Reducing STM Overhead with Access Permissions

Nels E. Beckman Yoon Phil Kim Sven Stork
Jonathan Aldrich

January 2009
CMU-ISR-09-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

While transactional memory makes concurrent programming more convenient, software transactional memory (STM) is typically associated with a high overhead. In this work we present a technique for reducing overhead associated with STM using *access permissions*, annotations on method parameters describing how references may alias. This information, which is statically checked for correctness, can be used to eliminate synchronization and logging operations. We have implemented this technique and show that it improves performance on a number of benchmarks.

This work was supported by a University of Coimbra Joint Research Collaboration Initiative, DARPA grant #HR0011-0710019, Lockheed Martin, and a grant from the National Science Foundation (CCF-0811592). Nels Beckman is supported by a National Science Foundation Graduate Research Fellowship (DGE-0234630).

Keywords: Transactional Memory, Optimizations, Permissions

1 Introduction

Transactional memory (TM) is a promising approach to decreasing the complexity of writing multi-threaded, shared-memory applications. Unfortunately, there are some obstacles to the wide-spread adoption of this approach. One obstacle is the relatively large overhead that existing transactional memory systems impose over standard lock-based synchronization. This paper presents a compile-time approach for reducing the overhead associated with software transactional memory systems.

Software transactional memory (STM) systems generally exhibit a high overhead, since they perform synchronization operations on many memory accesses, and keep track of all read and modified memory in separate read, write and undo logs. These logs are then used to detect threads that see an inconsistent view of memory, and to subsequently roll back their memory effects. Researchers have observed that access to thread local [17] or immutable [16] memory locations do not require the same level of synchronization as thread-shared, mutable objects. This insight motivates our approach. Unfortunately, accurate information about which objects will or will not be shared or mutated is not readily available, and is difficult to obtain using static analysis.

In this paper we present a technique for statically optimizing the run-time performance of STM systems using *access permissions*, which provide this aliasing information. Access permissions are static predicates associated with program references, and are provided as annotations by programmers. They describe how objects will be modified and aliased at run-time. Because permission annotations are provided at method boundaries, we can modularly perform the sort of checks that otherwise would require whole-program analysis [11, 17]. Access permissions are used for lightweight behavioral verification [3, 4], and therefore could conceivably already be present in a user's program. We use the static information about program references to eliminate unnecessary synchronization and logging.

This paper makes the following contributions:

1. We present a technique for the compile-time removal of unnecessary synchronization and logging based on access permissions, an existing alias control mechanism.

2. We have implemented this optimization in AtomicPower, a source-to-source implementation of STM based on AtomJava [14] and work by [1]. AtomicPower takes a program written in Java using STM primitives and translates it into an optimized, thread-safe pure Java program.

3. We have evaluated our optimizations on a number of benchmarks, including an open-source video game application. In general performance is improved, and in certain cases greatly improved.

We proceed as follows: Section 2 describes access permissions and their use in lightweight behavioral verification. Section 3 describes the initial implementation of software transactional memory and the optimizations that we perform. In Section 4 we describe our evaluation procedure, our benchmarks, and the results of our optimization. Finally, we discuss related work and conclude.

2 Background: Access Permissions

Access permissions are a static means of controlling aliasing for the purposes of program verification. The system we use was proposed by [4] for the purposes of statically verifying correct usage of object protocols, also known as typestate [18]. Access permissions are similar to other alias control schemes (e.g., ownership [8]) because they restrict the ways in which objects can be aliased. In general, it is extremely difficult to prove behavioral properties about systems with arbitrary aliasing without performing a whole-program analysis. In recent work [3] we extended typestate verification using access permissions to concurrent programs that use atomic blocks as a means of mutual exclusion. That work was the inspiration for our permission-based optimizations.

While a full description of the verification system is outside the scope of this paper, in this section we describe access permissions as they are used to annotate concurrent programs.

Access permissions are predicates that are statically associated with program references. These predicates tell us how the reference with which they are associated can be aliased and modified. They must be provided by the programmer at method boundaries, and as class invariants, but can otherwise be automatically tracked as they flow through method bodies. There are five kinds of access permissions, each of which denotes a different pattern of aliasing for the references with which they are associated:

Unique permission is associated with a reference that points to an object that can only be reached through that single reference. The reference can be used to read and modify the object. This is also known as a linear reference [19].

Immutable permission is associated with a reference that points to an object that many references may point to, but of those references none can be used to modify the object.

Full permission is associated with a reference that can be used to read and modify the object to which it points. Other references may simultaneously exist that point to the same object, but those other references cannot be used to modify the object.

Share permission is associated with a reference that can both read and modify the object to which it points. However, a share permission indicates that any number of other references may simultaneously point to the same object, and some of those references could be used to modify the object.

