Lock Free Data Structures using STM in Haskell

Anthony Discolo[1], Tim Harris[2], Simon Marlow[2], Simon Peyton Jones[2], Satnam Singh[1]

[1] Microsoft, One Microsoft Way, Redmond,
WA 98052, USA
{adiscolo, satnams}@microsoft.com
http://www.research.microsoft.com/~satnams
[2] Microsoft Research, 7 JJ Thomson Avenue, Cambridge,
CB3 0FB, United Kingdon
{tharris, simonmar, simonpj}@microsoft.com

Abstract. This paper explores the feasibility of re-expressing concurrent algorithms with explicit locks in terms of lock free code written using Haskell's implementation of software transactional memory. Experimental results are presented which show that for multi-processor systems the simpler lock free implementations offer superior performance when compared to their corresponding lock based implementations.

1 Introduction

This paper explores the feasibility of re-expressing lock based data structures and their associated operations in a functional language using a lock free methodology based on Haskell's implementation of composable *software transactional memory* (STM) [1]. Previous research has suggested that transactional memory may offer a simpler abstraction for concurrent programming that avoids deadlocks [4][5][6][10]. Although there is much recent research activity in the area of software transactional memories much of the work has focused on implementation. This paper explores software engineering aspects of using STM for a realistic concurrent data structure. Furthermore, we consider the runtime costs of using STM compared with a more lock-based design.

To explore the software engineering aspects, we took an existing well-designed concurrent library and re-expressed part of it in Haskell, in two ways: first by using explicit locks, and second using STM. The comparison between these two implementations is illuminating.

To explore performance, we instrumented both implementations. In particular, we instrument the implementations using a varying number of processors in order to discover how much parallelism can be exploited by each approach. Our results should be considered as highly preliminary, because our STM implementation is immature.

Finally, we draw a conclusion about the feasibility of lock free data structures in Haskell with STM from both a coding effort perspective and from a performance perspective. Although previous work has reported micro-benchmarks [3] and application level benchmarks [1] for various STM implementation schemes we focus here on benchmarks which compare explicitly locked and lock free implementations based on STM.

2 Background: STM in Concurrent Haskell

Software Transactional Memory (STM) is a mechanism for coordinating concurrent threads. We believe that STM offers a much higher level of abstraction than the traditional combination of locks and condition variables, a claim that this paper should substantiate. In this section we briefly review the STM idea, and especially its realization in concurrent Haskell; the interested reader should consult [2] for much more background and details.

Concurrent Haskell [8] is an extension to Haskell 98, a pure, lazy, functional programming language. It provides explicitly-forked threads, and abstractions for communicating between them. These constructs naturally involve side effects and so, given the lazy evaluation strategy, it is necessary to be able to control exactly when they occur. The big breakthrough came from using a mechanism called *monads* [9]. Here is the key idea: a value of type **`IO a`** is an "I/O action" that, when performed may do some input/output before yielding a value of type **`a`**. For example, the functions **`putChar`** and **`getChar`** have types:

```
putChar :: Char -> IO ()
getChar :: IO Char
```

That is, **`putChar`** takes a **`Char`** and delivers an I/O action that, when performed, prints the string on the standard output; while **`getChar`** is an action that, when performed, reads a character from the console and delivers it as the result of the action. A complete program must define an I/O action called **`main`**; executing the program means performing that action. For example:

```
main :: IO ()
main = putChar 'x'
```

I/O actions can be glued together by a *monadic bind* combinator. This is normally used through some syntactic sugar, allowing a C-like syntax. Here, for example, is a complete program that reads a character and then prints it twice:

```
main = do { c <- getChar; putChar c; putChar c }
```

Threads in Haskell communicate by reading and writing *transactional variables*, or **`TVars`**. The operations on **`TVars`** are as follows:

```
data TVar a
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

All these operations all make use of the *STM monad*, which supports a carefullydesigned set of transactional operations, including allocating, reading and writing transactional variables. The **readTVar** and **writeTVar** operations both return STM actions, but Haskell allows us to use the same **do {...}** syntax to compose STM actions as we did for I/O actions. These STM actions remain tentative during their execution: in order to expose an STM action to the rest of the system, it can be passed to a new function **atomically**, with type

```
atomically :: STM a -> IO a
```

It takes a memory transaction, of type **STM a**, and delivers an I/O action that, when performed, runs the transaction atomically with respect to all other memory transactions. For example, one might say:

