blocks in terms of the existing definition of correct use of locks.

As with violation freedom, supporting SGLA does not meet our goal of implementation flexibility. Furthermore, known implementations of SGLA (and the weaker definitions Menon et al. studied [15]) either involve pessimistic read locks or synchronization between non-conflicting transactions (again inconsistent with our goal for implementation flexibility and parallelism).

7.3 Transactional Fences

Spear et al. discussed several ways to implement privatization idioms correctly, proposing "transactional fences" and "validation fences" [22]. Unlike DS these require synchronization with all concurrent transactions, rather than just those accessing the object in question. Wang et al. used similar global operations in an implementation of atomic blocks for C [23]: a shared list of active transactions is manipulated when transactions start or commit.

7.4 Static Separation

Under "static separation" disciplines data is split into transacted data and non-transacted data. Several definitions of static separation have been considered, typically providing enforcing this distinction through a type system ([10, 1, 16]).

While static separation is appealing in functional languages like Haskell [10], it is less palatable in imperative languages where most data comprises mutable shared objects. There are two problems here. First, data has to be marshaled between different access modes by copying. Second, if static separation is expressed through a type system, then simple versions of static separation can impede code re-use (much like all simple type systems).

DS attempts to address these concerns by allowing code to be re-used in transacted and non-transacted contexts, and by allowing data to be marshaled between access modes without copying.

8. Conclusion and Future Work

This paper introduces the idea of dynamic separation for sharing data between transacted and non-transacted code. We believe DS has several appealing properties. It is built on a simple formal definition for correct synchronization which might serve as the foundation for further formal reasoning and for static checking (alongside the dynamic checking that we have already explored). It can be implemented over a wide range of STMs with weak atomicity, and it can also be implemented over HTMs with strong atomicity without imposing a runtime overhead for protection flags. It does not introduce synchronization between non-conflicting transactions. It allows unprotected data to be accessed freely by system calls and DMA transfers.

These benefits come at the cost of requiring explicit DS operations, and the restriction that these occur only in non-transacted code. This trade-off is motivated by the AME model: application code runs by default in atomic actions and DS is used at the boundary between this code and the non-transacted libraries that it calls into, much as explicit pinning is already used across such boundaries. Nevertheless, we are currently investigating the possibility of inserting the DS operations automatically for some classes of programs.

To what extent is DS an appropriate discipline for alternative programming models like C# or Java with `atomic` blocks? We previously showed that these constructs can be encoded in AME [1], so the theory carries over. However, the constraint that DS operations happen only outside transactions seems less palatable because boundaries between the two modes occur in application code, not just at the edge of it. It may be possible to remove this constraint for implementation without in-place, while retaining the ability of non-conflicting transactions to run and commit in parallel.

Acknowledgments

We are grateful to Katie Coons, Katherine Moore, Rebecca Isaacs, Yossi Levanoni, and JP Martin for helpful discussions and comments.

References

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[2] Martín Abadi, Tim Harris, and Katherine F. Moore. A model of dynamic separation for transactional memory. March 2008. Draft.

[3] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. *SIGARCH Comput. Archit. News*, 18(3a), 1990.

[4] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Proc. 2005 Workshop on Duplicating, Deconstructing and Debunking*, 2005.

[5] Dave Dice, Ori Shalev, , and Nir Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, 2006.

[6] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[7] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

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

[9] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[10] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[11] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[12] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *Computer*, 31(8), 1998.

[13] Michael Isard and Andrew Birrell. Automatic mutual exclusion. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, May 2007.

[14] Yossi Lev and Jan-Willem Maessen. Towards a safer interaction with transactional memory by tracking object visibility. In *Proceedings of SCOOL 2005*.

[15] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic STM. In *Proceedings of TRANSACT 2008*.

[16] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[17] Vance Morrison. Understand the impact of low-lock techniques in multithreaded apps. *MSDN Magazine*, October 2005.

[18] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proc. Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[19] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on*

Principles and practice of parallel programming, 2007.

[20] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[21] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, 2007.

[22] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report 915, CS Dept, U. Rochester, 2007.

[23] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. *Code Generation and Optimization, 2007. CGO '07. International Symposium on*, 2007.