Pure permission is associated with a reference that can be used to read an object. It differs from immutable because it indicates that other modifying references to the same object, for instance full or share , may exist.

Programmers specify method parameters and the receiver as requiring a certain kind of access permission. Then, at call sites for that method, our static checker will determine whether or not the proper permission is available and associated with the arguments that are passed to the method. Because certain permissions kinds are in a sense, "stronger" than others, it is often possible to call a method that requires a different permission for a parameter than is available on the argument,

using the "splitting" rules of our system. For instance, a unique permission on an argument will satisfy a method that requires a full permission for the corresponding parameter, since knowing that a reference is the *only* reference in a program pointing to some object is more powerful than knowing a reference is the only *modifying* reference.

The following example shows a class meant to hold parameters for a multithreaded benchmark. It is annotated with access permissions. The constructor returns the sole reference to the new object. The benchmark time limit can be changed as long as the caller has the only reference to the object. But one does not need modifying permission to query the limit using the getTimeLimit method.

```
class BenchmarkParams {
  @Perm(ensures="unique(this)"))
  BenchmarkParams() { ... }

  @Unique void setTimeLimit(int t) { ... }
  @Imm int getTimeLimit() { ... }
}
```

In the following code, createThread requires but does not return an immutable permission to its arguments, presumably so that it can store reference in the field of another thread object. While a call to createThread will succeed statically, the subsequent call to setTimeLimit will not, since it requires a unique permission, but the calling context has only an immutable one.

```
void createThread(@Immutable(returned=false)
    BenchmarkParams) { ... }

BencharkParams p = new BenchmarkParams();
p.setTimeLimit(2000);
createThread(p);
p.getTimeLimit(); // Okay...
p.setTimeLimit(500); // Error! need unique(p)
```

Lastly, note that several weak permissions to the same object can be recombined into stronger permissions in a process called "merging." We associate fractions [6] with share , pure , and immutable permissions, which can later be recombined. When the recombination reaches the whole number one, we are allowed to reconstitute a full or unique permission, as the case may be. This is important, for example, in applications where objects are created local to a thread, temporarily shared with other threads, and then later revert to a thread-local state. However this feature was not used to specify any of the benchmark applications in this paper.

3 Approach

In our approach, we developed a source-to-source implementation of software transactional memory and used static access permission annotations to remove unnecessary synchronization and logging. The performance improvements come primarily from immutable and unique permissions,

and to a lesser extent, the full permission. In this section we describe both our implementation of STM and our optimization scheme. Note that the initial implementation of AtomicPower was developed as part of Yoon Phil Kim's master's thesis [15].

3.1 Base Implementation

Our implementation of software transactional memory is a combination of AtomJava [14] and work by [1]. AtomJava is a source-to-source implementation of STM that uses a pessimistic synchronization strategy. It takes programs written in "Java plus atomic blocks" and outputs pure Java source code. We used AtomJava as a starting point, but rewrote much of the internals and runtime system in order to use the synchronization strategy proposed by [1]. While we have attempted to make our implementation as performant as possible, we do not claim excellent absolute performance. Rather, we claim that we can improve relative performance by reducing the number of synchronization and logging operations required. It is our belief that access permissions could help optimize many different implementations of STM, but that the optimization might be slightly different with other design choices.

Our implementation uses an optimistic read, pessimistic write strategy with object granularity. Each object is either owned, or unowned. Unowned objects can be read at will by any transaction, but in order to write an object, a transaction must be the owner of that object, and it remains the owner until the end of the transaction. Writers write to objects in place, and roll back the state of the object in case of transaction abort. We use a version numbering scheme in order to detect possibly-inconsistent reads.

The translation process begins by rewriting every object to (transitively) extend TxnObject which holds a TxnRecord for storing object metadata. The TxnRecord contains both an owner field, telling transactions whether or not the object is owned and by whom, and a version number. Every thread in the program is rewritten to extend TxnThread. TxnThread itself extends java.lang.Thread, but holds a TxnDescriptor object which contains additional data related to a transaction's status. TxnDescriptor holds three thread-local hash maps, one each for the read set, write set and undo log.

Our implementation must also rewrite atomic blocks and memory reads inside transactions. Like AtomJava, we create two copies of each method, the original version and a version to be called inside of atomic contexts. An atomic block is rewritten as a loop that initially calls txnStart, setting the current transaction's status to 'active.' The loop contains a try-catch block whose finally block attempts to commit the transaction, continuing the loop if the transaction commit fails. Field reads (and writes) in an atomic context are replaced with calls to txnOpenObjectForRead (or Write), which obtains the object's TxnRecord and calls txnOpenRecordForRead (or Write), whose implementations are shown in Figure 1. Note that the isOwned method has cost equivalent to a volatile read, and setOwner must perform an atomic test-and-set. logWriteSet performs a whole object copy and a hash table insert, while logReadSet performs just a hash table insert.