```
main = do { ...; atomically (getR r 3); ... }
```

Operationally, **atomically** takes the tentative updates and actually applies them to the **TVar**s involved, thereby making these effects visible to other transactions. The **atomically** function and all of the **STM**-typed operations are
built over the software transactional memory. This deals with maintaining a per-thread transaction log that records the tentative accesses made to **TVar**s. When **atomically** is invoked the STM checks that the logged accesses are *valid* – i.e. no concurrent transaction has committed conflicting updates. If the log is valid then the STM *commits* it atomically to the heap. Otherwise the memory transaction is re-

executed with a fresh log.

Splitting the world into STM actions and I/O actions provides two valuable guarantees: (i) only STM actions and pure computation can be performed inside a memory transaction; in particular I/O actions cannot; (ii) no STM actions can be performed outside a transaction, so the programmer cannot accidentally read or write a **`TVar`** without the protection of **`atomically`**. Of course, one can always write **`atomically (readTVar v)`** to read a **`TVar`** in a trivial transaction, but the call to **`atomically`** cannot be omitted. As an example, here is a procedure that atomically increments a **`TVar`**:

```
incT :: TVar Int -> IO ()
incT v = atomically (do x <- readTVar v
                        writeTVar v (x+1))
```

The implementation guarantees that the body of a call to **`atomically`** runs atomically with respect to every other thread; for example, there is no possibility that another thread can read **`v`** between the **`readTVar`** and **`writeTVar`** of **`incT`**.

A transaction can block using **`retry`**:

```
retry :: STM a
```

The semantics of **`retry`** is to abort the current atomic transaction, and re-run it after one of the transactional variables has been updated. For example, here is a procedure that decrements a **`TVar`**, but blocks if the variable is already zero:

```
decT :: TVar Int -> IO ()
decT v = atomically (do x <- readTVar v
                        if x == 0
                           then retry
                           else return ()
                        writeTVar v (x-1))
```

Finally, the **`orElse`** function allows two transactions to be tried in sequence: (**``s1 `orElse` s2``**) is a transaction that first attempts **`s1`**; if it calls **`retry`**, then **`s2`** is tried instead; if that retries as well, then the entire call to **`orElse`** retries. For example, this procedure will decrement **`v1`** unless **`v1`** is already zero, in which case it will decrement **`v2`**. If both are zero, the thread will block:

```
decPair v1 v1 :: TVar Int -> TVar Int -> IO ()
decPair v1 v2 = atomically (decT v1 `orElse` decT v2)
```

In addition, the STM code needs no modifications at all to be robust to exceptions. The semantics of **`atomically`** is that if an exception is raised inside the transaction, then no globally visible state change whatsoever is made.

3 Programming ArrayBlockingQueue using STM

We selected the **`ArrayBlockingQueue`** class from JSR-166 [7] as the basis for our experiment. We use this class solely as an example from an existing library, rather than intending to make comparisons between the Haskell versions and those in Java. The name **`ArrayBlockingQueue`** is a bit of a misnomer, since this class represents a fixed length queue but contains blocking, non-blocking, and timeout interfaces to remove an element from the head of the queue and insert an element into the tail of the queue. The combination of these interfaces in one class complicates the implementation.

We built two implementations of (part of) the **`ArrayBlockingQueue`** data type in Haskell. The first, **`ArrayBlockingQueueIO`**, is described in Section 3.1, and uses a conventional lock-based approach. The second, **`ArrayBlockingQueueSTM`**, is described in Section 3.2, and uses transactional memory. Our goal is to contrast these two synchronization mechanisms, so we have tried to maintain as much shared code as possible, aside from synchronization.

We did not implement all interfaces of the Java ArrayBlockingQueue class. Instead, we selected representative methods from each of the three interfaces, as well as a few methods from other utility interfaces:

- **`take`**: Removes an element from the head of the queue, blocking if the queue is empty
- **`put`**: Inserts an element at the tail of the queue, blocking until space is

available if the queue is full

- **`peek`**: Removes an element from the head of the queue if one is immediately available, otherwise return Nothing
- **`offer`**: Inserts an element at the tail of the queue only if space is available
- **`poll`**: Retrieves and removes the head of this queue, or returns null if this queue is empty
- **`pollTimeout`**: Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available
- **`clear`**: Atomically removes all elements from the queue.
- **`contains`**: Returns true if this queue contains the specified element.
- **`remainingCapacity`**: Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking
- **`size`**: Returns the number of elements in this queue
- **`toArray`**: Returns an array containing all of the elements in this queue, in proper sequence

3.1 The conventional locking implementation

Here is the Haskell data structure definition for the locking implementation:

