

Towards a Safer Interaction with Transactional Memory by Tracking Object Visibility

Yossi Lev
Brown University &
Sun Microsystems Laboratories
Yosef.Lev@sun.com

Jan-Willem Maessen
Sun Microsystems Laboratories
JanWillem.Maessen@sun.com

Mailstop UBUR02-311
1 Network Drive
Burlington, MA USA 01803

ABSTRACT

Lately there has been an increasing interest in Transactional Memory (TM), a programming API that helps programmers writing scalable concurrent programs using sequential code. It is well known that writing concurrent programs using locks is a difficult task: coarse grained locking results in poor performance, and fine grained locking introduces the risk of deadlocking, and makes program's maintenance difficult. With TM, the programmer only specifies *what* is required to be executed atomically, without worrying about the synchronization required to achieve this task.

There are suggested implementations for TM both in hardware (HTM) and software (STM). While HTM is much faster than STM, it also has more limitations than STM does, and therefore it is not likely that we will have a widely available TM that is implemented purely in hardware in the near future. While STM can compensate for most of the HTM limitations, it imposes a new difficulty on the programmer: STM does not allow concurrent access to an object by transactional and non-transactional code: that is, if an object is accessed using regular read and write operations while it is also being accessed by a transaction, the atomicity of the transaction involving this object might break.

Requiring the programmer to keep track of which objects may be involved in transactions will only result in more error-prone concurrent programs. On the other hand, accessing all objects by only transactional code would be safe, but probably not practical as long as we do not have an efficient and robust pure HTM so-

lution. We therefore introduce a new scheme that implicitly tracks object visibility: with our new scheme, objects that might be accessed by multiple threads are automatically guarded by transactions, and therefore are guaranteed not to be accessed by non-transactional code. Moreover, the scheme is transparent to the programmer, and can work with many different implementations of TM, including the lately suggested hybrid TM (HyTM) that uses STM only when HTM fails (or when HTM is not available).

1. INTRODUCTION

Writing scalable concurrent programs is a difficult task, because it requires reasoning about the interaction between different threads. Today's software uses locks to control the synchronization required when objects are accessed by multiple threads. Unfortunately, the use of locks leads to several well known performance and software engineering problems: using coarse grained locking schemes often results in poor performance due to synchronization bottlenecks; using fine grained locking requires careful reasoning about possible deadlocks, and therefore makes software maintenance difficult.

Transactional Memory (TM) [7, 12] is a concurrent programming API that helps writing scalable concurrent programs. Threads communicate by updating shared memory using transactions, which are code sections that seem to be executed atomically: that is, operations from transactions in different threads do not appear to be interleaved. The synchronization required to achieve the atomicity is transparent to the programmers, leaving them only the responsibility of specifying *what* should be executed atomically, and not *how* it should be achieved. We therefore believe that TM will soon be the synchronization method of choice, and indeed future programming languages like Chapel [3], Fortress [1], and X10 [2] provide syntax and compiler support for transactions. Herlihy and Moss [7] showed that TM can be implemented in hardware (HTM) with simple additions to the cache mechanisms of existing processors. While their HTM implementation should support transactions very

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SCOOOL 2005 San Diego, CA USA

Copyright ©2005 Sun Microsystems, Inc. All Rights Reserved.

efficiently, it is not robust: it only supports transactions up to a fixed size; some transactions cannot complete successfully. Moreover, the limitations of the HTM implementation are architecture-specific, which makes writing portable programs a difficult task. Although recent work [11] shows how to solve some of these problems in hardware, it will take time before HTM is *widely available*, and take longer for these *ad hoc* restrictions to be eliminated.

Software TM (STM) implementations [5, 6, 9, 12] are less efficient than HTM; on the other hand, they are available today and do not have most of the limitations of HTM. Moir [10] describes a hybrid TM (HyTM) implementation, which uses STM if HTM fails. While there is a growing consensus that hardware support for TM is desirable, it seems that in practice software will still play a role in many TM implementations.