We use a polite contention manager [13], and in order to avoid infinitely running transactions due to inconsistent reads we validate the read set by inserting a call to validateReadSet on back edges and method entries. This performs validation one every 1000 calls. Arrays are synchronized on TxnRecords held by a global array, since we cannot force them to extend a superclass of our

```
static void txnOpenTxnRecordForRead(TxnRecord rec) {
  TxnDescriptor txnDesc = getCurrentThreadTxnDescriptor();
  if ( txnDesc.writeSetContains(rec) ) return;
  do { if (!rec.isOwned()) {
         logReadSet(rec, txnDesc);
         return;
       }
       txnHandleContention(rec);
     } while (true);
}

static void txnOpenTxnRecordForWrite(LoggableObject obj, TxnRecord rec) {
  TxnDescriptor txnDesc = getCurrentThreadTxnDescriptor();
  if ( txnDesc.writeSetContains(rec) ) return;
  do { if (!rec.isOwned()) {
         if (rec.setOwner(null, txnDesc)) {
           logWriteSet(obj, rec, txnDesc);
           return;
         }
       }
       txnHandleContention(rec);
     } while (true);
}
```

Figure 1: The implementation of the methods txnOpenTxnRecordForRead() and txnOpenTxnRecordForWrite() in the STM runtime.

choosing. Our run-time system uses the array's hash code in order to index into the global array. This will occasionally cause access to disjoint arrays to be perceived as contention.

There are a few additional helpful features of our STM implementation. It supports the user abort primitive, retry [10] which allows users to perform inter-thread communication. Also, the source language accepted by AtomicPower is actually pure Java. We use Java labels with the text "atomic" and "retry" to delineate atomic blocks and retry statements, respectively. For example, the following method is treated by AtomicPower as having an atomic block and a retry statement:

```
void performNextStep() {
  atomic: { if( !nextStepReady )
              retry::;
          }
}
```

Finally, and in order to make our evaluation more realistic, our implementation performs some basic optimizations on both the base case and the optimized case. We do not open the receiver object for reading on an access to a final field. Additionally, we perform a basic intra-procedural flow analysis to remove redundant read and write open operations on the same object.

3.2 Optimization

In this section we describe a technique for statically optimizing the performance of programs annotated with access permissions. This is the primary contribution of this work. In this section we describe the optimization process, while in Section 3.3 we discuss some of the implications of this process.

The basic intuition behind the optimization is that if the permission associated with a reference indicates that the object is not shared, shared in a limited manner, or immutable , then we can remove "open" operations on that object. This is possible because, with the exception of static fields[1], the access permission associated with a reference is a sound approximation of the thread-sharedness of the object it points to [3].

During program translation, AtomicPower will examine the access permissions that are statically associated with each reference. The process proceeds as follows:

Rule 1 Objects of immutable permission will never be opened for reading, since no thread will change their value.

Rule 2 When writing to the fields of a unique object, it is not necessary to open that object for writing since no other thread can concurrently access the object. However, it is necessary to log the initial value of the object as the transaction still may be rolled back. Therefore, when writing to objects of this permission, a call to the txnOnlyLogWriteObject method is inserted, which logs a copy of the object, but does not perform an atomic test and set on its owner field.

Rule 3 Neither objects of unique nor full permission ever need to be opened for reading.

Rule 4 We would like the above three rules to always be sound. However, because unique and full permissions can be reached through fields of other thread-shared objects, we require that any share , full , or pure object be opened for writing before any method is called on a unique or full field of that object.

The first and third items above will lead to a reduction in the number of synchronizing operations in the resulting translated program, since no check will be performed to query the "owned" status of that object. These items will also lead to a reduction in the number of logging events, since their consistency will not need to be later checked. While logging is a thread-local operation, it does require inserting an item into a hash table. The second item will help to eliminate the synchronization overhead of an atomic test-and-set, which is required when acquiring ownership of an object.

Note that references associated with full permission still must be opened for writing, as other pure references may be used to concurrently read the same object.

In Figure 2 we have illustrated the effect of our optimization on the `contains` method of a linked list. This linked list is used for the buckets of a hash set, which we use as a benchmark and describe in detail in Section 4. Since the list is singly-linked, each element refers to the next with a unique reference. The receiver of the contains method is annotated with the `@Imm`

[1] In our analysis, static fields may only be annotated with pure , share , or immutable .

permission, since it does not perform mutation, but this is okay since the unique permission can be used to satisfy the immutable requirement. The primary difference between the optimized and unoptimized versions of this method are the removal of the call to `__aj_get_value(...)` in the optimized version. This call would normally open `this` for reading, but since we have a unique permission to the list node, we do not require synchronization. Also note that subsequent reads on fields of the receiver do not perform synchronization in either case, because of our basic optimizations.