```
data ArrayBlockingQueueIO e = ArrayBlockingQueueIO {
```

```
iempty :: QSem,
ifull :: QSem,
ilock :: MVar (),
ihead :: IORef Int,
itail :: IORef Int,
iused :: IORef Int,
ilen :: Int,
ia :: IOArray Int e
}
```

A bit of explanation is necessary for readers not familiar with Haskell. The data block defines a data structure with named fields. The *e* in the definition is a *type variable* enabling an arbitrary type to be used over the fields. The format of each field is <field name> :: <type>. The following lists the types used in the structure:

- **`QSem`**: a traditional counting semaphore
- **`MVar ()`**: a mutex
- **`IORef Int`** : a pointer to an integer
- **`IOArray Int e`**: a pointer to an array of objects of type *e* indexed over integers

Let's now take a look at the implementation of some of the methods. Here is the top-level implementation of **`takeIO`**, which removes an element from the queue:

```
takeIO :: ArrayBlockingQueueIO e -> IO e
takeIO abq
  = do b <- waitQSem (iempty abq)
       e <- withMVar
              (ilock abq)
              (\dummy -> readHeadElementIO abq True)
       return e
```

The **takeIO** method must first wait for the **iempty** semaphore using **waitQSem** and then lock the queue mutex with **withMVar**. The mutex is necessary because the **iempty** and **ifull** semaphores simply signal the availability of a queue element or an empty slot in the queue, and they do not guarantee mutual exclusion over any of the fields of the **ArrayBlockingQueueIO** structure. Given this structure may be accessed concurrently by multiple threads, the mutex is necessary. Therefore, after acquiring the **iempty** semaphore, the queue lock must also be acquired before calling **readHeadElementIO** to read a queue element. The complexity of managing the semaphores and the lock over all the methods is considerable, as we will see in the remainder of this section.

Here is the top-level implementation of **peekIO**, which looks at the first element of the queue, without removing it:

```
peekIO :: ArrayBlockingQueueIO e -> IO (Maybe e)
peekIO abq
```

```
  = do b <- tryWaitQSem (iempty abq)
       if b
          then do
            me <- withMVar
                    (ilock abq)
                    (\dummy -> do
                       u <- readIORef (iused abq)
                       if u == 0
                          then return Nothing
                          else do
                            e <- readHeadElementIO
                                   abq
                                   False
                            return (Just e))
            signalQSem (iempty abq)
            return me
          else return Nothing
```

Because **peek** is a non-blocking method, the acquisition of the **iempty** semaphore is attempted with **tryWaitQSem**, which returns true if the semaphore was acquired. The remainder of the peek logic is executed only if the semaphore is acquired. In addition, the **iempty** semaphore must be signaled since the queue element value was copied and not actually removed from the queue. Care has to be taken to prevent bugs such as returning without releasing the mutex or acquiring multiple mutexes in the correct order, for example. This shows how fragile the synchronization code is in the locking version.

In order to get a complete picture of the take/peek code path, we must look at the implementation of **`readHeadElementIO`**:

```
readHeadElementIO :: ArrayBlockingQueueIO e -> Bool
                     -> IO e
readHeadElementIO abq remove
  = do h <- readIORef (ihead abq)
       e <- readArray (ia abq) h
       if remove
          then do let len = ilen abq
                      newh = h `mod` len
                  u <- readIORef (iused abq)
                  writeIORef (ihead abq) newh
                  writeIORef (iused abq) (u-1)
                  signalQSem (ifull abq)
          else return ()
       return e
```

Here, the different types of synchronization require different logic from the implementation. The locking version **`readHeadElementIO`** requires that the initial acquisition of the **`iempty`** semaphore and the queue mutex occur outside the method invocation. If **`readHeadElementIO`** were only used by **`takeIO`** and **`peekIO`**, then this would not be the case, but we invite the curious reader to look at the implementation of **pollTimeoutIO** and **pollReaderIO** below for yet another synchronization requirement imposed by that code path. The **readHeadElementIO** method takes a remove parameter that specifies whether the head element is copied or removed from the queue. If the element is removed, then the **ifull** semaphore must be signaled to signify that the queue has shrunk by one element.

Finally, let us look at the implementation of the most complex method in the **ArrayBlockingQueue** implementation: **pollTimeoutIO**. This method issues a blocking read from the head of the queue with a timeout.

