LockSTM: Mixing Locks and Transactional Memory

Nicholas Kuilema
University of Maryland, College Park
nicnak@cs.umd.edu

Abstract

Atomic sections are a technique used to simplify the writing of parallel programs. Actions performed during the execution of an atomic section should not be visible to other components in the program until after the section completes. Usually the atomicity requirement is enforced through optimistic transactional memory. However, transactional memories have the drawback that not every action is allowed to occur inside of an atomic section. In particular, I/O actions cannot be rolled back on most systems, and so they may not appear within transactions.

This paper examines supporting atomic sections that can contain arbitrary actions, using a combination of locks and transactions. We present a system that infers a locking strategy, identifies problematic atomic sections and creates a hybrid solution.

This paper presents the algorithm, argues the process is sound and reports experimental results.

1. Introduction

The computing world is going parallel in a big way with the advent of multi-core chips in most new consumer desktops and laptops. However, current methods of writing parallel code are cumbersome and difficult to get correct. This has led to a dearth in the number of software developers that have the ability to create parallel code. Easier techniques for creating safe concurrent programs are needed to help drive the industry forward. The atomic section is one such simplifying technique. An *atomic section* is a portion of code that runs as if in isolation, neither affecting external actors, nor being affected by them. Atomic sections can be created by grouping interactions with shared variables into logical components. For example, checking to see if a list contains elements and then removing an element would be a single logical component. By wrapping those two actions in a atomic section, the developer is guaranteed no other thread will modify the list between the actions.

One way of enforcing atomic sections is with exclusive locks. Course grained locking is easy to implement, but it limits parallelism. Fine grained locking increases parallelism but it introduces issues like deadlock. Locking is also not composable.

Typically atomic sections are discussed as running optimistically with Transactional Memory (TM). A TM can be either hardware or software based. These HTM and STM solutions are a good way to implement atomic sections, but they fail when calls to a non-revocable action are in an atomic section. A *non-revocable action* is any action that once undertaken cannot be undone by the TM. Examples can range from an ATM spitting out money to simply printing to a console. Because TMs cannot undo the results of such actions, those actions are either forbidden or delayed until the section completes. TMs also have the benefit of allowing as much parallelism as fine grained locking and do not suffer from composability problems.

This paper presents a method of combining transactions with locks to allow a high degree of concurrent execution, while allowing arbitrary actions within an atomic block. Each atomic section is categorized either as a non-revocable section, a revocable section that can conflict with non-revocable sections, or a revocable section that only conflicts with other revocable sections. Atomicity of non-revocable sections is enforced by locks. All the other sections' atomicity is enforced by an STM and additional locks to prevent conflicts with the non-revocable sections.

A static analysis based lock inference algorithm, LockPick, is used to create a graph that indicates possible conflicts between sections. An additional analysis is performed on each atomic section to determine if it is revocable or not. These results are combined in LockSTM which performs a source-to-source translation to create a hybrid lock/STM program.

LockPick and LockSTM are implemented as CIL [15] modules. They act as source-to-source translators on C programs with atomic annotations. They use a sound alias analysis to produce conservative but safe results.

LockPick and LockSTM were run on a variety of benchmarks to show the effectiveness of the method. Experiments include a micro-benchmark to show the concept works, an STM testing suite (STAMP) and a real world web caching sever, memcached. The analysis results show that given an appropriate application the technique can show a large performance gain. However a majority of examined applications did not show improvement. Performance could be improved by a more precise analysis on interactions between atomic sections. A larger improvement should occur by using an STM implementation that allows memory allocation within atomic sections. Despite the current limitations, the result is promising and may lead to greater performance in the future.

2. Atomic Sections

An atomic section is a portion of code that runs as if in isolation. In a single threaded application, this is always the case. For example, take the function that swaps two integers shown in Figure 1. In the single threaded program, the instructions on lines 3, 4 and 5 occur in order, and no other actions can occur between them that would violate atomicity of `swap`.

In multi-threaded programs this is not the case. If multiple threads call `swap` at the same time and there is no synchronization between threads, then it is possible for the threads to interact in a way that violates sequential consistency. *Sequential consistency* was first defined by Lamport [13] as "... the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

The following chart shows how the interleaved actions of two threads could violate sequential consistency. Two sequential runs of `swap` should result in both variables retaining their original values. Instead, this interleaving causes the value of `X` to be overwritten. This happens because thread 2 starts the swap before thread 1

```
void swap()
{
  int tmp = X;
  X = Y;
  Y = tmp;
}
```

Figure 1. swap function

finishes. Then thread 2 reads an updated value for X and an non-updated Y.

Time	Thread 1	Thread 2	X	Y
0	swap()	swap()	1	2
1	int tmp = X;		1	2
2	X = Y;		2	2
3		int tmp = X;	2	2
4		X = Y;	2	2
5	Y = tmp;		2	1
6		Y = tmp;	2	2

Different interleavings can produce all four different combinations of X and Y. But by making swap atomic, all of the bad interleavings will be prohibited and sequential consistency can be achieved.

Enforcing the atomicity of swap() can be done pessimistically by acquiring a mutual exclusion lock so only one thread can execute swap() at a time. Or, it can be enforced by a TM that optimistically allows both threads to proceed. At each function return, the TM detects if an atomicity violation has occurred. If there was a violation, the effects of the section are undone and it is rerun until it completes without violating atomicity. Violation detection is discussed in Section 2.3. Each approach has benefits and drawbacks.

Locking has the drawback that it limits possible concurrency. Generating fine grained locking strategies can alleviate that issue, but it can require a complex implementation which can introduce other problems like deadlock. Locking also introduces composability problems.

STMs have the benefit of maximum concurrency but they can be slower than using locks. Because conflicts and roll backs may occur at any frequency, execution times can vary widely. In addition, STMs cannot be used on an atomic section if the section contains a non-revocable action. Because any section can be rolled back with an STM, it is possible that the non-revocable section could need to be undone. The external action cannot be undone, so the STM method simply fails. Most STMs forbid non-revocable actions within atomic sections. Others allow them, and when a rollback occurs the non-revocable action is run again each time. Rerunning the action is usually not desired because it is a change in program behavior.

In this paper a non-revocable action can usually be thought of as an interactive event with an actor outside the visible scope of the STM. Examples could be a remote procedure call, or interaction with a user. It could even be a call to an existing software library that has unknown effects and therefore cannot be rolled back.