In the next section we further discuss the ramifications of our changes.

```
@Imm boolean
    contains(@Pure Object item) {
  if( this.value.equals(item) )
    return true;
  else if( next == null ) return false;
  else return next.contains(item);
}
```

```
boolean contains_atomic(Object item)
    throws TransactionException {
  txnPeriodicValidation();
  if (UniqueLinkedList.
        __aj_get_value(this).equals(item))
    return true;
  else if (next == null) return false;
  else return next.contains_atomic(item);
}
```

```
boolean contains_atomic(Object item)
    throws TransactionException {
  txnPeriodicValidation();
  if (this.value.equals(item))
    return true;
  else if (next == null) return false;
  else return next.contains_atomic(item);
}
```

Figure 2: The `contains` method of a linked list, before translation (top), and as translated for use in atomic contexts without (middle) and with (bottom) optimization.

3.3 Discussion

We have presented a technique for optimizing the performance of STM programs using access permissions that will potentially reduce overhead on thread-local, immutable objects, and other

objects that are used in restricted aliasing patterns. However, there are some more subtle points that deserve further discussion.

The first thing to note is that while we can reduce or even eliminate the overhead associated with reading and writing references of immutable or unique permission, those are the sorts of operations that, by themselves, do not need to be performed inside of an atomic block at all. Moreover, our permission checker [3] already tells a programmer statically which memory accesses must and need not be performed inside of a transaction, based on the same static access permissions (essentially obviating the need for strong atomicity). The point is that because of this, we mainly expect to get performance improvements out of unique and immutable objects that are "captured" inside of atomic blocks because of actions being performed on other, thread-shared objects.

The next thing to note is that objects reached through unique references are not necessarily thread-local objects, although we optimize unique object accesses as if they were. For example, an object reachable by several threads through share permissions could have a field that points uniquely to another object. This is the reason why we impose the fourth rule in the previous section. When unique or full objects are accessed via fields of full , share or pure objects, we preemptively open the outer object for writing, giving the current transaction freedom to drop synchronization operations as appropriate. Opening the outer object for writing creates a protected zone around the thread, as illustrated in Figure 3. If we are accessing a full or unique permission through the field of another unique object, we need not open the outer object for writing, since we know the outer object is either a.) only reachable from one thread, or b.) has already been protected with an "open for write" operation somewhere back in the reference chain.

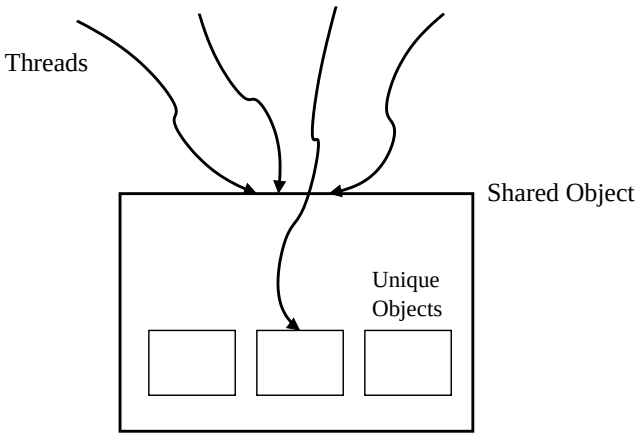

Figure 3: unique and full fields accessed via a thread-shared object are protected since the outer object has necessarily been opened for writing. The single owner thread is free to modify inner, unique objects at will, as other threads attempt to acquire ownership of the outer object.

Occasionally, because of rule 4, our optimization may insert open for write optimizations that were not otherwise necessary. Therefore, we must ask if the potential increase in contention is worth the reduction in overhead. Recent work has suggested that overhead, not contention, is the primary cause of poor performance in STM implementations [9]. For programs that gener-

ally access disjoint regions of memory, the increased granularity will hopefully not matter. We specified our HashSet benchmark (Section 4) twice in order to observe the effect of this increased granularity, and saw that overhead was lowered but contention increased as the number of threads increased. Interestingly, the performance is not dramatically worse, even though the hash set is constantly being accessed by every thread in the program.

