Implementation and Use of Transactional Memory with Dynamic Separation

Martín Abadi[1,2], Andrew Birrell[1], Tim Harris[3]
Johnson Hsieh[1], and Michael Isard[1]

[1] Microsoft Research, Silicon Valley
[2] University of California, Santa Cruz
[3] Microsoft Research, Cambridge

Abstract. We introduce the design and implementation of dynamic separation (DS) as a programming discipline for using transactional memory. Our approach is based on the programmer indicating which objects can be updated in transactions, which can be updated outside transactions, and which are read-only. We introduce explicit operations that identify transitions between these modes of access. We show how to guarantee strong semantics for programs that use these DS operations correctly, even over an STM implementation that provides only weak atomicity. We describe a run-time checking tool (analogous to a data-race detector) that can test whether or not a program is using DS operations correctly. We also examine the use of DS in an asynchronous IO library.

1 Introduction

Recently there has been much work on implementing atomic blocks over transactional memory (TM [1]). This approach provides an alternative to using locks and condition variables for shared-memory concurrency. Much effort has focused on the language constructs that are exposed to the programmer [2–5] and the semantics that an implementation of these constructs must obey [6–8]. The interaction between program fragments running transactionally and those running concurrently in normal code has been found to be particularly subtle [6, 9, 7, 5]. A problematic example is the "privatization" idiom [10–12, 6, 5]:

```
// Initially x==0, x_shared=true

      // Thread 1                              // Thread 2
T1.1: atomic {                          T2.1: atomic {
T1.2:   x_shared = false;               T2.2:   if (x_shared) {
T1.3: }                                 T2.3:     x ++;
T1.4: // Access x non-transactionally:   T2.4:   }
T1.5: x = 100;                          T2.5: }
```

A programmer might reason that Thread 1's update to `x_shared` at line `T1.2` allows its subsequent update to `x` at `T1.5` to be made as a normal non-transactional store. After these fragments have run, a programmer might expect that `x==100` whichever order the atomic blocks ran in. However, implementations over software transactional memory (STM [13]) lead to other results, e.g., `x==1` if the

implementation of Thread 2's atomic block was still writing back a buffered update to `x` concurrently with Thread 1's non-transactional store at `T1.5`.

In this paper we describe the design and implementation of a technique called dynamic separation (DS) for controlling such interactions. With DS each object has a "protection mode" that says whether or not the object can be accessed transactionally, and the programmer explicitly indicates when this mode should be changed. In our privatization example, the programmer would add a statement at line `T1.4` to change the protection mode of `x`.

DS provides more flexibility to the programmer than existing notions of static separation that require each piece of data to be accessed either only transactionally or only non-transactionally [6, 2, 7]. With static separation we could not have objects like `x` which change protection modes. At the same time, DS provides less flexibility to the programmer than violation-freedom [6], which allows objects like `x` to change protection modes implicitly. DS also provides less flexibility to the programmer than disciplines that offer strong atomicity [14–16].

In a companion paper we study DS and its relationship to other programming disciplines from a formal point of view [17]. In this paper we focus on the design and practical implementation of DS. Our technical report [18] provides further details.

We introduce DS in detail in Section 2. We define criteria for a program using DS to be "correctly synchronized". Informally, these criteria mean that the program never tries to access transactional data from outside a transaction, nor to access non-transactional data from inside a transaction. These criteria provide a contract between the programmer and the language implementor: if a program is correctly synchronized then the implementation of DS must run the program consistently with a simple interleaved model of execution in which complete transactions run as single atomic steps and the effects of program transformations or relaxed processor memory models are not visible. We call this the "strong semantics" [6].

Such a contract benefits programmers by insulating them from TM implementation details: correct programs run with strong semantics on all correct implementations. This guarantee is convenient in the short term, since it simplifies the task of learning to use transactional constructs. In the longer term it is important that programs be portable: that they run efficiently, with identical semantics, over a range of STM implementations, and that they continue to run with the same semantics, and without unnecessary overheads, once hardware transactional memory (HTM) becomes widely available. In addition, such a contract benefits the language implementor by providing a clear definition of which program transformations and implementation techniques are correct.

We discuss the application of DS to C# in Section 3, along with its implementation over an STM with in-place updates and optimistic concurrency control [19]. We sketch a correctness argument—in our companion paper we prove the correctness of a model based on our implementation [17].