Atomic sections are not a cure all that solves every difficulty existing with parallel programs. Atomic sections do not prevent livelock or priority inversion, and they still rely on a programmer correctly specifying them.

2.1 Atomicity Enforcement with Locks

There are multiple ways to enforce atomicity with locks. Figure 2 shows a running example that helps to explain the various methods of enforcing atomicity. All six atomic sections are shown. A main function spawning multiple threads that call the atomic functions has been omitted for brevity.

```
int x[100];
int y[100];
int z[100];

atomic void PromptForX(int i)
{
  printf( "%d", x[i] );
  scanf( "%d",  x+i );
}

atomic void SwapXY(int i, int j)
{
  int tmp = x[i];
  x[i] = y[j];
  y[j] = tmp;
}

atomic void SetY(int i, int v)
{  y[i] = v;  }

atomic int GetY(int i)
{  return y[i];  }

atomic void SetZ(int i, int v)
{  z[i] = v;  }

atomic int GetZ(int i)
{  return z[i];  }
```

Figure 2. Atomic Functions from a Sample Program

The example has three shared arrays of integers. There are four atomic functions that get and set values from the y and z arrays. SwapXY is basically the same as the swap function above. PromptForX displays the current value to the user and replaces it with the value the user enters.

The simplest way to enforce atomicity with locks is to use a single global mutex lock. At the beginning of each atomic section it is acquired, then it is released at the end of the section. This solution is guaranteed to be correct because only one atomic section can run at a time. Therefore the sections cannot interact in any way that would violate atomicity. The drawback to this method is it severely limits concurrency. For example, GetY cannot run at the same time as GetZ even though it would be perfectly safe for them to do so. Using a single global lock is trivial to implement automatically.

At the other extreme, we could use one lock for each memory location. In this example there are 300 shared memory locations, one for each element of each shared array. For example, SwapXY using per-memory location locks could appear as:

```
void SwapXY_atomic(int i, int j)
{
  lock( x_locks[i] );
  lock( y_locks[j] );
  SwapXY(i,j);
  unlock( y_locks[j] );
  unlock( x_locks[i] );
}
```

This fine grained locking approach allows for maximum parallelism, but it can cause additional problems. If there were a function ClearX() that set every element in array x to be zero, it would need

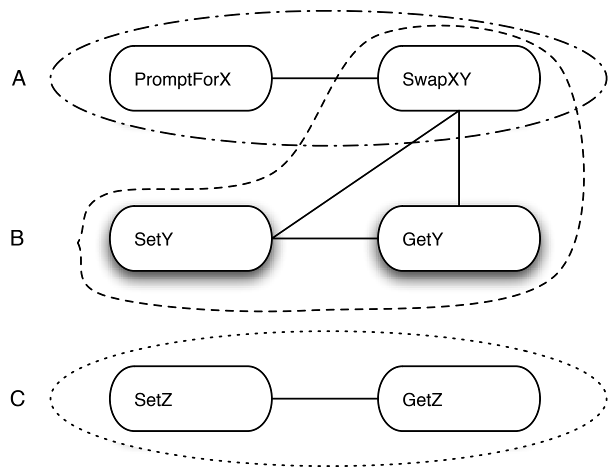


Figure 3. LockPick graph for Figure 2

to acquire 100 locks, set the variables to zero, then release the 100 locks. This can introduce a large time overhead and the additional memory constraints of requiring a large number of locks.

Most programmers take a middle ground by grouping similar memory locations together, then using one lock per group. One intermediary solution for the example would be a lock for each of the three arrays. This step is often the source of bugs in programs when programmers get confused which locks protect which locations.

2.2 LockPick

LockPick [16] is a system that uses a whole program static analysis technique to ensure synchronization of programmer specified atomic sections. In short, it will derive a set of locks that soundly enforce atomicity.

LockPick can be run in two modes, a slower but more accurate context-sensitive mode and a less resource intensive but less accurate context-insensitive mode. Both modes are safe, but the insensitive mode is more likely to incorrectly alias memory locations. This will result in a more constrictive locking strategy.

LockPick operates in four steps. First, an alias analysis is performed to determine which memory locations can be associated with each variable. Second, a shared variable analysis is performed to identify which variables are accessed by multiple threads. Third, a graph is constructed with the atomic sections as nodes. An edge exists between two nodes if they access a common memory location. Finally, the graph is searched for cliques. Each clique is assigned a color that represents possible interference between its nodes. Atomicity can be enforced by representing each color with a mutex lock the transformed program.

Running LockPick on the example in Figure 2 generates the graph shown in Figure 3. There are three colors `A`, `B`, and `C`. Each color guards all of the memory locations in one of the arrays. All six atomic functions are represented in the graph because they all access shared locations. A connection between two nodes means they share memory accesses and should not be executed together. The dashed lines enclosing the cliques indicate which nodes belong to each color. `SwapSY` has two colors: `A` and `B`. In complex programs, sections can have many different colors. If a section contains accesses to shared variables, but does not conflict with any other section, it will have its own color.

The colors are then translated into exclusive locks and inserted into the code as shown in Figure 4. The colors are represented by

```
MutexLock A, B, C;

atomic void PromptForX(int i)
{
  exclusive_lock( A );
  printf( "%d", x[i] );
  scanf( "%d",  x+i );
  unlock( A );
}

atomic void SwapXY(int i, int j)
{
  exclusive_lock( A );
  exclusive_lock( B );
  int tmp = x[i];
  x[i] = y[j];
  y[j] = tmp;
  unlock( B );
  unlock( A );
}

atomic void SetY(int i, int v)
{
  exclusive_lock( B );
  y[i] = v;
  unlock( B );
}

atomic int GetY(int i)
{
  exclusive_lock( B );
  int tmp = y[i];
  unlock( B );
  return tmp;
}

atomic void SetZ(int i, int v)
{
  exclusive_lock( C );
  z[i] = v;
  unlock( C );
}

atomic int GetZ(int i)
{
  exclusive_lock( C );
  int tmp = z[i];
  unlock( C );
  return tmp;
}
```

Figure 4. LockPick generated code from Figure 2

global locks. At the beginning of each function, the required locks are acquired, and they are released before the function returns. The locks have a total ordering and are acquired in order so deadlock cannot happen. Temporary local variables are created to correctly propagate return values outside of the locked section, as shown in `GetY` and `GetZ`.