This paper deals with a fundamental problem of all recent STM implementations: STM cannot guarantee the atomicity of a transaction if an object is accessed concurrently by that transaction and by non-transactional read and write operations. Requiring the programmer to keep track of which objects may be involved in transactions is prone to errors, eliminating the software engineering benefits promised by TM. On the other hand, although STM implementations keep improving, it is not likely that it would be practical to replace *all* regular memory accesses with their transactional version in the absence of robust and efficient HTM.

We therefore introduce a new technique to assist the programmer in writing safe and efficient programs using STM. The idea behind our new solution is simple: any object that is visible to more than one thread should be protected during concurrent access. We therefore assure that all accesses to the fields of such an object are transactional. This guarantees that there will not be concurrent accesses to an object by both transactional and non-transactional code, because all objects that may be accessed concurrently by more than one thread are accessed only by transactional code. On the other hand, objects that are visible to only one thread can still be accessed by regular read and write operations.

Our algorithm tracks the visibility of an object at run-time by storing an *access mode* in the object's header. When a field of an object is accessed, the mode is checked and, if necessary, the field is accessed using a transaction. Our solution has the following properties:

- *Transparency to the programmer:* Our solution is transparent to the programmer. The visibility tracking is done implicitly at run time. Although the programmer may provide hints to the compiler regarding the expected visibility of an object, these are used only for performance optimization and can safely be ignored.
- *Independence of the TM implementation:* Our solution only requires basic functionality likely to be provided by any TM implementation. Also, although our solution is intended for object oriented languages and requires basic object infrastructure, it assumes nothing about the granularity of the

TM support, and therefore can work with both word-based and object-based transactional memory. We therefore believe that it can be used with many different implementations of TM, including existing STM and HyTM.

- *Transparency to the TM implementation:* Our solution can be implemented completely at the compiler level. The compiler replaces the code that accesses a field with the code that first checks the mode of the field's object and chooses whether to use a transactional or non-transactional access accordingly. Therefore the solution does not require any modification of the TM implementation, although we can get some performance benefits if the TM can examine the mode of the objects accessed by a transaction.
- *Portability:* Because the solution is transparent to the programmer, the only tool that needs to be aware of the TM implementation is the compiler that translates the programmer's code and interfaces with the TM support. We can therefore easily port code between different machines as long as they provide the basic TM functionality. Moreover, the compiler can optimize our solution for the specific implementation of TM on the machine: for example, an STM library might include support for reads and writes of transactional data outside a transaction. Similarly, if robust and efficient HTM is supported, the compiler might eliminate visibility tracking and just access all objects using transactions.
- *Low overhead:* Although we introduce a potentially big overhead for accessing fields of local objects, in many cases this cost can easily be optimized away by the compiler. We therefore believe that the overall cost of our solution would be low, especially compared to the alternative of making all memory accesses transactional.

Our proposed solution dynamically tracks object thread escape. Similar techniques have been described elsewhere, particular in the literature on garbage collection; King [8] provides a summary of current work on the subject, of which Domani *et al.* [4] has the greatest similarity to the present work. Many of the possible refinements described in this paper address specific issues and problems in applying these techniques to TM.

The rest of the paper is organized as follows: Section 2 describes our model, including the required TM and language support for our solution to work. In Section 3 we describe our basic solution and how it should be implemented. Next, in Sections 4 – 7, we describe many improvements and optimizations to the basic solution. Our goal is to lay out the design space as thoroughly as possible, laying the groundwork for future exploration in this area. We conclude in Section 8.

2. THE TM AND LANGUAGE MODELS

2.1 The Transactional Memory Support

We assume a TM implementation that supports the following basic operations:

- **Begin** - Starts a new transaction.
- **Read/Write** operations - Used within the scope of a transaction to access memory.
- **Abort** - Abort a transaction. That is, end the transaction, discarding all of its writes.
- **Commit** - Try committing the transaction. That is, end the transaction and make all its writes visible so it seems as if they were done atomically. The commit operation may fail, in which case the result is as if we had called the Abort operation.

For simplicity, our solution makes use of *nested* transactions in which inner transactions are merged with the outermost one. If the **Begin** operation is called by a thread that is already in the process of executing a transaction, then that **Begin** operation is ignored, as is the corresponding **Commit** operation. An **Abort** operation inside a nested transaction aborts the outermost transaction. This form of nesting is easy to emulate if it is not supported by the underlying TM implementation.