In Section 4 we describe how we compile programs using DS in a debugging mode that dynamically checks whether or not a program is correctly synchro-

nized from the point of view of a particular program run. The checking method gives no false alarms (i.e., no errors are reported for programs that are correctly synchronized), and no missing error reports (i.e., if there is no error report then the program executed with strong semantics in that run).

We evaluate the use of DS in a set of applications built over a concurrent IO library (Section 5). We wrote these applications in the AME programming model [3] where—in contrast with typical approaches that employ atomic blocks—the transactions are not block-structured, and the majority of execution occurs within transactions rather than outside them. We examine the extent to which DS may work in a "traditional" use of atomic blocks in Section 6.

We discuss related work in Section 7 and conclusions in Section 8.

2 Dynamic Separation

DS can be summarized as follows:

- We distinguish dynamically between transactional ("protected") data, non-transactional ("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 (`protect`/`unprotect`/`share`) to move data between these modes.
- For a program to be "correctly synchronized" it must use these operations so that it obeys two rules when run under strong semantics:
  - Rule-1: The program accesses data only in the correct mode: read-only data may be read anywhere but not updated, protected data may be accessed freely inside transactions, and unprotected data may be accessed freely outside transactions.
  - Rule-2: The DS operations to move data between these modes occur only outside transactions.

If a program obeys these rules then the language implementation is required to run it with strong semantics even if the underlying STM provides weaker guarantees. As an illustration, we return to the privatization example from the Introduction with an explicit `unprotect` operation added at line `T1.4`:

```
// Initially x==0, x_shared=true.
// Both variables are initially protected.

T1.1:  atomic { // A1                    T2.1:  atomic { // A2
T1.2:   x_shared = false;                T2.2:    if (x_shared) {
T1.3:  }                                 T2.3:      x ++;
T1.4:  unprotect(x);                     T2.4:    }
T1.5:  x = 100;                          T2.5:  }
```

To show that the example is correctly synchronized we need to consider the different possible executions under strong semantics, and show that none of the conditions in Rule-1 and Rule-2 is violated.

Rule-2 is satisfied because the only DS operation, `T1.4`, occurs outside the atomic blocks. Rule-1 is satisfied with respect to the accesses to `x_shared` because that variable is initially protected, and is accessed only inside the atomic blocks. We must consider two cases to show that Rule-1 is also satisfied with respect to the accesses to `x`: (i) if `A1` executes before `A2` then `A2` will see `x_shared==false`, so `A2` will not attempt to access `x`; (ii) if `A1` executes after `A2` then `A2` will access `x` when `x` is still protected. In either case, the accesses to `x` satisfy Rule-1.

Since the example is correctly synchronized, an implementation that supports DS must run it with strong semantics. To illustrate why this requirement may pose a problem, consider the execution of this example using an STM, such as Bartok-STM [19], that employs a combination of commit-time conflict detection and in-place updates. Suppose that `A2` executes up to line `T2.3`, and `A1` then executes in its entirety. The implementation will allow `A1` to commit successfully, and will force `A2` to roll back only at the point when it tries to commit. However, before `A2` reaches that point, `A2` will execute line `T2.3` and will increment the value of `x`. The implementation of `unprotect` must ensure that `T2.3` does not race with `T1.5`. Our implementation does this by causing an `unprotect` operation to block until conflicting transactions have finished rolling back. We present our implementation based on Bartok-STM in detail in Section 3.

Next, we resolve some subtleties in the details of DS, and discuss the rationale for the design (in particular the reasons for Rule-1 and Rule-2). Three principles motivate many of our design decisions:

1. The "fundamental property" [20]: The criteria for using DS correctly should be defined in terms of a program's execution under strong semantics. This makes programs portable across TM implementations.
2. Compatibility with normal memory accesses: We want to avoid needing to modify non-transactional memory accesses; we do not want to modify accesses from within the kernel, and we cannot add barriers to reads and writes performed by direct-memory-access (DMA) from devices.
3. Implementation flexibility and parallelism: We want to support a wide range of implementations—for example STMs which make in-place updates (e.g., [19, 21]), STMs which defer updates until transactions commit (e.g., [11]) as well as HTMs and implementations based on lock inference. We want to avoid introducing contention between non-conflicting operations and to avoid adding costs to implementations with strong native guarantees (e.g., implementations based on lock inference should not have to dynamically track which objects are protected).