Our system has some nice benefits over existing work. Among others, objects stored exclusively as fields of Thread objects can indeed be treated as thread-local. In earlier work, [17] noted that fields of a thread could not necessarily be optimized as thread-local, since a new thread object is always reachable from its spawning context. This reduced their opportunities for optimization. In our approach, the start method on a thread can be specified as consuming the entire unique permission to the thread object. Figure 4 shows just such an example. This prevents the spawning thread from modifying or reading the newly created thread, thus providing us with another opportunity for optimization.

```
class ConsumerThread {
  @Perm(ensures="unique(this)")
  void start() { super.start(); }

  @Unique void run() {
    atomic: {
      Object i = this.input.get();
      doWork(i);
      this.output.put(i);
    }
  }
}

void spawnConsumer() {
  ConsumerThread t =
    new ConsumerThread();
  t.start();
  //Cannot access thread object
}
```

Figure 4: The start method of the ConsumerThread class consumes the entire unique permission produced at construction-time. The result is, ConsumerThread need not be opened when reading and writing its fields inside an atomic block.

Finally, it is interesting to point out that sometimes with our system, a programmer's specification goals may conflict with his performance goals. When writing a method specification for the purposes of behavioral verification, a programmer generally wants to write the weakest pre-condition possible. This will make the method useful in the largest number of contexts. In our system, this means requiring as weak a permission as possible (e.g., pure or share) to the parameters of a method. However, when performing optimization, since we must assume conser-

vatively that references of pure or share permissions are being thread-shared, this may result in under-performance when a stronger permission was available. For example, if the programmer has unique permission to an object, they would like pure method calls on that object to not require any synchronization. This is a natural use for method specialization, since, statically, we can identify the points at which a caller has a stronger permission than strictly required by the method. Creating a copy of that method with reduced synchronization would help improve program performance. While we have not implemented this specialization feature in AtomicPower, we plan to do so in the future. Additionally, some of our benchmarks have been "hand-specialized" in order to take advantage of this observation.

4 Evaluation

In order to evaluate our technique, we have used our optimizations on a suite of annotated benchmarks of varying sizes and we compared those results to our baseline implementation. In this section we describe the results of these benchmarks. We also describe our experiences specifying these concurrent programs, and report on interesting patterns.

4.1 Methodology

For the purposes of evaluation, we chose several benchmarks, consisting of microbenchmarks, popular STM benchmarks, and an open source Java video game. For programs that were not originally written to used atomic blocks, we replaced existing synchronization constructs. Then, we used access permissions to specify as many of the methods and classes as possible, in order to describe the program's aliasing behavior. This required a good understanding of each program's run-time behavior. After specification, we used NIMBY [3], our static permission checker, to check the consistency of our specifications. This process verifies that the access permissions we wrote were actually correct, with respect to the aliasing behavior of the program. While the primary goal of NIMBY[2] is to check typestate behavioral properties, we did not specify any for the purposes of this experiment. Figure 5 describes the number and type of full , unique and immutable permissions that were used in each benchmark, since these permissions are the ones that provide performance benefit. For the largest benchmarks, we did not specify all of the references in the system. Specifically, we ignored methods and objects that were never used in transactions and we did not specify methods of pure or share permission that did not interact with other permissions in meaningful ways. Since the default reference annotation is share in our system, this is sound.

After permission verification, we took each benchmark and ran it through AtomicPower, our source-to-source translator, with and without our permission optimizations turned on. For each benchmark, our optimization removed a different number and type of call into the STM runtime system. Figure 5 describes the number of calls in source code that were statically removed for each benchmark, as well as the number of additional open for read calls that were inserted. However, in

[2] `http://code.google.com/p/pluralism/`

Benchmark	Refs. Annotated			Open Calls Removed (Total)		Extra OW
	immutable	unique	full	read	write	Calls Inserted
4InALine	124	23	1	41 (289)	8 (100)	1
HashSet	0	4	0	1 (16)	0 (5)	1
ListSet	0	5	0	4 (19)	2 (18)	0
ReadHeavy	2	2	0	1 (1)	0 (0)	0
WriteHeavy	0	4	0	0 (0)	1 (1)	0

Figure 5: Number of references annotated with helpful access permissions, and the number of open for read/write calls this removed. The last column lists the number of additional open for write calls inserted due to rule 4.

general the removal and insertion of STM operations at different locations in the source program will have a disproportionate effect on overall benchmark performance.

In general our STM implementation is not sound if it is not used to translate every file in an application. However, some of our benchmarks used classes from the Java standard library. While many of these classes could be translated from source, some could not due to bugs in our source to source translation (primarily due to use of anonymous inner classes and some features of Java generics). In a few cases we created new implementations which could more readily be translated by AtomicPower. In such cases we attempted to be as faithful as possible to the original implementation.

Each benchmark has its own measure of performance, usually elapsed time or number of operations performed. We ran each with and without optimizations for 1000 runs (unless otherwise noted), varying the number of threads when appropriate.[3]

We will now briefly describe each of the benchmarks we used.