2.2 The Programming Language

Note that any implementation of ubiquitous STM or HyTM requires compiler support: every memory access during a running transaction must use transactional **Read** and **Write** rather than ordinary loads and stores. For ease of exposition, we assume that when the compiler encounters a field access, it is translated into a call to an accessor method. The compiler generates an accessor method for every object field; these methods enforce transactional behavior if they are called in the context of a transaction. In practice, the accessor methods are not strictly necessary—a compiler can generate the appropriate code in line rather than inserting an accessor call.

Note that our solution targets object oriented languages in which object access is through an object reference. In particular, we do not protect objects from concurrent accesses done by pointers pointing to the “middle of an object”.

We assume that the programming language exposes the TM functionality to the programmer using *atomic blocks*. With this interface, the programmer specifies a code segment that needs to be executed atomically by wrapping it with the `atomic{...}` notation. The compiler takes care of beginning and ending the transaction, and retrying the transaction if it fails to commit. Finally, nested atomic blocks correspond to nested transactions as described in Section 2.1. Figure 1 demonstrates the use of atomic blocks to atomically increment those values in a list which are larger than some threshold `minVal`.

3. BASIC SOLUTION

We say that an object has *local visibility* if it is only visible to one thread, and that it has *shared visibility* otherwise. For brevity, we refer to objects with local

```
void FetchAndIncrement(Integer I) {
    atomic {
        I.value = I.value + 1;
    }
}

class List {
    class Node {
        Integer value;
        Node next;
    }
    Node head;
    ...
    ConditionalIncrement(Integer minVal) {
        atomic {
            Node currNode=this.head;
            while (currNode != null) {
                if (currNode.value > minVal)
                    FetchAndIncrement(currNode.value);
                currNode = currNode.next;
            }
        }
    }
}
```

Figure 1: The use of atomic blocks: The ConditionalIncrement method of the List class atomically scans the list and uses the FetchAndIncrement function to increment the value stored in each list node if that node’s value is bigger than minVal. Note that the atomic block of the FetchAndIncrement method is nested in the surrounding atomic block of the ConditionalIncrement method.

and shared visibility as local and shared objects, respectively. Trivial examples of local objects are objects that are allocated on the thread’s stack, or objects that were allocated by a thread but are not yet referenced by any shared object. Note that shared visibility is a transitive property: once an object becomes shared, all objects referenced from it, directly or indirectly, become shared as well.

Our solution is based on the following rule: Access to an object that has shared visibility must be done from within a transaction. On the other hand, while it is safe to access all objects transactionally, access to an object that has local visibility can use regular read and writes. This achieves much faster access (especially when STM is involved) and reduces the chance that two transaction will be unable to execute in parallel because of resource limitations in the TM implementation (false conflict). The only problem is ensuring that no object is concurrently accessed by transactional and non-transactional code.

3.1 Naive Static Solution

We might imagine using a type qualifier `local`, similar to `const` in C++, to distinguish objects which always have local visibility from those which might be shared. Objects which are always local would be accessed using ordinary reads and writes; all other objects would need to be accessed transactionally.

Making this distinction in the type system, however,

presents immediate problems in an object-oriented language. Fundamentally, methods must be written assuming that their arguments have a particular visibility (either `local` or not). Once we decide that an argument ought to be `local`, we can no longer pass in a shared object, and vice versa. Similarly, a class whose objects might be shared cannot have any `local` fields. Thus, decisions about visibility are infectious: once we decide that an object can be shared, all its contents must be shared as well.

These restrictions complicate code maintenance. In general, programmers must either restrict objects to be visible to a single thread, limiting code reuse, or must assume that objects are always shared, eliminating the opportunity to use simple loads and stores when sharing does not occur. This is a particular problem if the programmer would prefer to declare a different visibility from that assumed by a base class or by library code.