The semantics of DS requires several delicate design choices. For example, what if `protect` is called on a location that is already protected? Could DS operations be called anywhere (that is, could Rule-2 be eliminated)? What happens if data is accessed in the wrong way: should the access fail with an exception, or continue regardless? If such an access is implemented by a transaction, then should the transaction block, or be rolled-back and re-executed? What if code tries to write to read-only data?

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

- HTM with strong atomicity: We do not want to impose the overhead of tracking per-object protection states when the underlying TM provides strong atomicity. Hence 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 implemented with in-place updates and optimistic concurrency control: Considering this particular kind of STM motivates the rule that DS operations cannot occur inside transactions. The following example, which does not obey Rule-2, illustrates this point:

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

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

  If we were to allow DS operations within atomic blocks 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. We therefore decide that DS operations cannot be invoked inside atomic blocks. Again, one could make other decisions if interest were restricted to particular implementation techniques. We return to this point in Section 6.

3 Implementing Dynamic Separation in C#

In this section, we discuss implementations of DS. First, we describe how we apply the idea of DS to the C# language (Section 3.1). Second, we describe how we extend the Bartok-STM implementation to support correctly synchronized programs with strong atomicity (Section 3.2).

3.1 Dynamic Separation in C#

Three general questions arise in applying dynamic separation to C#:

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 mode dynamically. Our reasoning is that static fields often represent read-only state that is accessed by many threads in different protection modes: the field and the data reachable from it remain read-only. This engineering choice could readily be revisited.

The second design question is how to represent the DS operations. Rather than adding keywords we make the operations virtual methods on the `Object` superclass. By default these methods 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 our design is motivated by considering a range of implementation techniques, and where problems or overheads would be incurred. This led us to the general principle that we police only accesses to the normal fields of objects (or, in the case of arrays, their elements); accesses to read-only information such as virtual method tables are permitted anywhere. Our technical report considers a number of language features in detail [18].

3.2 Implementation in Bartok-STM

Bartok-STM [19] uses weak atomicity with in-place updates and optimistic concurrency control. This combination of features has been found to perform well [21] and also to be particularly troublesome in terms of problems like privatization [6, 5]. Therefore we focus in detail on it because we believe that this is the most challenging setting in which to implement DS correctly.

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 re-checked 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 [11]. 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. We modify the STM word to include a flag in place of one bit of the version number. If the flag is set then

```
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
} } }
```

Fig. 1. Production implementation of open-for-update supporting DS.

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 4 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 [17] 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 update only 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-transactional code.

We maintain this invariant by (i) modifying the function `OpenForUpdate` so that it provides access only 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 (note that since DS operations can be used only outside transactions, this does not provide a way to create deadlock between transactions), and (iii) our restriction that DS operations occur only 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 aim to guarantee strong semantics only for correctly synchronized programs: if a program is correctly synchronized, and a transaction running in it is still valid, then it will read only 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 1 shows `DSOpenForUpdate` in pseudo-code. (We use a `DS` prefix on functions provided by the new run-time with DS, and an `STM` prefix on the underlying functions provided by the existing STM.) The implementation starts by opening the object for update, leaving the protection bit unchanged. Then, before the transaction can update the object, it examines the protection bit. If the object is protected then the transaction proceeds as usual. Otherwise, if the object is not protected, then the transaction is validated. If it is valid then

```
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
} } } }
```

Fig. 2. Production implementation of `DSUnprotect`.

the program is not correctly synchronized: it is about to access an unprotected object transactionally so 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`, and `share`. We show `unprotect` in pseudo-code in Figure 2. 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. In the second case, execution cannot proceed 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 STM does not need to distinguish read-only objects from unprotected ones. 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 Dynamically Checking Correct Usage

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

We do not place any dynamic checks on accesses to local variables since stacks are thread-local in C#. We handle accesses to static fields during compilation: the compiler generates two versions of each method, one for use inside transactions, and another for use outside. We compile correct-mode accesses as usual and incorrect-mode accesses to code that will report an error if it is executed.

Object accesses are handled by checking protection information in the object's STM word. Unlike in the production implementation we must distinguish

between unprotected data and read-only data, in order to report errors where unprotected code attempts to update putatively read-only data. We make this distinction by reserving a further bit from the STM word. (We still have 27 bits of version number space and mechanisms to recover from overflow [19].)