2.3 Atomicity Enforcement with STM

Using an STM to protect the example code is much simpler than the lock based solutions. No reasoning about relationships between atomic sections is needed.

An STM works by logging each memory read and write that occurs in the atomic section. The writes optimistically happen under the assumption that the sections will likely commit without a problem. When a section attempts to complete, the STM looks through the log and checks to see if there is a atomicity violation. The violation check involves looking at the combination of the reads and writes performed by all the atomic sections currently executing. All of these must be examined because the STM has no advance knowledge of possible interactions between atomic sections.

If a violation is noticed, one of the two sections that conflicted will be rolled back. Rolling back means every memory location written to will have its original value restored and the computation will be restarted. Some STMs have additional strategies to improve performance, like reverting to a pessimistic approach if too many rollbacks occur.

A partial STM transformation to Figure 2 is quite simple. However because STMs are an emerging technology there is no consensus on syntax. When using the TL2 [4] STM, `SwapXY` would become:

```
void SwapXY(int i, int j)
{
  TM_BEGIN();
  int tmp = TM_SHARED_READ(x[i]);
  TM_SHARED_WRITE( x[i], TM_SHARED_READ(y[j]) );
  TM_SHARED_WRITE( y[j], tmp );
  TM_END();
}
```

Each read and write must be converted into a call to the STM so that it can insert the appropriate logging and checks. For comparison, the Intel STM prototype compiler [12] uses a new keyword and automatically detects shared variables. The Intel transformation would be:

```
void SwapXY(int i, int j)
{
  __tm_atomic{
    int tmp = x[i];
    x[i] = y[j];
    y[j] = tmp;
  }
}
```

However, a full STM transformation cannot be performed on the example. No STM can be properly used on `PromptForX` because it contains a non-revocable action. It prompts the user for a new value then stores the value back in the array. If the STM tried to roll it back and re-prompt the user, the behavior of the program would have been changed. Because a single atomic section cannot be STMed, the traditional STM method cannot be used.

3. Combining Locks with STM

The example in Figure 2 provides a compelling case for wanting to use both locks and transactions. The increased parallelism of the STM is desired, but the program cannot be STMed.

Using locks on atomic sections that STMs cannot function on and using the STM on the remaining sections should increase performance. The difficulty arises when trying to ensure the locked sections and STMed sections do not interfere with each other. This paper does not attempt to use fine grained locking for the solution, but rather proposes a solution that fulfills the following three requirements.

1. Only one instance of `PromptForX` should be allowed to run at a time.
2. `PromptForX` should not be allowed to run in parallel with `SwapXY`.
3. Every other combination should be allowed.

Each atomic section can be identified as being one of three types. *Non-revocable sections* are sections that contain any non-revocable code. *Hybrid sections* contain only revocable code, but could conflict with non-revocable sections because of shared memory accesses. *Pure STM sections* contain only revocable code and cannot conflict with non-revocable sections.

First, a coloring is obtained from a LockPick analysis on the program. Second, the non-revocable sections are located by examining their call graph and looking for calls to non-locally defined code. This includes standard library calls such as `printf`() and `scanf`().

By using the coloring atomic sections can be separated into the hybrid and pure classes. Any section that shares a color with a non-revocable section becomes hybrid. All of the remaining sections are labeled as pure.

The pure STM sections are guarded by the STM only. The non-revocable sections are guarded using mutually exclusive locking for each of the colors that LockPick assigned. The hybrid sections are guarded by both the LockPick assigned colors and the STM.

If mutex locks were used to enforce this arrangement, requirements one and two would be accomplished, but requirement three would not be. Referring to the LockPick coloring in Figure 4, we can see that `PromptForX` and `SwapXY` share the color `A`. The mutex lock would forbid multiple copies of `SwapXY` from running concurrently. This can be fixed by using shared access locks.

Shared access locks are a locking construct that allows multiple callers to acquire access at the same time (read mode) or allows a single caller to acquire exclusive access (write mode). The most common form of a shared access lock is the reader/writer lock.

Non-revocable sections will acquire exclusive access to a lock for each color LockPick assigned to them. Hybrid sections will acquire shared access to a lock for each color they share with a non-revocable section.

Thus the final algorithm for using locks with an STM can be expressed fairly simply. NRC is the union of all the colors applied to non-revocable sections. Lines 4 and 5 initialize this set. `ColorsOf` is a function that returns the set of colors that LockPick derived for the section. Non-revocable atomic sections are transformed at line 12. Hybrid and pure STM sections are transformed at line 14.

```
 1.  Perform LockPick coloring algorithm
 2.
 3.  NRC = $\emptyset$
 4.  foreach Non-revocable section S
 5.     NRC = NRC $\cup$ ColorsOf( S )
 6.  foreach C $\in$ NRC
 7.      Create a SharedLock for C
 8.
 9.  foreach Atomic Section S
10.     C = ColorsOf( S ) $\cap$ NRC
11.     if nonRevocable( S )
12.         Protect S with exclusive access for C
13.     else
14.         Protect S with STM and shared access for C
```

This algorithm is referred to as LockSTM. The result of its transformation on the example from Figure 2 is shown in Figure 5. This transformation uses the Intel style STM notation. There are two important things to notice about the transformation. First, in `SwapXY` the lock is acquired outside of the transaction. This is required because acquiring a lock would be a non-revocable action for the STM.

The second thing to notice is the lack of locks for the colors B and C. They were eliminated in line 10 of the algorithm when taking the intersection with the non-revocable colors. The use of the NRC set is not strictly necessary, but is done for efficiency reasons. If the step were not done, then the functions SetZ and GetZ would acquire a shared lock for color C. But in the new source code no function would ever attempt to acquire exclusive access to the color C. If there is no exclusive access, the request for shared access will never block. Therefore, the acquires do nothing except cause overhead.

This color reduction ensures that pure STM sections will never acquire any locks. It also means hybrid sections may acquire fewer locks than they did under the LockPick solution.

As the following argument shows, the LockSTM algorithm produces a correctly synchronized program. The coloring that LockPick produces has been shown to be correct in its initial paper [16]. All the colors assigned to the non-revocable sections are retained and are used to ensure safety on those sections. All of the remaining sections have safely being enforced by the STM. Therefore every atomic section is protected from interference by every other section, and the resulting program is safe.