ReadHeavyTest and WriteHeavyTest: In order to get a feel for the potential of our optimization, we created two synthetic benchmarks, ReadHeavyTest and WriteHeavyTest. Both programs access objects inside of a transaction, but do so with only a single thread. ReadHeavyTest creates a chain of objects, each of which refers to the next with immutable permission, and then inside of a transaction reads from fields of every object in the chain. The entire process is performed 1000 times inside of a loop, and was designed to illustrate the effect of removing an open for read operation. WriteHeavyTest is the same, except that each object in the chain refers to the next object with a unique permission, and during the transaction each object in the chain is modified. This benchmark was designed to give us a feel for the amount of overhead that can be reduced when removing the ownership acquire operation, but retaining the object copy operation. For comparison purposes, and because unique and immutable references do not need to be accessed inside of a transaction, we also ran the same two experiments without any synchronization.

FourInALine: We wanted to evaluate our optimizations on a real program representative of com-

[3]Note that all of our performance numbers come from executing programs on a Dell PowerEdge 2900 III with 2 Quad Core Intel Xeon X5460 processors, running at 3.16GHz (1333MHz FSB) with 2x6MB of L1 cache, 32 GB of RAM, and running Linux 2.6.23.1-001-PSC and Sun's Java SE Runtime Environment (build 1.6.0_07-b06).

mon multi-threaded OO programs. For this purpose, we chose FourInALine[4], a GUI-based video game that is a clone of the board game Connect Four. We chose this program because it was relatively large (5471 loc in 62 classes), it was well designed and documented, and seemed at first glance to contain a number of immutable and thread-local objects that were being accessed inside of critical regions. FourInALine stores shared game data in a server object that is accessed by client threads, one per each player in the game, and by a GUI update thread. These threads will each take a copy of the current game board, which they use to either calculate a next move, or to determine the visual representation of board.

FourInALine required some modification before it could be used as a benchmark. We replaced synchronized blocks with atomic blocks (57) and a retry statement (1). This program uses JFrame, a Swing framework class which allows users to create GUI windows. We created a wrapper class that would be introduced as an intermediary by AtomicPower. This wrapper ensures that user subclasses of JFrame will be properly synchronized without requiring us to translate large portions of the Swing framework. In practice, this translation strategy worked well, resulting in a program without flickering or obvious synchronization defects.

For the experiment, we ran FourInALine in a deterministic AI versus AI game on the weakest difficulty level, and gathered the elapsed time from game start to completion.

ListSet: ListSet is an STM benchmark from a paper by [12]. It is an implementation of a List. This benchmark is interesting for our purposes because it creates local objects inside of transactions that escape from their allocation context and are later accessed, but are not shared with other threads. Note that each node does *not* have a unique pointer to the next node, as one might expect of a singly-linked list. Therefore the entire backbone of the list is annotated with share permissions. For our benchmark, we created a number of threads and then measured the total number of insert/remove/contains operations those threads could cumulatively execute during two seconds. Each thread performed 30% updating operations.

HashSet: We created our own implementation of a hash set for benchmarking purposes. In this implementation, the hash set holds an array of bucket nodes that each point to a linked list. Inside the linked list, each node points to the next with unique permission. The top hash node object, however, points to each node with share permission, so that it will not become a contention bottleneck. In order to evaluate the effects of rule 4, which may occasionally insert extra open for write operations, we also specified a "high contention" version of the same program. In this version, the outer hash set object points to its buckets with unique permission. This will eliminate synchronization internal to the data structure, but will effectively serialize access to it. For this benchmark, we created a number of threads and made each perform 100000 operations, 30% of which were updating. We measured the elapsed time.

4.2 Results and Discussion

The results of our benchmarks are shown in Figures 6 through 9. In general, our optimizations improved performance, although to varying degrees. Most improvements can be attributed to unique and immutable references.