We must distinguish four sources of memory accesses:

1. Transactional code: At runtime we must 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.
2. Non-transactional 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.* [14]: we expand each non-transactional 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.
3. Runtime system (RTS) code: The GC and other pieces of the RTS are implemented in C# and compiled along with the application. The RTS performs its own concurrency control—e.g., using locks to protect free-lists in the memory allocator, or ensuring that all application threads are stopped before the GC traverses the heap. We must not report errors from such 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, so the resulting duplication is limited to a small number of system classes (e.g., `System.UIntPtr` whose instances represent pointer-sized integers).
4. Native code: In correctly synchronized programs an object passed to native code must have been pinned in unprotected code. We test that (i) an object is unprotected when it is pinned, and (ii) an object being protected is not pinned.

5 Evaluation

We have used the implementation described in Section 3 to study the effectiveness of DS. We evaluate DS within the AME programming model [3]. In this setting, all code runs inside a transaction by default and non-transactional code is explicitly delimited by the programmer. In Section 6 we briefly discuss how DS might be used in a traditional TM programming model with atomic blocks.

The performance of a program with 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.

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 2. DS operations block only 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 typical software implementations of strong atomicity [14–16]. 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.

These performance characteristics would change slightly for an STM with deferred updates: the DS operations would never need to wait for transactions to roll back, though they might still block while a transaction is committing. Again, these costs resemble those of a non-transactional 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 a program to be correctly synchronized under DS. There are both positive and negative effects. In comparison with static separation, DS may allow marshaling code to be removed. In comparison with violation-freedom or a single-global-lock discipline, DS requires the DS operations themselves, of course, and also that the program be structured so that the DS operations are called appropriately. Moreover, while the DS operations add a cost, the underlying implementations of more permissive models limit scalability by introducing synchronization between non-conflicting transactions [8] and preclude the use of in-place updates which have been found to perform well [21].

We examined the performance of two applications built over an IO library used with the AME programming model [3]. Most of the code in these applications executes transactionally, with brief calls out into normal code to perform IO requests that have been enqueued by a transaction. Buffers are transferred between these modes by using DS operations. We describe the design and implementation of the IO library more thoroughly in our technical report [18].

The first application, `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 loop CPU-bound, highlighting the overhead added by the DS operations. The second application, `WebProxy`, is a caching web proxy which interacts with multiple concurrent clients and web servers, maintaining an on-disk page cache. 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. Both 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 3 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 4. `WebProxy` performs and scales

	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	1.41	1.27	1.00	1.11	1.27	1.49
Serialized + 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

Fig. 3. Performance of test applications—execution time, normalised against "baseline" and, for `WebProxy`, a 1-client workload.

identically to a (much more complicated) alternative built using traditional synchronization.

As expected, the overhead of "Baseline + DS" over "Baseline" is less than 1%, even in the CPU-bound program. However, the "Baseline" is not a correct implementation because it may allow undetected conflicts between transactional and non-transactional 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" implementation that serializes transactions with a global lock wrapped around the baseline STM implementation. This implementation correctly supports DS with the operations compiled as no-ops. In "Serialized + DS", we add the normal DS implementation.

Finally, we studied an alternative implementation of the IO library built to maintain static separation between transactional and non-transactional 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 made the total execution time over 10 times longer than "Baseline + DS".

6 Using Dynamic Separation with Atomic Blocks

We designed the DS operations alongside the AME programming model [3]. There are several differences between AME and typical proposals to extend mainstream languages with atomic blocks. First, in the AME model, a program consists almost entirely of atomic sections. These are punctuated by "unprotected" code blocks which finish the ongoing atomic section, execute non-transactionally, and then start a new atomic section. Consequently the atomic sections are not necessarily block-structured. The second difference is that unprotected blocks are primarily intended for use in low-level libraries (such as the IO library of the examples in Section 5). They typically occur at the interface between code written in C# and native code, and include low-level operations like pinning objects in memory so that the GC does not move or reclaim them. In this context it seems reasonable to add other explicit operations, like those for DS.

To what extent is DS an appropriate discipline for programming with block-structured transactions in a mainstream language? We previously showed that such a language can be encoded in AME [6], so the theory carries over. The question is whether DS forms a palatable programming model.