4. Implementation Details

LockPick and LockSTM are written as modules in the CIL framework. They build off LockSmith [17]. LockPick and LockSTM work as source-to-source translators producing C code that should work with any compiler. The two modules are about 1,000 lines of OCaml and make extensive use of the 12,000 lines in LockSmith.

The Intel icc compiler with prototype STM support [12] was used for the STM transformation.

4.1 The Process

Conceptually the LockPick and LockSTM processes are quite simple, with four basic steps.

1. Identify and mark Atomic sections Atomic sections need to be identified manually. If the original program was written with locks and is assumed to be correctly synchronized, it may be a simple matter of looking for lock acquires and releases. Adding atomic sections in the design phase of an application is a good alternative to trying to retrofit them into an already written program. In addition, atomic sections should not be nested. Nesting is discussed in more depth in Section 4.2.

Instead of adding new syntactic constructs to the C language, atomic sections are annotated with a gcc attribute. To make a function atomic, the programmer should replace

```
void foo(...)
```

with

```
__attribute__ ((atomic)) void foo(...)
```

Currently, only functions can be marked as atomic. There is no technical reason why atomic sections cannot be within a function. However that choice was made for simplicity in implementation. In cases where an atomic section is not already a function, the developer must manually create one. In the experiments sections of this paper, atomic sections that have been created this way use the naming convention of foo_1 for the first atomic section created from function foo. An alternative would be to coarsen the atomic section to encompass the entire function. After the atomic annotations have been inserted, all the source and header files are merged into a single source file using the CIL merger.

2. Analyze shared memory locations The shared memory locations are identified and the LockPick algorithm described in Section 2.2 is run to generate a set of colors for each atomic section.

```
SharedLock A;

atomic void PromptForX(int i)
{
  exclusive_lock( A );
  printf( "%d", x[i] );
  scanf( "%d",  x+i );
  unlock( A );
}

atomic void SwapXY(int i, int j)
{
  shared_lock( A );
  __tm_atomic{
    int tmp = x[i];
    x[i] = y[j];
    y[j] = tmp;
  }
  unlock( A );
}

atomic void SetY(int i, int v)
{
  __tm_atomic{
    y[i] = v;
  }
}

atomic int GetY(int i)
{
  int tmp;
  __tm_atomic{
    tmp = y[i];
  }
  return tmp;
}

atomic void SetZ(int i, int v)
{
  __tm_atomic{
    z[i] = v;
  }
}

atomic int GetZ(int i)
{
  int tmp;
  __tm_atomic{
    tmp = z[i];
  }
  return tmp;
}
```

Figure 5. LockSTM generated code from Figure 2

The LockSTM version performs the additional algorithm described in Section 3.

3. Transform code based on the analysis Each atomic section is assigned a guard and a release action. The guard starts the atomic section either by acquiring a lock or by calling the STM. The release is the counterpart at the end of the section.

In LockPick, the guard is `pthread_mutex_lock(c)` and the release is `pthread_mutex_unlock(c)`, where c is a color.The LockSTM transformation is slightly different because there are now three types of atomic sections.

1. Non-revocable sections
   Guard: `pthread_rwlock_wrlock(c);`
   Release: `pthread_rwlock_unlock(c);`
2. Hybrid sections
   Guard: `pthread_rwlock_rdlock(c);` `__tm_atomic{`
   Release: `}` `pthread_rwlock_unlock(c);`
3. Pure STM sections
   Guard: `__tm_atomic{`
   Release: `}`

For each atomic function `foo()`, a wrapper function `foo_atomic()` is created. The wrapper is used to simplify cases where complex functions may have return statements at arbitrary points. This is necessary because each return statement requires a release before it. Because the Intel STM uses a lexically scoped block to indicate the start and end of a transaction, multiple releases do not work.

The wrapper acquires the guard, passes the arguments to the original function, releases the guard and returns a value if needed. All function calls to the old function are then replaced with calls to the newly created wrapper instead.

Both versions create a global lock for each color used. LockPick uses `pthread_mutex_t` and LockSTM uses `pthread_rwlock_t`. For each section that has multiple colors, multiple locks are acquired and released as needed. By having a total global ordering of the locks and no nesting of atomic sections, deadlock is impossible.

4. Compile and run the transformed code The resulting LockPick C code can be compiled with any standard C compiler. The resulting LockSTM code needs to be compiled with transaction support turned on with the Intel STM compiler.

4.2 Nested Atomic Sections

Nested atomic sections bring up a few interesting issues. The first is effect propagation. When one atomic section calls another, the callee's effects propagate upward to the caller during the analysis phase. This causes the two sections to share a color. The caller will acquire the locks for the shared colors in its guard acquire. Then the callee will try to reacquire those same locks in its guard acquire. If the atomic sections are being enforced by mutex locks, this would cause a deadlock. A fix is to use reentrant locks. Similarly, if an STM is being used to enforce atomicity, additional care has to be taken to not start or commit an additional transaction from within a transaction. A dynamic check could determine if nesting was occurring, but we opted to forbid it for simplicity.

In the experiments of this paper, nesting was not an issue. There were nested atomic sections, but a manual analysis showed that the nested sections were always called from within other atomic sections. These occurred when the original program used two locks, with one at a finer granularity. This meant that the atomic annotation on the nested section was no longer needed. Discovering nesting of this type would not normally occur when designing a program with atomic sections, but it can occur when retro-fitting atomic sections into an existing program.

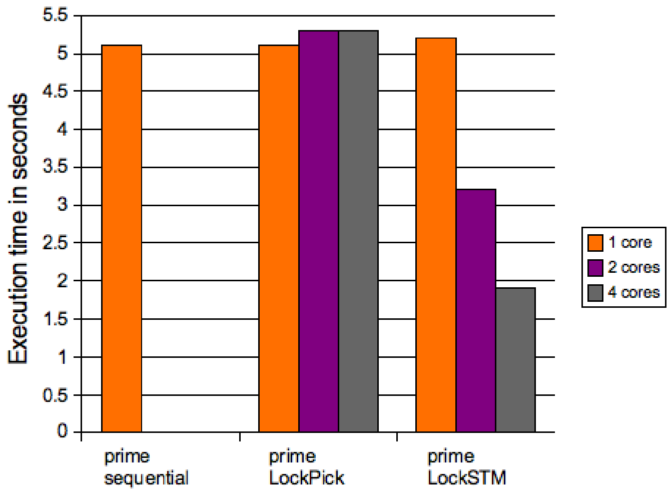


Figure 6. Prime execution time vs number of cores

5. Experiments

We evaluated the LockPick and LockSTM methods on a variety of applications. All of the timings were achieved by running each application eleven times and averaging the results.