There may still be a justification for type system support for sharing, which we discuss further in Section 6.2. However, tracking sharing using types is a poor solution by itself. In practice, objects are created and initialized locally even if they are later shared. Some classes will be used in a mixture of shared and local contexts. In order to take advantage of these patterns of use, we track object visibility *dynamically* rather than approximating visibility at compile time using types.

3.2 Our Dynamic Solution

Our solution maintains an *access mode* for each object, for example by adding an extra field to the object header. The access mode specifies whether reads and writes to an object's fields must use transactional code. This is achieved by adding code to the getter and setter methods of the object which checks the object's access mode and, depending on its value, decides whether to enforce transactional behavior.

In its simplest form, our algorithm uses two access modes, *local mode* and *shared mode*, and maintains the following invariant: If an object has shared visibility, then it is in shared mode. We maintain the invariant as follows: when a thread creates an object the object is in local mode. If a reference to a local-mode object is stored in a shared-mode object, the local-mode object must first be *published*. The values of global variables, static object fields, and other globally-visible data must also be published, as does a reference to an object which might be passed between threads using OS support. An object is published by setting its access mode to shared, then recursively publishing all objects it references. The publication action is invoked by the setter methods of the shared mode object, as shown in Figure 2.

As Figure 2 shows, the getter and setter methods of a local-mode object access the object's fields using regular read and write operations. No atomicity is required, since we know the object cannot be modified asynchronously, nor can its status be changed by another thread. Ordinary reads and writes may be used even if these accessor methods are called from within transactional code, since no synchronization is required for accessing fields that are visible to only one thread. Note, however, that if a transaction aborts there is still

```

field_X_getter() {
    if (this.access-mode() == local) {
        return this.X;
    }
    else {
        atomic { return this.X; }
    }
}

field_X_setter(object Y) {
    if (this.access-mode() == local) {
        this.X = Y;
    }
    else {
        if (Y.access-mode() == local) {
            Y.publish();
        }
        atomic {
            this.X = Y;
        }
    }
}

publish() {
    if (this.access-mode() == local) {
        this.access-mode() = shared;
        for each field f of this {
            publish(this.f)
        }
    }
}

```

Figure 2: The new getter and setter methods, and the publish method. Here *X* is the accessed field name, and `o.access-mode()` returns the access mode of object *o*.

a need to revert any modifications to local objects which existed before the transaction began. This does not impose any new requirements upon the TM implementation, however, because the TM implementation should already support transactional access to local variables on the stack, which are treated in the same way. Of course, if no such special support is provided it is always safe to access a local object transactionally from within an atomic block, and still access the object from non-transactional code using simple loads and stores. Finally, note that the special field that stores the object's access mode can always be treated as local (that is, be accessed using ordinary loads and stores). When the object is local, then by definition there is only one thread that can access it. When the object is shared, this field indicates shared mode and it will never be modified again; concurrent accesses to it from other threads will always return the same value.

4. CONTROLLING THE COST OF PUBLISHING OBJECTS

The technique presented in the previous section has the virtue of being simple and correct. However, publishing an object can be an expensive proposition. If an object has many reference fields (for example, it is a large array), each reference must be checked and published if

it is in local mode. Even if an object has a small number of fields, one might refer to a large recursive data structure, such as a long linked list, in local mode.

Publication cost can create additional problems. We are using transactions to control access to shared objects. Publication occurs when a shared object is updated; thus, publication actions occur during the course of a running transaction. This increases the running time of the transaction, in turn increasing the chance that a transaction will be aborted due to conflict. Since by default we are treating the mode as an ordinary local field, publication actions will be rolled back if a transaction fails. However, it is quite likely that, when the transaction is retried, it will publish the same objects. Here we examine some mechanisms to reduce publication cost, or to shift it to a more favorable point in execution, without affecting the basic invariants established in Section 3.2.

4.1 User control over publication

At the most basic level, we can give the programmer a method `share()` to explicitly publish an object if it is not already in shared mode. This method can be used to publish an object before starting a transaction in which it will become shared, or to publish objects before they are stored into a local object (avoiding the cost of later recursively traversing a large linked structure).

A refinement of this is to permit the user to *allocate* certain objects in shared mode. Any value stored into the object's fields will be published, even if those fields are written before the object is actually shared.