```
data TimeoutContext e = TimeoutContext {
  done :: MVar Bool,
  val :: Chan (Maybe e)
  }

newTimeoutContextIO :: IO (TimeoutContext e)
newTimeoutContextIO
  = do d <- newMVar False
       c <- newChan
       return (TimeoutContext d c)

pollTimeoutIO :: ArrayBlockingQueueIO e
                 -> TimeDiff -> IO (Maybe e)
pollTimeoutIO abq timeout
  = do ctx <- newTimeoutContextIO
```

```
      forkIO (pollReaderIO abq ctx)
      forkIO (pollTimerIO timeout ctx)
      me <- readChan (val ctx)
      return me
```

In order to achieve a temporarily blocking read, the implementation of **pollTimeoutIO** forks two new threads, one responsible for the read (**pollReaderIO**) and one responsible for the timeout (**pollTimerIO**). A data structure (**TimeoutContext**) is shared between them to synchronize which thread finishes first and to hold the return value, if any.

```
pollReaderIO :: ArrayBlockingQueueIO e
             -> TimeoutContext e -> IO ()
pollReaderIO abq ctx
  = do waitQSem (iempty abq)
       modifyMVar
         (done ctx)
         (\d -> do
            if not d
               then do
                 e <- withMVar
                        (ilock abq)
                        (\dummy ->
                          readHeadElementIO abq True)
```

```
                  writeChan (val ctx) (Just e)
                else signalQSem (iempty abq)
             return True)
```

The **pollReaderIO** method requires a bit of explanation. It first must wait for the queue's **iempty** semaphore to become available, signifiying an element is able to be read. It then atomically reads the **TimeoutContext's** done flag to see if the timeout thread has already completed. If the timeout has not occurred, then it reads the element from the queue, placing it in the **TimeoutContext's** result channel. If the timeout has occurred, then the thread signals the **iempty** semaphore, effectively making the element readable by another thread. After the checks have been made, then the done flag is released.

```
startTimerIO :: TimeDiff -> IO (Chan ())
startTimerIO timeout
  = do c <- newChan
       forkIO (timerIO c timeout)
       return c

timerIO :: Chan () -> TimeDiff -> IO ()
timerIO c timeout
  = do let td = normalizeTimeDiff timeout
       let ps = (tdSec td) * 1000000
       threadDelay ps
```

```
        writeChan c ()
        return ()

pollTimerIO :: TimeDiff -> TimeoutContext e -> IO ()
pollTimerIO timeout ctx
  = do c <- startTimerIO timeout
       readChan c
       modifyMVar
         (done ctx)
         (\d -> do
            if not d
               then writeChan (val ctx) Nothing
               else return ()
            return True)
```

The **pollTimerIO** method implements a timer with respect to the read being performed by the **pollReaderIO** thread, It uses the **startTimerIO** method that simply writes **Nothing** into a channel after the timeout has occurred[1]. The **pollTimerIO** method simply issues a blocking read on the timer channel to wait for the timeout, and then writes **Nothing** into the **TimeoutContext**'s result channel signifying a timeout has occurred only if it is the first thread to access the **TimeoutContext** structure.

3.2 The STM implementation

Here is the Haskell data structure definition for the STM version:

```
data ArrayBlockingQueueSTM e = ArrayBlockingQueueSTM {
  shead :: TVar Int,
  stail :: TVar Int,
  sused :: TVar Int,
  slen :: Int,
  sa :: Array Int (TVar e)
  }
```

The following lists the types used in the structure above:

- **`TVar Int`**: a transacted integer
- **`Array Int (TVar e)`**: a array of transacted objects of type *e* indexed over integers

Note that the **`ArrayBlockingQueueSTM`** data structure definition is considera-

[1] While `threadDelay` could be used directly instead of calling `startTimerIO` in the locking version, the additional thread is required by the STM implementation. See the next section for more detail.

bly simpler because it lacks the two semaphores and one mutex that are present in the **ArrayBlockingQueueIO** implementation. As we will see, this simplicity translates in to simpler implementations for all methods as well. For example, here is **takeSTM**:

```
takeSTM :: ArrayBlockingQueueSTM e -> IO e
takeSTM abq
  = do me <- atomically
               (readHeadElementSTM abq True True)
       case me of
         Just e -> return e
```

The atomic block in **takeSTM** provides the only synchronization necessary in order to call **readHeadElementSTM** in the STM version. The implementation of peek is equally simple:

```
peekSTM :: ArrayBlockingQueueSTM e -> IO (Maybe e)
peekSTM abq
  = atomically (readHeadElementSTM abq False False)
```

Again, in comparison with the locking version, there is considerably less complexity in the STM version, because the **readHeadElementSTM** method is simply called within an atomic block. Here is the implementation of **readHeadElementSTM**:

```haskell
readHeadElementSTM :: ArrayBlockingQueueSTM e
                     -> Bool -> Bool -> STM (Maybe e)
readHeadElementSTM abq remove block
  = do u <- readTVar (sused abq)
       if u == 0
          then if block
                  then retry
                  else return Nothing
          else do h <- readTVar (ihead abq)
                  let tv = sa abq ! h
                  e <- readTVar tv
                  if remove
                     then do
                       let len = slen abq
                       let newh = h `mod` len
                       writeTVar (shead abq) $! newh
                       writeTVar (sused abq) $! (u-1)
                     else return ()
                  return (Just e)
```

The STM version **readHeadElementSTM** takes a **remove** parameter and a **block** parameter. In contrast to **readHeadElementIO**, the **readHeadElementSTM** method contains all the synchronization logic for the take/peek path. Note how the blocking read path is implemented with a **`retry`** statement. This effectively restarts the entire atomic block from the beginning and is much easier for the programmer to utilize correctly than the combination of semaphores and mutexes. The entire implementation of **`readHeadElementSTM`** is more concise and clear than the implementation of **`readHeadElementIO`**.

Finally, here is **`pollTimeoutSTM`**:

```
pollTimeoutSTM :: ArrayBlockingQueueSTM e
               -> TimeDiff -> IO (Maybe e)
pollTimeoutSTM abq timeout
  = do c <- startTimerIO timeout
       atomically ((do readTChan c
                       return Nothing)
                   `orElse`
                   (do me <- readHeadElementSTM
                               abq True True
                       return me))
```

Compared to **`pollTimeoutIO`**, notice how concise and natural the implementation of **`pollTimeoutSTM`** is with the use of the **`orElse`** statement within the atomic block. Fundamentally, there are three steps: (1) start the timer, (2) try to read from the timer channel signifying timeout period has elapsed, and if successful return **`Nothing`**, and (3) try to read an element from the head of the queue, and if successful return the element. If neither (2) nor (3) are satisfied, then the atomic block is restarted until one of these branches is successful.

The **`retry`** and **`orElse`** methods are very powerful features of the Haskell STM implementation and deserve more discussion. The **`retry`** method can be invoked anywhere inside the STM monad and restarts the atomic block from the beginning. The Haskell STM runtime manages transacted variables in an intelligent way and transparently blocks the transaction until one of the transacted variables has been modified. (Note there *cannot* be non-transacted variables within the STM monad.) In this way, the atomic block does not execute unless there is some chance that it can make progress.

Conditional atomic blocks or join patterns can be implemented with the **`orElse`** method. Note how the locking version forks two worker threads with a custom synchronization data structure, and how the custom synchronization logic between the two worker threads affects the synchronization logic throughout the rest of the program. In the STM version, the worker threads and custom synchronization logic are replaced by *one* **`orElse`** statement. This one statement more accurately reflects the programmer's intent in that allows the runtime to more efficiently and intelligently manage the execution of the atomic block. For example, if additional processors are available, each branch of the **`orElse`** statement may be executed on different processors and synchronized within the runtime, or each branch may be run sequentially.