[4] `http://code.google.com/p/fourinaline/`

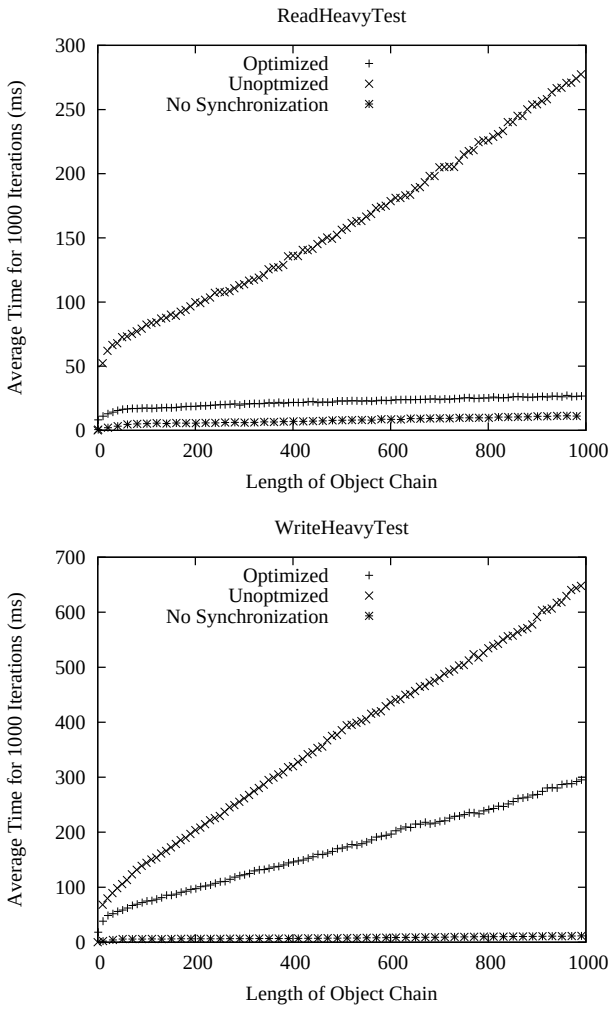

Figure 6: The results from running ReadHeavyTest and WriteHeavyTest (less is better).

The results from the ReadHeavyTest and the WriteHeavyTest (Figure 6), show that there is potentially a great deal to be gained by optimizing access to unique and immutable objects. In particular, removing the open for read operation provides a big benefit, since this makes a memory read essentially free. The synchronization-free benchmark is always faster even for the read-only case, since there is some overhead associated with starting and committing the 1000 transactions that are performed during each run.

The performance of ListSet (Figure 9) was improved because it uses a number of thread-local objects that happen to be trapped inside of atomic blocks. ListSet creates a Neighborhood object on each lookup. This object escapes its allocation context, but is immediately used by the caller, which is still inside a transaction, to determine the result of a search. This process happens once per operation.

However, in our system, objects do not have to be thread-local to be optimized. Uniquely

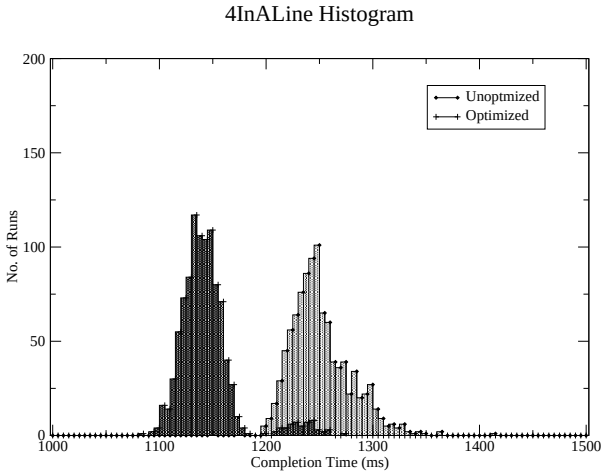

Figure 7: Histogram of completion times for 4InALine (left is better, x axis begins at 1000).

referred objects can still be part of a thread shared data structure, such as the bucket lists in the HashSet benchmark (Figure 8). Because the randomized inserts, contains and remove operations generally hash to different buckets, threads do not contend, and therefore the overhead that is saved because the entire linked list is being locked once at the head results in better performance. Furthermore, note the large standard deviation for the unoptimized case. We speculate that this is due to transaction aborts, which are generally expensive. Because the buckets are locked at the front, aborts are extremely rare in the optimized case, but can occur in the unoptimized case, where a thread may traverse the list, have it modified behind it, and then be forced to abort since its read set is now out of date. For our high contention specification, as expected overall performance is better for smaller numbers of threads, since almost all synchronization operations will be removed, but degrades as more threads attempt to access the data structure and the single lock becomes a bottleneck.

4InALine (Figure 7) benefits from its use of a number of immutable objects. There are many pieces inside the model (which itself is thread-shared and mutated) that are never modified, and therefore numerous reading methods, such as calls to equals, are sped up. Also, 4InALine uses a number of immutable collections, such as a cache for storing lines that are known to be winning lines. Each line is implemented as an immutable list of immutable pieces, although to take full advantage of immutability, we had to perform hand-specialization, copying certain methods and re-specifying them as taking an immutable receiver.

5 Related Work

There has been much previous research attempting to optimize the performance of software transactional memory and to reduce its overhead.