4.2 Shared content mode

Rather than publishing an object immediately, we can add a new mode: *shared content* mode. Objects in this mode may be accessed using ordinary reads and writes, as they are local, but any local mode object stored into a shared content mode object must be published. Publishing a shared content mode object is a simple matter of changing its mode to shared—it is not necessary to traverse its fields recursively.

We would have several options in using shared content mode:

- Add a new method, `shareContent()`, which publishes all the fields of an object and places that object into shared content mode.
- When shared allocation is requested, allocate objects in shared content mode. The object can be published immediately after the constructor runs, or publication can be deferred until required by an implicit publication action.
- Place objects containing a large number of references (such as arrays) into shared content mode by default, since the overhead of publication would otherwise be unacceptable. In this case there might need to be a `local()` method to declare to the system that a large object is unlikely to be shared.
- Make shared content the default behavior of the `share()` method, and of shared allocation. The

additional overhead of transitioning to shared mode is small, and will be paid by implicit publication only when necessary.

4.3 Sticky publication

Ordinarily, we expect *all* the effects of an aborted transaction to be rolled back. But should this include publication? A change of mode should not affect the behavior of a program—and as we argued at the beginning of the section, it is quite likely that if a transaction retries, an object which was published will be published again. It may therefore be preferable not to roll back the publication of the object. There are a number of ways we might imagine deciding whether to use this *sticky* publication action:

- Unconditionally use sticky publication.
- Roll back publication if user-level abort occurs (indicating that a different path must be taken on successful retry), but not in case of simple conflict (where control flow on retry is likely to be the same).
- Permit the programmer to specify whether sticky publication will be used for a particular `atomic` block.
- Add modes which require sticky publication for particular objects.

Combining these options yields a rich design space; we expect that a relatively small set of those options will prove worth supporting in practice.

4.4 Defer publication until commit

Ordinarily, transactional reads and writes must be used as soon as an object is published, even if the object is published by transactional code. Yet such objects will not actually become shared until the transaction attempts to commit. It therefore makes sense to defer publication actions until just before committing, and until then use the cheaper local protocol to access these objects.

Deferring publication requires logging publication actions which occur during the transaction. The ordinary transaction commitment mechanism must be augmented to play this log. If the transaction aborts, we can simply discard the publication log, or play it before retrying, resulting in a form of sticky publication.

Some implementations of STM permit other threads to “help” with the commit process. In such a setting, we must be careful to ensure that objects are published before they are visible to non-helping threads. Some optimizations may permit helping threads to manipulate not-yet-published objects (and even share the work of publication), but this will of course depend on the details of the underlying protocol.

5. REDUCING CHECKING COST

The chief remaining drawback with our technique is that we must check the mode of an object before we access it to guarantee that transactional and non-transactional

```

x = r.a
y = r.b
z = y.c
⇒
if (mode(r)==shared) {
  x = transactional read of r.a
  y = transactional read of r.b
  z = transactional read of y.c
} else {
  x = simple load of r.a
  y = simple load of r.b
  if (mode(y)==shared) {
    z = transactional read of y.c
  }
  else {
    z = simple load of y.c
  }
}

```

Figure 3: The compiler can combine and eliminate mode checks.

accesses are not intermixed. Given the overheads of STM, this extra check is relatively cheap for shared objects. However, for an object in local mode it can represent the majority of the cost of a read or a write. Fortunately, the compiler can take advantage of several invariants to amortize this cost over multiple accesses:

- An object in shared mode will remain in shared mode (see Section 7 for an exception).
- Any reference obtained from an object in shared mode will itself be in shared mode.
- An object in any local mode (including, for example, shared content mode) will only change mode as a result of a store by the current thread, or a call to a special method such as `share()`. Note that any local-mode object which might be reachable from an object which is published must be re-checked.

Optimizing mode checking requires us to abandon our simple accessor-based approach to encapsulating these checks. Instead, the compiler must generate the code for mode checking in line and specialize it based on context. For example, in Figure 3 the reads from `r` only check the object mode once, and once we know `r` is shared we know anything it points to, such as `y`, is shared as well. In effect we compile multiple versions of a code fragment which operates on multiple objects, each version specialized to a particular combination of modes of the objects examined. This can even be done interprocedurally, for example by compiling copies of a method specialized to different sharing patterns. We obtain the greatest benefit if the compiler can move a check out of a loop or a recursion.