3.3 Summary

It should have become clear by now that it is much easier to write thread-safe code using STM than using locks and condition variables. Not only that, but the STM code is far more robust to exceptions. Suppose that some exception happened in the middle of **`takeIO`**. For example, a null-pointer dereference or divide by zero. If such a thing could happen, extra exception handlers would be required to restore invariants and release locks, otherwise the data structure might be left in an inconsistent state. This error recovery code is very hard to write, even harder to test, and in some implementations may have a performance cost as well.

In contrast, the STM code needs no modification at all to be robust to exceptions, since **`atomically`** prevents any globally visible state changes from occurring if an exception is raised inside the atomic block.

4 Performance measurements

Once we completed the locking and lock-free implementations of ArrayBlockingQueue, we measured their performance under various test loads.

4.1 Test setup

The test harness includes the following command line parameters:

- test implementation (locking or STM)
- number of reader and writer threads
- number of iterations per thread

- length of the ArrayBlockingQueue

For this paper, we chose to investigate the performance of the blocking ArrayBlockingQueue methods. Specifically, we wanted to determine whether the respective implementations ran faster as additional threads are created and/or additional processors are added, keeping all other parameters the same.

The test creates an ArrayBlockingQueue of type integer is created, and an equal number of reader and writer threads are created that simply loop for the specified number of iterations performing take or put operations on the queue. The test completes when all threads have terminated.

For each processor configuration (1-8 processors), we varied only the number of threads in each test, so that the parameters of each test were {2, 4, 6, 8, 10, 12, 14} reader/writer threads, 100000 iterations, and queue length 100.

All of our measurements were made on our prototype implementation of STM in Haskell. This implementation is immature and has received little performance tuning. In particular, memory is reclaimed by a basic single-threaded stop-the-world generational collector. This degrades the performance of the STM results because the current STM implementation makes frequent memory allocations. In ongoing work we are developing a parallel collector and also removing the need for dynamic memory allocation during transactions. Nevertheless the measurements are useful to give a very preliminary idea of whether or not the two approaches have roughly comparable performance.