For instance, work has been done in statically identifying objects that were allocated inside of a transaction using a whole program analysis [11, 1]. [17] use a whole-program alias analysis in

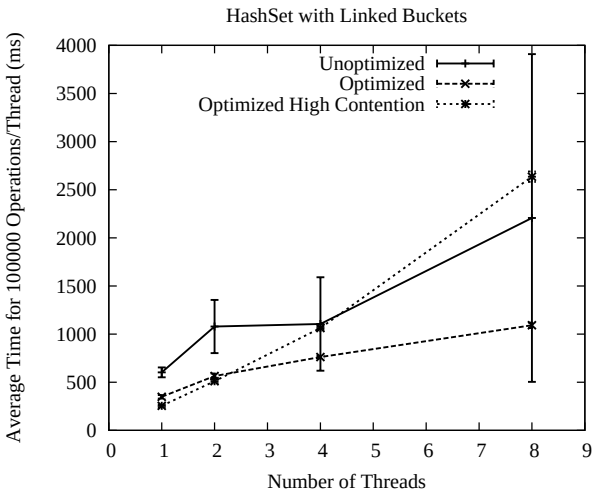

Figure 8: Mean completion times for the HashSet benchmark for different numbers of threads, using 30% modifying operations (less is better). Note the large standard deviation for the unoptimized case.

order to identify objects that are never accessed inside of a transaction, and additionally perform a dynamic escape analysis in order to find thread-local objects. [2; 5] and [7] also perform an interprocedural analysis in order to identify synchronization operations that can be removed, although not in a TM context.

Our work is different in a few ways. First, all of our optimizations are performed statically. Most importantly, our approach is modular, and uses only intra-procedural analysis. This is feasible because of the static access permissions which are provided by programmers, but checked for correctness. This may make it easier for our approach to scale to very large applications. Moreover, our approach is consistent with a language that uses dynamically linked libraries. As long as the code that we link against has been annotated, or we can do so externally, the optimizations we perform on our own code will be sound. Our analysis is sometimes more precise than existing approaches, because the designer's intent is encoded in the annotations. As mentioned in Section 3.3, we do not have to assume that a thread object is thread-shared just because it is reachable from its spawning context.

6 Conclusion

In this paper we presented a static technique for reducing the overhead of software transactional memory based on access permission annotations. Access permissions are a modular system for describing the ways in which a particular reference may alias other references. This information allows us to remove unnecessary synchronization and logging operations that traditionally require a whole-program analysis. Moreover, access permissions have been used for behavioral specification of programs that use atomic blocks [3], so that programmers willing to use our behavioral specifications will get performance increase without additional effort. We have implemented our

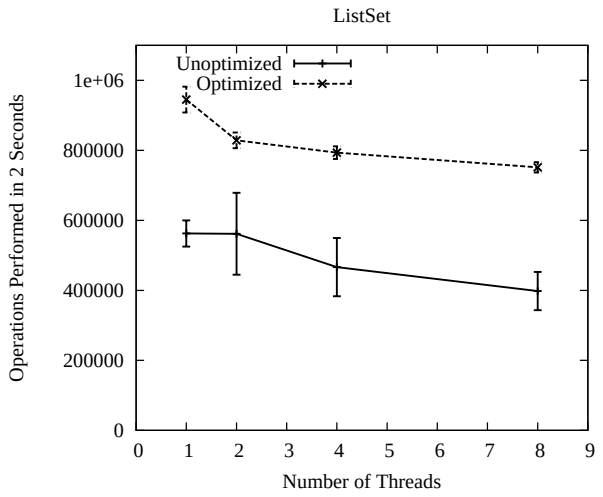

Figure 9: Mean number of total operations performed in 2 seconds in the ListSet benchmark, using 30% modifying operations (more is better).

technique in a tool called AtomicPower, and showed improved performance on a number of benchmarks.

References

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *The 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37. ACM Press, 2006.

[2] Jonathan Aldrich, Emin Gün Sirer, Craig Chambers, and Susan J. Eggers. Comprehensive synchronization elimination for java. *Science of Computer Programming*, 47(2-3):91–120, 2003.

[3] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *The 2008 Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 2008.

[4] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *The 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 301–320. ACM Press, 2007.

[5] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *The 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM Press, 1999.

[6] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[7] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *The 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM Press, 1999.

[8] David G. Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *The 15th European Conference on Object-Oriented Programming*, pages 53–76. Springer-Verlag, 2001.

[9] D. Dice and N. Shavit. What really makes transactions faster? In *Proc. of the 1st TRANSACT 2006 workshop*, 2006.

[10] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *The tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM Press, 2005.

[11] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006.

[12] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *The 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262. ACM Press, 2006.

[13] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *The twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.

[14] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *The 2006 workshop on Memory system performance and correctness*, pages 82–91. ACM Press, 2006.

[15] Yoon Phil Kim. Permission-based optimization for effecient software transactional memory. Master's thesis, Carnegie Mellon University, 2008.

[16] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In *The 2002 joint ACM-ISCOPE conference on Java Grande*, pages 202–211, New York, NY, USA, 2002. ACM Press.

[17] 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. *SIGPLAN Notices*, 42(6):78–88, 2007.

[18] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[19] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359, 1990.