We can use the results of classic program analyses such as thread escape analysis to prove that certain objects have a particular access mode, eliminating mode checks entirely. Similarly, we can use the results of pointer analysis to prove that publishing one object will not cause another object to be published, thus eliminating the need to re-check a local object’s status after every publication event.

6. PREVENTING UNINTENDED PUBLICATION

Sometimes it is useful to know that an object *cannot* be shared. This will catch concurrency bugs which cause unintended publication of local objects; such bugs are otherwise difficult to debug. But, as described in Section 3.1, any solution to this problem must not require extensive program modifications simply to change object visibility.

6.1 Always local mode

We can add a new mode, *always local mode*, to identify objects which must never be published. Any publication action on an object in always local mode should result in an exception being thrown. Any broadcasting write will be skipped, and the enclosing transaction will be aborted.

An object can either be allocated in always local mode, or be placed in this mode by a method call `alwaysLocal()`; if the object is already shared this call throws an exception. Once in always local mode, the mode cannot be changed again, *e.g.* by calls to `share()` or `shareContent()`. This fact can be used by the compiler when optimizing visibility checks: an object in always local mode need never be checked again.

6.2 Static solutions revisited

In Section 3.1, we described the use of a `local` type annotation. By declaring parameters, classes, and variables as `local`, the compiler can statically check that data is not mistakenly exposed to concurrent update. These annotations provide a mechanism for library code to enforce object locality in its interface. But visibility annotations are pervasive: objects which are `local` can never be intermixed with ordinary objects. To permit `local` data to interoperate with non-`local` libraries, a safe cast between the two is required.

This can be achieved by combining static and dynamic checking. We can provide a method `withLocal()` which takes a `local` object and returns the object with non-`local` type, after ensuring the object is in always local mode. This allows the programmer to safely pass a `local` object to code that can deal with arbitrary sharing. The `alwaysLocal()` method can be given a `local` return type. Together, these methods provide a safe cast between `local` and non-`local` types. The compiler will catch *unintended* mixing of `local` and non-`local` objects, preserving static checking where it is required. At run time, mode checking will guarantee that `local` objects are not shared.

7. RE-LOCALIZING OBJECTS

The solutions we have described thus far share one important property: an object, once shared, remains shared for the rest of its lifetime. We know that in practice some objects are allocated in one thread and handed off to another; such objects will never be referenced by more than one thread at a time. Similarly, objects can be placed into a collection (such as a stack, heap, or queue) and later removed and processed locally by another thread. Once an object is no longer shared, we

can once again use simple reads and writes rather than accessing the object transactionally. In this section we consider some simple techniques for *object localization*.

7.1 Localization by copying

The simplest way to localize an object is to copy its contents to a new object. Most languages provide facilities for copying objects. In Java, for example, the `clone()` method suffices. This new object will be local to the copying thread.

The disadvantage of this straightforward approach is that we pay the copying overhead even for objects which are already local. We can include a method, `localize()`, which returns a copy if the object is in shared mode, and otherwise returns the original object. However, this form of localization can lead to unexpected program behavior. Imagine we localize an object obtained from an aggregate data structure such as an array, then we change the contents of the newly-localized object. If the object is shared, the changes will only be visible in the localized copy; the copy referenced by the array will not have changed. However, if the object is local, the changed version of the object is still visible through the array. We therefore prefer the simpler solution: always copy objects to localize them.

7.2 Exploiting garbage collection

Most garbage collection algorithms are based upon object reachability. We can exploit this fact to localize shared objects. If the garbage collector (GC) can determine that an object is visible to only one thread, that object can be placed back into local mode.