While the GHC runtime has a wide variety of debugging flags that can be used to monitor specific runtime events, this paper only focuses on the elapsed time of the tests. We ran each test on a Dell Optiplex 260 Pentium 4 3GHz CPU with 1GB RAM running Windows XP Professional SP2 and with successive processors enabled on a 4-way dual core Opteron HP DL585 multiprocessor with 1MB L2 cache per processor and 32GB RAM running Windows XP Server 2003 64-bit SP1 resulting in nine runs total per test. Our main interest was not the actual elapsed time values, but how the performance changed as additional processors were enabled.

4.2 Performance Results

The performance results are shown in the following figures.

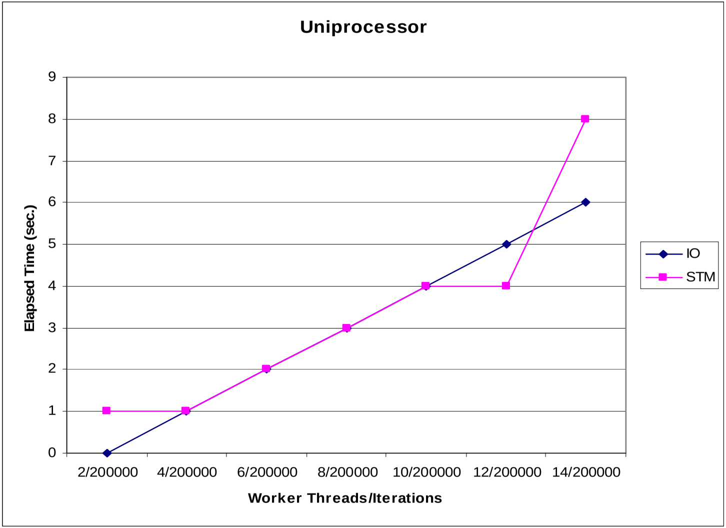

Figure 1: Uniprocessor Performance

Processors=2

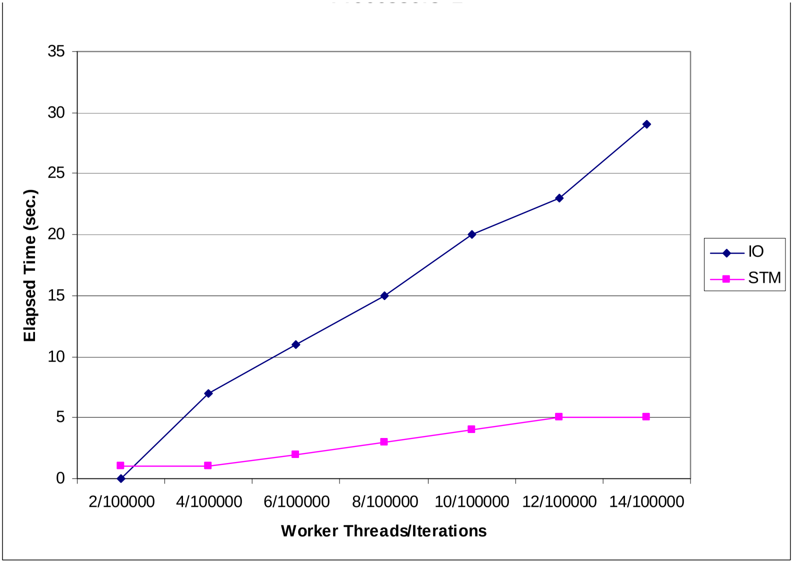

Figure 2: Two Processor Performance

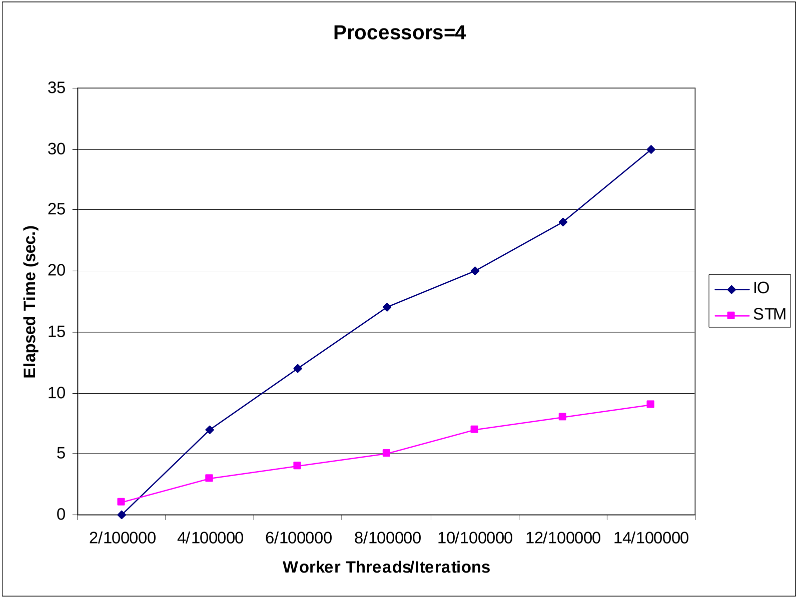

Figure 3: Four Processor Performance

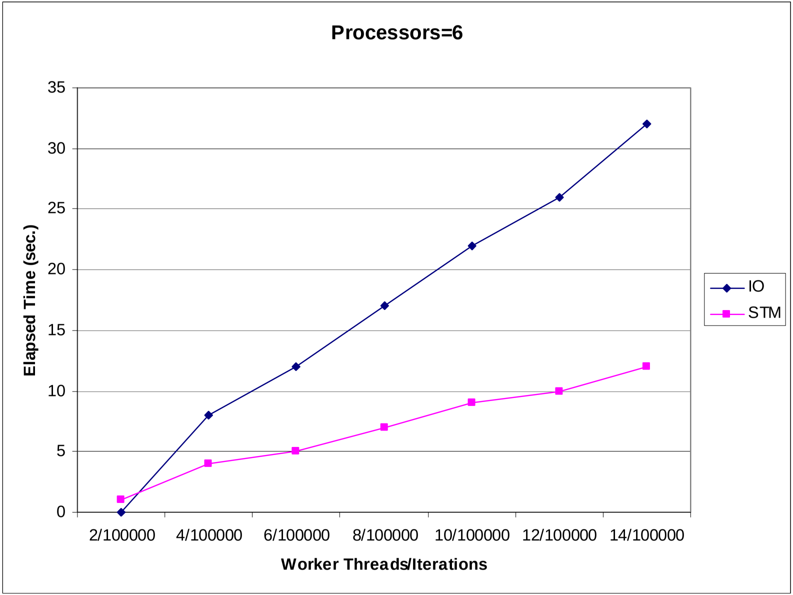

Figure 4: Six Processor Performance

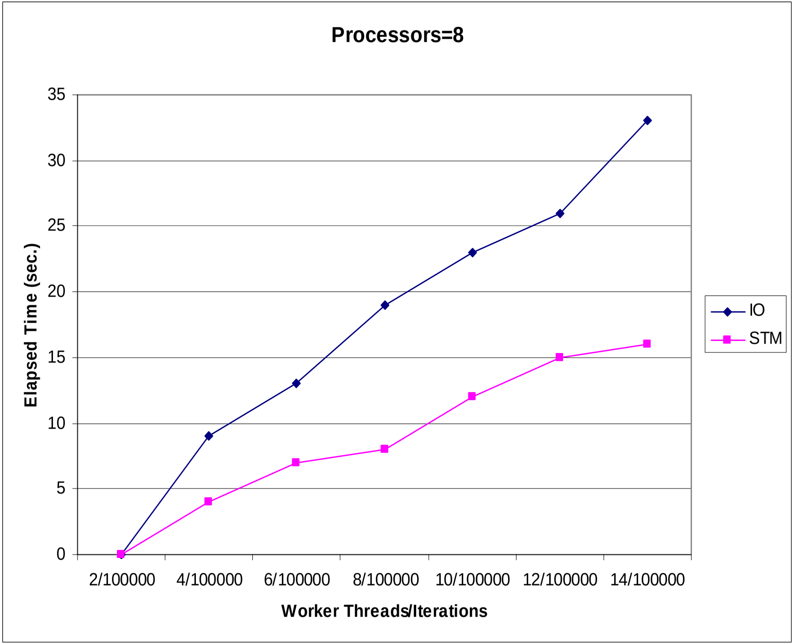

Figure 5: Eight Processor Performance

The results are very encouraging. On a uniprocessor, the performance of the locking version and the STM version were virtually identical. Once two or more processors were enabled, the STM version was consistently faster than the locking version running on the same number of processors. In fact, the fastest times on two or more processors were achieved by the STM version.

The STM and locking implementations are not exactly identical in their behavior. An important difference is the way in which each implementation deals with exceptions. The STM implementation has, in effect, a built in default exception handler which will cause the transaction to be rolled back. The locking version does not have this robust behavior, so that unexpected exceptions could leave the queue in an inconsistent state. It could be modified to handle exceptions and restore invariants, but that would make the code more complicates still, and would degrade performance. Consequently we should expect the performance of the STM implementation to improve even further over the locking version once error handling code has been added.

5 Related Work

Related work by Carlstrom et. al. [1] has shown that the conversion of lock based Java programs to versions that use transactions is often straightforward and sometimes leads to performance improvements. Hammond et. al. [3] used the TestHistrogram micro-benchmark which counts random numbers between 0 and 100 in bins. They also observed scaling for a transactional version up to 8 CPUs. Many benchmarks compare a single processor lock version of an algorithm against a transactional version that can scale up with the number of processors. In our experimental work we designed both the lock version and transactional versions to have the potential for exploiting extra processors.

6 Future Work

We are now investigating to what extent our observations apply to STM implementations in an imperative language. We are taking the same **`ArrayBlockingQueue`** example and coding it in C# with explicit locks and then again using an STM library called SXM [5]. The SXM library has user configurable conflict managers which will give us greater experimental control to help understand what kind of polices work best under different kinds of loads. We believe the best way to understand the characteristics of various STM implementation schemes is to build libraries and applications that place representative stresses on the underlying implementation.

7 Conclusions