The experiments were performed on a 2.66 GHz quad core Intel Xeon X5355 machine running Red Hat Enterprise Linux AS release 4 update 6 with 4GB of memory. All tests were compiled with the Intel 10.0.504 icc compiler with prototype STM support. Optimization level 3 (-O3) was used when compiling.

5.1 Prime

Prime is a simple example that shows the performance improvement possible for the LockSTM method given the right application. Prime takes a large list of randomly generated numbers and determines which ones are prime. In the multi-threaded version, the input list is partitioned equally between the threads. In addition, there is a status thread that periodically polls the current prime count and displays the result to the user.

Prime was written specifically to showcase the best possible performance of the LockSTM method. The source code to prime is in Appendix A.

LockPick ran in less than one second. Prime has two atomic sections that share a single color. The computation atomic section is revocable and the status atomic section is non-revocable. Figure 6 shows the performance of prime running in sequential mode and the LockPick version with one to four cores. Because the two sections share a single color, LockPick makes both sections mutually exclusive and sees no performance gain when increasing thread count. The application cannot be purely STMed because of the non-revocable section. As the figure shows, LockSTM is able to overcome this and provide a good speedup. The LockSTM performance gain is slightly below a perfect linear speedup.

5.2 STAMP

The Stanford Transactional Applications for Multi-Processing, STAMP [2] is a benchmark suite intended for use in transactional memory research. It consists of five applications with parallel and serial implementations along with reference data sets.

STAMP was developed to offer researchers a set of non-trivial applications that use coarse grained transactions to accomplish tasks in parallel. All of the transactions can be fully rolled back on a conflict detection. Therefore, it is not an ideal test case for

			LockPick			LockSTM			
	Lines of code	Analysis Time (s)	Atomic Sections	Shared Variables	Colors	Non-Revocable sections	Hybrid sections	Pure STM sections	Colors
bayes	9150	49.9	15	51	1	11	4	0	1
genome	5675	18.0	5	40	1	4	1	0	1
kmeans	2043	0.7	3	13	3	0	0	3	0
labyrinth	5224	7.2	3	42	2	2	0	1	1
vacation	6970	6.9	3	16	1	3	0	0	1

Figure 7. LockPick and LockSTM analysis on STAMP

combining locks with an STM for increased parallelism. However, it is possible to modify the constraints slightly so it is an appropriate test case. The STAMP applications will work with any STM that implements its interface. It supports function calls to locally defined functions and provides STM-safe versions of malloc/free. But by treating the memory allocation calls as external code, the STAMP applications are transformed into a mixture of revokable and non-revokable transactions.

The resulting mixture of revocable and non-revocable transactions did not seem unreasonable given the observed results of the memcached analysis in Section 5.3. Also the version of the Intel compiler used for these experiments does not support memory allocation in atomic sections.

Briefly, the applications are:

Bayes Bayesian networks are a way of representing probabilities in a compact graph form. This program uses a hill-climbing strategy to derive the network from a random data set.

Genome This program tries to reconstruct gene sequence given multiple small fragments. It performs a sliding algorithm to match gene segments using the Rabin-Karp string search algorithm.

Kmeans This program performs spatial clustering of random points in an iterative fashion until a fix point clustering solution is found.

Labyrinth This program is a simple maze solver that finds the shortest path between entry and exit points.

Vacation This program is a travel reservation simulation where multiple customers interact with multiple in-memory database tables.

5.2.1 STAMP analysis

Figure 7 shows the results of running the LockPick and LockSTM versions of all five programs. The programs ranged from two to nine thousand lines of code. Analysis times were fairly quick with all programs finishing in under a minute. Analysis speed correlated with the size and number of atomic sections. The STAMP tests are simple enough that LockPick is able to run in the more accurate context sensitive mode.

The number of atomic sections ranged from three to fifteen. LockPick's shared memory analysis discovered between 13-51 different shared memory locations in each program. When performing the coloring on these variables, LockPick reduced the amount of colors from the shared variable count to the count in the colors column.

LockSTM identified each atomic section as one of the three categories: Non-Revocable, Hybrid and Pure STM. The colors column in the LockSTM portion indicates how many colors were left after the reduction step.

In **bayes**, LockPick derived a single color solution and assigned that color to all atomic sections. The coloring graph would be a complete graph with fifteen vertices. The single color caused one global mutex lock to be used for all atomic sections. This eliminated all parallelism between atomic sections just like in the prime benchmark.

LockSTM was able to do more with bayes. Although it has eleven non-revocable sections, it also has the most STMable sections with four. The single color remains and must be enforced across all fifteen sections. As such, the STMable sections are locked with shared locking. Parallelism should increase slightly because of the locked STM sections.

Genome had the same LockPick result as bayes, with a single color and no available parallelism. Its LockSTM results were similar to bayes with a small increase in possible parallelism due to the single hybrid section.

Kmeans has more interesting lock inference results. Each of the three atomic sections has disjoint memory accesses from the other sections. LockPick assigned each section a unique color which gives the possibility of a 3× increase in speed if all sections run all the time.

Kmeans has only revocable sections, and therefore LockSTM was able to eliminate all the colors and create a pure STM solution. This is ideal for program performance, and shows the combination lock and STM method is not always needed.

Labyrinth had a similar LockPick result as kmeans but with two colors. One atomic section has its own color and the other two share a color. The atomic section with its own color was also the only revocable section. LockSTM eliminated the lock and converted it to a pure STM section.

Vacation is the least interesting because the LockPick results again show a single constraint across all atomic sections, but also all sections are non-revocable so LockSTM can do no better.

Overall the STAMP applications are fairly simple. The memory accesses were limited to the point where no atomic section had more than one color. This is likely because they were written for the express purpose of research. However, they do provide a broad view of the types of applications that can occur with the variation in LockSTM results.