One seemingly attractive feature of programming with atomic blocks is the notion that arbitrary sections of code may be placed in atomic blocks, so long as the program is correctly synchronized. This feature would not hold under our DS design, since changes in an object's protection mode may not occur inside atomic blocks (Rule-2 of Section 2). Consequently, any code that uses DS operations may be executed only non-transactionally, and programmers must be aware of whether or not functions that they call might use DS operations internally.

However, we speculate that programs in which data changes between transactional and non-transactional accesses will need to be carefully written anyway, in order to avoid race conditions. It may not be unreasonable, therefore, to imagine that programmers will already need to be aware of whether or not a given function call will attempt to change the access mode of a given piece of data.

If Rule-2 were to prove problematic, we believe it would be possible to permit DS operations to occur anywhere in a program, at the loss of some implementation flexibility. In particular, given the last example from Section 2, we believe that this change would restrict DS to STMs that make deferred updates or that detect conflicts eagerly.

7 Related Work

Adve and Hill pioneered the approach of requiring correctly synchronized programs to run with sequential consistency, and the use of a programmer-centric definition of which programs are correctly synchronized [22]. Hill subsequently argued that hardware should provide sequential consistency [23]. However, the design of a language's memory model must consider not only the properties of hardware but also program transformations made by compilers. Spear *et al.* [5] and Abadi *et al.* [6] concurrently identified the link between Adve and Hill's work and languages implemented over TM with weak atomicity.

Many papers have examined semantics and corresponding programming disciplines for the use of TM:

Strong programming disciplines. Shpeisman *et al.* showed how to guarantee strong atomicity over an STM that natively provides weak atomicity [14]. Subsequent work has improved on the performance of such implementations [15, 16]. Lev and Maessen introduced the idea of compiling non-transactional memory accesses to include a run-time check of whether the data being accessed is visible to transactions [24]. If so, the data is accessed using the TM. They track data's visibility at run-time, marking objects as transactional when they are made reachable via an existing transactional object. Without hardware support, none of these approaches meets our goal of allowing implementations with weak atomicity in which the kernel or DMA transfers can access program data without copying.

Violation-freedom. Violation-freedom [6] formalizes the notion that the same data should not be accessed transactionally and non-transactionally at the same time. Running violation-free programs with strong semantics seems to conflict with our goal of implementation flexibility: it can preclude STM implementations with optimistic concurrency control and in-place updates [6].

Single-Global-Lock Atomicity (SGLA). Menon *et al.* [8] defined a "single-global-lock atomicity" semantics for transactions in Java. SGLA relates the behavior of a program with atomic blocks to one where those blocks are replaced by synchronized regions on a process-wide lock. The transactional program is correctly synchronized if the resulting lock-based program is correctly synchronized under the Java memory model. Supporting SGLA (like assuming violation-freedom) does not meet our goal of implementation flexibility. Known implementations of SGLA involve either pessimistic read locks or synchronization between non-conflicting transactions. The same is true for the weaker definitions studied by Menon *et al.* [8].

Transactional fences. Dice and Shavit discuss the need for an operation to "quiesce" a transactionally accessed object before it is deallocated after a transaction [10]. This operation ensures that the STM implementation has finished all accesses to the object before, for example, the page holding it might be returned to the operating system. Wang *et al.* used a form of quiescence in an implementation of atomic blocks for C [12]: a shared list of active transactions is manipulated when transactions start or commit. Wang *et al.*'s implementation uses this list to ensure that code running after a transaction will see updates made by other transactions that have been serialized before it. This operation requires synchronization with all concurrent transactions, rather than just those accessing a specific object. Spear *et al.* discuss several ways to implement privatization idioms correctly, including explicit "transactional fences" and "validation fences" [5]. A thread calling a transactional fence is blocked until any concurrent transactions have committed. A validation fence is similar, except that a thread may proceed once concurrent transactions have been validated. Unlike SGLA and violation-freedom, supporting these fences seems compatible with a wide range of TM implementations that allow non-conflicting transactions to run without synchronization between their implementations.

Static separation. Under static separation disciplines, each piece of data is accessed either only transactionally or only non-transactionally. Several definitions of static separation have been considered, typically implemented via type systems ([6, 2, 7]). While static separation is appealing in functional languages like Haskell [2], it is less palatable in imperative languages where most data comprises mutable shared objects. Data has to be marshaled between different access modes by copying. Moreover, 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 allows data to be marshaled between access modes without copying. Our implementation of DS aids code re-use by checking dynamically that data is accessed in the correct mode, rather than using a simple type system.