It has been claimed that lock free concurrent programming with STM is easier than programming explicitly with locks. Our initial investigation into the re-implementation of a concurrent data structure and its operations from the JSR-166 suite in a functional language suggests that it is indeed the case that STM based code is easier to write and far less likely to be subject to deadlocks. Furthermore, the optimistic concurrency features of the STM implementation that we used offer considerable performance advantages on SMP multi-processor and multi-core systems compared to pessimistic lock based implementations.

The STM programming methodology is much easier to understand, concise, and less error-prone than traditional locking methodology using mutexes and semaphores. A key feature of the Haskell STM implementation is the **`orElse`** combinator which we used to *compose* small transactions into composite transactions. Haskell's type system also helped to statically constrain our programs to avoid stateful operations in transactions which can not be rolled back by prohibiting expressions in the IO monad in transactions.

Even with a very early implementation of the STM multiprocessor runtime, the STM implementation consistently outperformed the locking version when two or more processors were available. We expect even better STM performance as the language runtime implementation matures.

The encouraging initial results with library level units of concurrent data structures and operations paves the path for us to now build and instrument entire applications built out of lock free data structures designed using the functional language based methodology presented in this paper. Although locks still have a role in certain kinds of low level applications we believe that application level concurrency may often be tackled more effectively with STM as demonstrated in this paper.

References

1. B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Minh, L. Hammond, C. Kozyrakis, K. Olukotun. Transactional Execution of Java Programs. SCOOL 2005.
2. T. Harris, S. Marlow, S. Peyton Jones, M. Herlihy. Composable Memory Transactions. PPoPP 2005.
3. L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency. In Proceedings of the 11^{th} International Conference on Architecture Support for Programming Languages and Operating Systems, Oct. 2004.
4. T. Harris and K. Fraser. Language support for lightweight transactions. In OOPSLA ’03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 388–402. ACM Press, 2003.
5. M. P. Herlihy, V. Luchangco, M. Moir, and W. M. Scherer. Software transactional memory for dynamic-sized data structures. In Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing, July 2003.
6. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In Proceedings of the 20th International Symposium. on Computer Architecture 1993.
7. Itzstein, G. S, Kearney, D. Join Java: An alternative concurrency semantics for Java. Tech. Rep. ACRC-01-001, University of South Australia, 2001.
8. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In 23rd ACM Symposium on Principles of Programming Languages (POPL’96), pp. 295–308.
9. S. Peyton Jones and P. Wadler. Imperative functional programming. In 20th ACM Symposium on Principles of Programming Languages (POPL’93), pp. 71–84.
10. N. Shavit and S. Touitou. Software transactional memory. In Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Canada, August 1995.