5.2.2 STAMP performance

We benchmarked four version of each application. Figure 8 shows the results. The four versions are the original sequential version on one core, the LockPick and LockSTM versions on 1-4 cores and with the reference STM based on TL2 [4]. This reference version of the applications has all atomic sections STMed and should give a baseline for how much performance is lost by the non-revocable malloc assumption.

By looking down the single core column, the overhead of each transformation method can be observed. Because there is only one thread running, no locks will block and no transactions will be rolled back for a conflict. For the LockPick version, this is the cost of acquiring and releasing the locks without ever having to wait to get the lock. The LockSTM and TL2 rows show how the STM transformations can slow down code, even when there are no rollbacks.

Program	1 core	2 core	4 core
bayes	45.5		
bayes LockPick	44.7	55.5	51.7
bayes LockSTM	44.8	51.7	52.4
bayes TL2	42.9	52.8	43.2
genome	5.8		
genome LockPick	5.83	5.78	6.23
genome LockSTM	5.62	7.14	7.01
genome TL2	7.88	4.54	2.82
kmeans	16.3		
kmeans LockPick	18.5	11	7.7
kmeans LockSTM	34.6	27.1	16.2
kmeans TL2	40.1	22.5	14.3
labyrinth	3.58		
labyrinth LockPick	3.54	3.55	3.55
labyrinth LockSTM	3.58	3.6	3.63
labyrinth TL2	5.41	2.92	2.16
vacation	23.3		
vacation LockPick	23.6	42.9	45.5
vacation LockSTM	23.5	45.6	46.7
vacation TL2	98.4	69.4	56.2

Figure 8. STAMP runtimes in seconds

Looking across the table at the two and four core results shows whether each particular application increased or decreased in performance when running in parallel.

Bayes For some reason performance increased slightly for the non-sequential versions while running under a single core. This could be due to different optimizations being performed on the transformed code. Performance from increasing the number of threads was poor all around. Both LockPick and LockSTM slowed down due to lock contention when more then one core was used. The full STM implementation also does not perform well with run times not improving even with four cores. In conclusion, bayes does not seem to parallelize well.

Genome The LockSTM version showed a slight increase in speed over the sequential version. However the TL2 version had a considerable slowdown. Because of the single color solution LockPick performance was expected to be flat, perhaps slowing a bit with more cores as penalties for lock contention started to take effect. This behavior occurred as expected. LockSTM had the possibility of a slight improvement over LockPick, but instead performance dropped by about 20%. The full STM version scales nicely with a 2.8× increase in speed over its own single core version and a 2× increase over the sequential version. This shows the program itself scales well, but the inferred coloring is too restrictive to achieve good performance.

Kmeans The single core slowdown was large for this application. The STM versions slowed down by more than a factor of two. In contrast to the previous two applications, the LockPick results allowed a lot of additional parallelism. With three mutex locks, the best case speedup would be 3× if three sections were running the whole time. With four cores it achieved a 2.4× increase over its own single core performance.

Because LockSTM used no locks, kmeans gives a good view of a direct comparison of the two STM implementations. While both the Intel and the TL2 implementations show speed improvements, TL2 shows slightly better results. However, the LockPick solution outperformed both of them so this is not a compelling case for using transactions on this application.

Labyrinth Single core performance was again almost flat with TL2 showing a slowdown. The only version to show improvement with more cores was also the TL2 version.

The atomic section that had its own color and was converted to a pure STM function turned out to have almost no effect on performance. It is a 32 line piece of code that checks to see if a queue contains objects and removes an object if it does. The majority of the work is performed in the other two sections. The is the main reason that LockPick and LockSTM demonstrated no performance gain.

Vacation The single core slowdown for TL2 was dramatic with this version taking four times as long as the sequential version. The TL2 version does improve, but not very well. A single color and no STMable sections result in very poor performance with this benchmark for the LockPick and LockSTM versions.

In conclusion, the STAMP benchmarks did not prove a compelling case for mixing locks and STMs for increased performance. In addition, the large slowdowns caused by TL2 do not provide a compelling case for using an STM either.

5.3 Memcached

Memcached [5] is a high performance in memory object caching system intended for use in dynamic web based application. Its primary purpose is to hold commonly used objects in memory to alleviate load caused by repeated queries to a database.

Memcached was developed for LiveJournal to handle 20 million accesses per day and is currently used by many other large dynamic web sites such as Slashdot and Wikipedia. It achieves parallelism both through multiple running processes and multiple threads running in each process. The experiments here focus on a single process while varying the number of running threads. The source to version 1.2.5, released on March 4th, 2008, is approximately 16,700 lines of C code.

Data races are guarded against with five mutex locks, with an additional lock used to coordinate initialization. The code is not broken down into atomic sections, but rather acquires and releases the locks as needed. I manually located atomic sections by examining where locks were acquired. Typically these started at the original lock acquire and ended at the release. However, there were some cases where this caused atomic sections to be called from within other atomic sections. Which as discussed above is not allowed in LockPick or LockSTM. To fix the nesting issue, the granularity of some of the sections was coarsened. Or if the section was called only from within atomic sections, the atomic annotation was removed. The result was 39 atomic sections.

Because of the size of the application, LockPick ran out of memory when performing a context sensitive analysis. Thus we ran a context insensitive analysis instead, which took only eleven seconds. LockPick located 87 shared memory locations that were accessed from within atomic sections and derived an 18 lock solution. Figure 9 shows each atomic section along with the assigned colors. All of the atomic sections had from 1-5 colors, except for two; `stats_reset` and `process_stat_1`. These two functions cleared and displayed the statistics that other atomic sections gathered.

LockPick derived a solution with many more locks than the authors used because it was able to differentiate between each statistic gathered and derive a solution that allowed simultaneous execution when different stats were being updated. However it was able to reduce the total number of locks from the initial allocation of one for each of the 87 shared memory locations.

A majority of processing time is spent in the `mt_*` sections. Most of these share the colors `A` and `E`, while many of the other constraints, {`B`, `D`, `F`, `G`, `H`, `I`, `J`, `K`, `L`, `M`, `N`, `O`, `P`, `Q`}, are linked to statistics gathering. Color `C` guards access to a work queue.