8 Conclusion

We believe that DS has several appealing properties. It can be used over a wide range of TM implementations. It does not introduce synchronization between non-conflicting transactions, and it allows unprotected data to be accessed freely by system calls and DMA transfers. When used with HTMs or with lock inference, it avoids imposing a runtime overhead for protection flags. Finally, DS is based on a simple, precise definition for correct synchronization which may serve as the foundation for further formal reasoning and for static checking.

Acknowledgements We are grateful to the anonymous reviewers, and to Katie Coons, Rebecca Isaacs, Yossi Levanoni, Jean-Philippe Martin, Mark Moir, and Katherine Moore for helpful discussions and comments.

References

1. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA '93, 20th International Symposium on Computer Architecture. (May 1993) 289–301
2. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP '05, 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (June 2005) 48–60
3. Isard, M., Birrell, A.: Automatic mutual exclusion. In: HotOS '07, 11th Workshop on Hot Topics in Operating Systems. (May 2007)
4. Smaragdakis, Y., Kay, A., Behrends, R., Young, M.: Transactions with isolation and cooperation. In: OOPSLA '07, 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications. (October 2007)
5. Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization techniques for software transactional memory. Technical Report 915, CS Dept, U. Rochester (February 2007)
6. Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. In: POPL '08, 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (January 2008) 63–74
7. Moore, K.F., Grossman, D.: High-level small-step operational semantics for transactions. In: POPL '08, 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (January 2008) 51–62
8. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R.L., Saha, B., Welc, A.: Practical weak-atomicity semantics for Java STM. In: SPAA '08, 20th Symposium on Parallelism in Algorithms and Architectures. (June 2008) 314–325
9. Blundell, C., Lewis, E.C., Martin, M.M.K.: Deconstructing transactional semantics: The subtleties of atomicity. In: WDDD '05, 4th Workshop on Duplicating, Deconstructing and Debunking. (June 2005) 48–55
10. Dice, D., Shavit, N.: What really makes transactions faster? In: TRANSACT '06, 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing. (June 2006)
11. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: DISC '06, 20th International Symposium on Distributed Computing. (September 2006) 194–208

12. Wang, C., Chen, W.Y., Wu, Y., Saha, B., Adl-Tabatabai, A.R.: Code generation and optimization for transactional memory constructs in an unmanaged language. In: CGO '07, International Symposium on Code Generation and Optimization. (March 2007) 34–48
13. Shavit, N., Touitou, D.: Software transactional memory. In: Proc. 14th Annual ACM Symposium on Principles of Distributed Computing. (August 1995) 204–213
14. Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing isolation and ordering in STM. In: PLDI '07, ACM SIGPLAN Conference on Programming Language Design and Implementation. (June 2007) 78–88
15. Schneider, F.T., Menon, V., Shpeisman, T., Adl-Tabatabai, A.R.: Dynamic optimization for efficient strong atomicity. In: OOPSLA '08, 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications. (October 2008) 181–194
16. Abadi, M., Harris, T., Mehrara, M.: Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In: PPoPP '09, 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (February 2009)
17. Abadi, M., Harris, T., Moore, K.F.: A model of dynamic separation for transactional memory. In: CONCUR '08, 19th International Conference on Concurrency Theory. (August 2008) 6–20
18. Abadi, M., Birrell, A., Harris, T., Hsieh, J., Isard, M.: Dynamic separation for transactional memory. Technical Report MSR-TR-2008-43 (March 2008)
19. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing memory transactions. In: PLDI '06, ACM SIGPLAN Conference on Programming Language Design and Implementation. (June 2006) 14–25
20. Saraswat, V.A., Jagadeesan, R., Michael, M., von Praun, C.: A theory of memory models. In: PPoPP '07, 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (March 2007) 161–172
21. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: PPoPP '06, 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (March 2006) 187–197
22. Adve, S.V., Hill, M.D.: Weak ordering – a new definition. ACM SIGARCH Comput. Archit. News **18**(3a) (1990) 2–14
23. Hill, M.D.: Multiprocessors should support simple memory-consistency models. Computer **31**(8) (1998) 28–34
24. Lev, Y., Maessen, J.W.: Towards a safer interaction with transactional memory by tracking object visibility. In: SCOOL '05, Workshop on Synchronization and Concurrency in Object-Oriented Languages. (October 2005)