There are several potential pitfalls to GC-based localization:

- Object localization is not timely—we must wait for the GC to run before localization will occur. Thus, the programmer must choose either to pay the cost of a copy and achieve timely localization, or to pay the cost of transactional access until the GC notices the object can be localized.
- Without careful engineering it is possible for the programmer to publish an object by calling `share()` only to have it immediately re-localized by the GC before it can actually be shared. This may require distinguishing programmer-imposed shared mode from the shared mode which results from an automatic publication action.
- GC-based localization violates the invariant from Section 5 that shared objects will remain shared. Thus, a compiler which optimizes mode checks must insert code to re-check the visibility of shared objects whenever the garbage collector might run. On systems which make use of compiler-inserted safe points for garbage collection, this is relatively straightforward; on systems which do not use safe points, we may lose optimization opportunities.
- Care must be taken to avoid localizing objects which are currently being accessed by a transaction. This is safe only if subsequent non-transactional

accesses will not violate the atomicity of the underlying TM mechanism.

8. CONCLUSION AND FUTURE WORK

All STM implementations we know of suffer from the problem that atomicity is not guaranteed in the presence of concurrent transactional and non-transactional access to the same memory location. We introduced a solution that tracks object visibility dynamically. This allows the programmer to use TM without worrying about this issue, and without the need to explicitly manage which objects are accessed transactionally and which are not.

At the moment we do not have a working implementation of any of the techniques we have described. Our implementation of the Fortress programming language will experiment with different points in this design space. It is likely, based on the explorations presented in this paper, that our initial implementation will include a shared content mode, sticky and deferred publication, and simple intra-procedural optimizations of mode checks. We will incorporate some method of preventing publication of shared objects, but have not decided whether a static or a dynamic method will be more appropriate. We are interested in seeing how this solution can be integrated into other existing and future programming languages. Non-OO languages may pose a particular challenge: it is possible, but potentially expensive, to track escape at address granularity rather than at object granularity. Finally, we await the (far-off) day when ubiquitous, scalable hardware transactional memory makes the optimizations described in this paper unnecessary.

Acknowledgments

We would like to thank our colleagues at Sun Microsystems Laboratories, particularly Dave Detlefs and the members of the Scalable Synchronization and the Programming Languages research groups. The feedback they gave helped improve our proposed solution. Special thanks to Mark Moir, for suggesting the idea of preventing unintended publication.

9. REFERENCES

- [1] ALLEN, E., ET AL. The Fortress language specification (version 0.707). Available from <http://research.sun.com/projects/plrg/>, July 2005.
- [2] CHARLES, P., DONAWA, C., EBCIOGLU, K., GROTHOFF, C., KIELSTRA, A., VON PRAUN, C., SARASWAT, V., AND SARKAR, V. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Oct 2005).
- [3] CRAY INC. Chapel specification 0.4. Available from <http://chapel.cs.washington.edu/>, Feb 2005.
- [4] DOMANI, T., GOLDSHTEIN, G., KOLODNER, E. K., LEWIS, E., PETRANK, E., AND SHEINWALD, D. Thread-local heaps for java. In

- ISMM '02: Proceedings of the 3rd international symposium on Memory management* (New York, NY, USA, 2002), ACM Press, pp. 76–87.
- [5] HARRIS, T., FRASER, K., AND PRATT, I. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing* (2002), pp. 265–279.
 - [6] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, W. Software transactional memory for dynamic data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing* (2003).
 - [7] HERLIHY, M., AND MOSS, J. Transactional memory: Architectural support for lock-free data structures. Tech. Rep. CRL 92/07, Digital Equipment Corporation, Cambridge Research Lab, 1992.
 - [8] KING, A. C. *Removing Garbage Collector Synchronisation*. PhD thesis, University of Kent at Canterbury, September 2004.
 - [9] MOIR, M. Transparent support for wait-free transactions. In *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms* (London, UK, 1997), Springer-Verlag, pp. 305–319.
 - [10] MOIR, M. Hybrid transactional memory. Submitted for publication, July 2005.
 - [11] RAJWAR, R., HERLIHY, M., AND LAI, K. Virtualizing transactional memory. In *The 32nd Annual International Symposium on Computer Architecture* (2005).
 - [12] SHAVIT, N., AND TOUITOU, D. Software transactional memory. *Distributed Computing* 10, 2 (February 1997), 99–116.