The LockSTM transformation analysis was also performed on memcached. Of the 39 atomic sections, 23 made calls to non-revocable code. Of the remaining 16 STMable sections, 13 were hybrid sections and three were pure STM sections. The color `C` was only assigned to the pure STM sections and was therefore eliminated in the LockSTM version reducing the number of locks from 18 to 17.

The locked STM sections primarily had to do with statistics and, as such were relatively short. The non-revocable sections made calls to malloc/free, printf, strcpy, memset, perror and other variants. Some of these could likely be made revocable by using an STM that can allocate memory and by replacing some standard library calls with locally written equivalents. Further experimentation should be able to determine if a performance gain could be achieved this way.

We benchmarked memcached with the memslap [1] testing tool from Tangent Software. Figure 10 shows the results from the three versions of enforcing atomicity while varying from 1-4 threads of execution. The LockPick inferred locks were slightly slower when using a single core which was expected because of the additional locking. When running with two cores, the LockPick version gained about a two percent performance advantage. However since execution times varied by up to a half second, this small of a variance is not statistically significant. The LockSTM version had additional overhead on a single core and lagged behind the others in dual core test too.

In all versions of memcached, the performance flatlined after two threads. This primarily has to do with the benchmarking setup. Memcached is designed to be a very light-weight process, so much so that the client for the benchmark takes as much CPU time to execute as the server does. Because testing was done on a single machine, only two cores worth of CPU time were left for the server. Increasing the server to four threads caused the operating system to only run two threads at time while adding the overhead of context switching. Performance suffered slightly as expected.

Attempts to run the client software on another machine in the same network resulted in so much latency that execution times were increased by a factor of ten. With the increased latency it was not possible to load on the server to greater than one and varying the number of threads had no effect on performance. In a real memcached deployment, it is normal and suggested that the server and client run on the same machine. Therefore, we feel this is acceptable behavior.

Experimentation indicated that the slowdown with the modified versions may have been due to the additional locking caused by the statistics gathering. The code was refactored to remove the atomic sections that statistics gathering caused. Instead, the original calls to lock and unlock were left in even if those occurred from within another atomic section. The new version has 26 atomic sections for which LockPick derived a four lock solution. Five of these remaining atomic sections can be safely STMed. Figure 10 has the performance results of this method. With the statistics lock reinserted, single threaded performance for both LockPick and LockSTM is approximately equal to the original locking scheme. The two core LockSTM performance increased to within two percent of the original as well.

What this shows is although LockPick was able to derive a solution that allowed more parallelism, this did not translate into increased speed. The additional locks mainly increased overhead and offered no noticeable benefit. This is not an unexpected result because memcached is supposed to be a highly optimized program to start with.

A more interesting future experiment would be to use an STM that can treat more of the atomic sections as revocable.

Overall, this experiment showed that it is possible and reasonable to extract additional performance from real world applications. This needs to be done carefully so as to not generate excessive overhead like with the statistics. It also shows that in a complex application there are usually multiple atomic sections that can easily be STMed. The types of external calls in memcached were mostly memory related, so memcached could easily be combined with another STM to improve the portion of STMed sections.

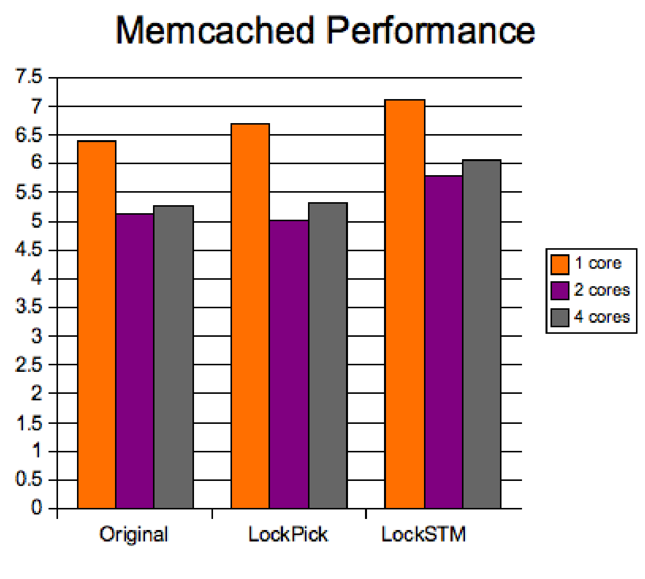
Atomic Section	Colors	Not Revoc	Hybrid	Pure STM
cqi_new_1	C			X
cqi_new_2	C			X
cqi_free	C			X
conn_new_1	B		X	
conn_new_2	K, Q		X	
conn_close_1	Q		X	
complete_nread_1	G		X	
process_get_cmd_1	L		X	
process_get_cmd_2	M		X	
process_get_cmd_3	N		X	
try_read_udp_1	O		X	
try_read_network_1	O		X	
transmit_1	P		X	
drive_machine_1	O		X	
mt_conn_from_fl	A		X	
mt_item_unlink	A, E, F		X	
mt_conn_add_to_fl	A	X		
mt_run_deferred_dels	A, E, F	X		
mt_item_alloc	A, E, F, H	X		
mt_item_get_notedel	A, E, F	X		
mt_item_link	A, E, J	X		
mt_item_remove	A, F	X		
mt_item_replace	A, E, F, J	X		
mt_item_update	A	X		
mt_defer_delete	A, F, I	X		
mt_add_delta	A, E, F, H, J	X		
mt_store_item	A, E, F, H, J	X		
mt_item_flush_expired	A, E, F	X		
mt_item_cachedump	A, I	X		
mt_item_stats	A	X		
mt_item_stats_sizes	A	X		
mt_assoc_mv_next_bkt	A, E	X		
mt_slabs_stats	F	X		
stats_prefix_rec_get	D, E	X		
stats_prefix_rec_delete	D, E	X		
stats_prefix_rec_set	D, E	X		
stats_prefix_dump	D	X		
stats_reset	D, G, H, J, K, L, M, N O, P	X		
process_stat_1	B, G, H, I, J, K, L, M, N, O, P, Q	X		

Figure 9. Memcached atomic sections and constraints

6. Related Work

There have been previous efforts to control access to shared memory via locking and atomic sections [16, 14, 11]. There have also been efforts to use software transactions to manage atomic sections [7, 8, 9] but there are relatively few that have attempted to use both locks and an STM concurrently[22, 19] and those use different techniques than presented here.

Transactions have been in use for quite some time [10, 21], and have been implemented in both hardware and software. STMs typically operate by logging each read and write to memory within an atomic section. Upon completion of the section, a commit is attempted. The log is scanned and if any conflicts are discovered,



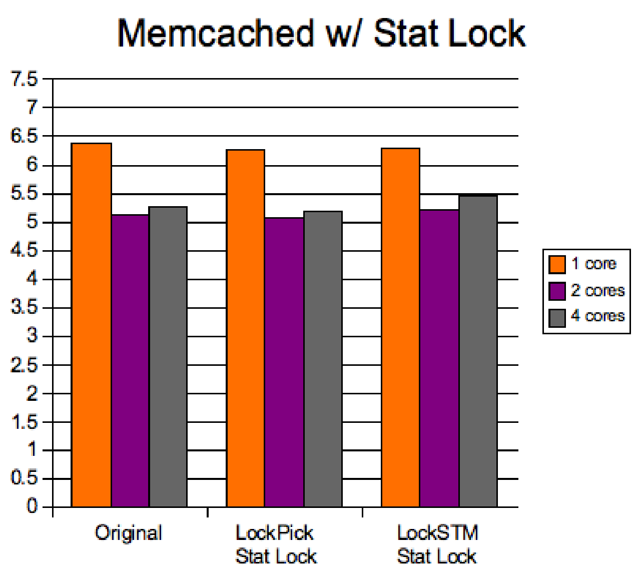


Figure 10. Memcached execution time vs number of cores

the writes of the transaction are rolled back and the transaction is re-run. A common extension to this is the conditional critical region (CCR) where the atomic section is not run until a precondition is met [7, 18]. CCRs can improve performance and can be useful to developers when structuring atomic sections.

STMs are usually implemented in a run-time environment [20] like the Java Virtual Machine or the OCaml run-time system. Operating at this level affords many opportunities for optimization [18]. In a lower level language like C, using a virtual machine is not optimal. To compensate for not having a run-time, calls to an external STM library are usually used. Developers must either make sure that they have correctly identified all the locations at which calls to the STM are needed, or they can use an automatic translation tool.

The most common solution to dealing with I/O or other code that results in external actions, is to forbid them from appearing within an atomic section. Harris suggests buffering I/O with a pre-registered operation that is executed outside of the atomic section [6]. Ringenburg and Grossman suggest that external calls which modify only local state can be safely executed. Additionally, external rollback and commit code should be written to compensate for other external calls [18]. However, this assumption relies on not interrupting external calls and the run-time environment being single threaded. Neither of these assumptions exist in a pure C environment.

Using locks to enforce the STAMP benchmarks has been examined as well [3]. While their approach focuses only on pessimistic locking, it uses read/write locks to enforce exclusion based on whether or not a particular atomic section reads or writes a shared location. In addition, they discovered fewer atomic sections in the bayes application. This difference can be attributed to a different set of #defines that enable and disable different portions of the program.

7. Conclusion

This paper presented a new way of combining pessimistic locks with optimistic transactions. This should allow increased parallelism in programs that would otherwise be restricted to using locking. The technique is practical and can be applied to C programs. However, its benefit is limited to a narrow class of applications and is hampered by current STM implementations. It does hold promise for the future as many atomic sections do perform actions that are not currently revocable. For significant progress to be made, either STMs will need to become more robust at revoking actions or the techniques in this paper will need to be refined. The optimal solution will likely be a combination of both.

References

[1] B. Aker. libmemcached 0.17. http://tangent.org/552/libmemcached.html.

[2] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture.* ", Jun 2007.

[3] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, 2008. To appear.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking 2. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC), Stockholm, Sweeden*, Sept. 2006.

[5] B. Fitzpatrick. memcached: a distributed memory object caching system. Available at http://www.danga.com/memcached/.

[6] T. Harris. Exceptions and side-effects in atomic blocks. In *Elsevier Science of Computer Programming*, Mar. 2005.

[7] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03*, pages 388–402, Oct. 2003.

[8] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05*, June 2005.

[9] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[11] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 82–91, New York, NY, USA, 2006. ACM Press.

[12] Intel. C++ stm compiler, prototype edition 1.0. Available at

http://softwarecommunity.intel.com/articles/eng/1460.htm.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[14] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL'06*, pages 346–358. ACM Press, 2006.

[15] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[16] P. Pratikakis, J. S. Foster, and M. Hicks. Lock Inference for Atomic Sections. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, Canada, June 2006.

[17] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, New York, NY, USA, 2006. ACM Press.

[18] M. F. Ringenburg and D. Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, New York, NY, USA, 2005. ACM Press.

[19] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[20] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[21] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[22] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *ECOOP 'O6*, pages 148–173, 2006.

A. Prime Source

```
int* data;
int primeCount;

// returns the smallest factor of p, or 0 is p is prime
// return values are in the range 0...sqrt(MAX_INT)
int isprime(int p)
{
  if(p <= 3)
    return 0;
  if(p % 2 == 0)
    return 2;

  int i;
  for (i = 3; (i*i) <= p; i += 2)
    if (p % i == 0)
      return i;
  return 0;
}

__attribute__((atomic)) void findFactor(int i)
{
  int f = isprime(i);
  if(f == 0){
    ++primeCount;
  }
}

void* primeThread(void* arg)
{
  int id = (int)arg;
  long numThread = global_params[PARAM_THREAD];
  long size = (1 << global_params[PARAM_SIZE]);

  int mySize = size/numThread;
  int myOffset = id * mySize;

  int i;
  for(i=0; i<mySize; ++i){
    findFactor(data[i+myOffset]);
  }
  return 0;
}


__attribute__((atomic)) void printPrimeCount()
{
    printf("PrimeCount: %7d\n",primeCount);
}

void* statusThread(void* arg)
{
  while(1){
    sleep(1);
    printPrimeCount();
  }
}

int main(int argc, char* const argv[])
{
 ...
  primeCount = 0;

  pthread_create(&status_thread,NULL,statusThread,NULL);
  for(i=0; i<numThread; ++i){
    pthread_create(threads+i,NULL,primeThread,(void*)i);
  }
  for(i=0; i<numThread; ++i){
    pthread_join(threads[i],NULL);
  }

  printf("Done  ");
  printPrimeCount();

  return 0;
}
```