From Locks to Transactional Memory: Lessons Learned from Porting a Real-world Application

Alexandre Skyrme
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
askyrme@inf.puc-rio.br

Noemi Rodriguez
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
noemi@inf.puc-rio.br

Abstract

Lock-based constructs such as mutual exclusion and condition variables are commonly employed for concurrency control, especially when preemptive multithreading with shared memory is used. However, locks have received a fair amount of criticism, in particular due to their complexity. Transactional memory has been proposed as a simpler alternative to lock-based synchronization, but it is still not clear what it takes to port existing real-world applications developed with locks to use transactions instead. In this paper we undertake the challenge of porting luaproc, an existing concurrent programming library for the Lua programming language, to use transactions instead of locks for its internal synchronization. We rely solely on the transactional memory support included in the latest stable version of the GNU C Compiler Collection (GCC) for our research. We present the lessons learned during the porting process, such as the need to use busy waits to replace condition variables, as well as the results for a comparative performance evaluation between the lock-based and the transactional memory versions of luaproc, which show that the busy waits can reduce performance in some test cases.

Categories and Subject Descriptors D [*I*]: 3

General Terms Transactional Memory, Mutual Exclusion, Condition Variables, GNU C Compiler Collection, Shared Memory

Keywords transactions, locks, multithreading, Lua, luaproc, mutex, condition, GCC

1. Introduction

A common concurrency model is to use preemptive multithreading with shared memory, for communication and synchronization, and locks, for concurrency control. Despite its popularity, this model has been recurrently criticized [1, 15], in particular due to the complexities associated with locks [19, 25, 26]. Among the cited problems with locks are the difficulty in reasoning about and debugging concurrent programs that rely on lock-based synchronization, the potential for deadlocks and race conditions that can lead to data corruption, as well as the the lack of composability.

TRANSACT '13 March 17, 2013, Houston, Texas, USA.

Transactional memory [11] has been proposed as an alternative to lock-based synchronization. Since support for Hardware Transactional Memory (HTM) is still limited [4, 20], Software Transactional Memory (STM) [23] is more readily available as it offers better portability. Transactional memory is supposed to be less error-prone [22] and to have simpler semantics than locks. The question, though, is: what does it take to use Software Transactional Memory, at its present state, as a replacement for lock-based synchronization in existing real-world concurrent programs? Related work, with few exceptions [17, 21, 29], is mostly focused on using standard benchmark suites to measure transactional memory performance alone [5], on implementing parallel versions of sequential programs using transactional memory [8] from the start, and on using transactional memory to fix existing concurrency bugs [28].

To answer that question, we take up the challenge of porting an existing real-world concurrent program that was originally developed with lock-based synchronization to a version using transactional memory instead. For that purpose, we chose the luaproc [24] concurrent programming library for the Lua programming language [13]. The luaproc library uses the POSIX Threads Library (pthreads) and thus relies on typical lock-based constructs, such as mutual exclusion and condition variables, to control concurrency. It includes typical concurrency patterns that can be found on concurrent programs that use lock-based constructs, therefore it is a sound choice for a case study. We take advantage of the Software Transactional Memory support included in the latest stable GNU Compiler Collection (GCC) version and use it as the foundation for our research.

This paper is organized as follows. In Section 2, we present an overview of related work. In Section 3 we present the target program used in our case study, the luaproc concurrent programming library, and explain how it controls concurrency. In Section 4 we provide a summary of the transactional memory constructs supported by the GCC. In Section refsec:fromlocks we present some remarks about our overall experience and lessons learned in porting luaproc to use transactional memory instead of locks. In Section 6 we show and discuss the results of a comparative performance evaluation between the lock-based implementation and the transactional memory-based implementation. Finally, in Section 7 we present some concluding remarks.

2. Related Work

Gajinov et al. [8] present a case study based on the implementation of a parallel version of the Quake multiplayer game server using transactional memory. Although they include a discussion about some of the challenges they encountered, their focus is not on what it takes to port a concurrent program that already uses lock-based synchronization to use transactional memory; instead, they take a

sequential version of the game server and implement a parallel version that is built to use transactional memory from the start. On the one hand, this approach, which differs from ours, allows for more flexibility to determine a parallelization strategy and consequently to model synchronization. Also, they present a performance evaluation focused on execution times, frame rates and transactional statistics. We, on the other hand, are less concerned with performance and more concerned with contention and overall processor use as compared with lock-based synchronization.

Zyulkyarov et al. [29] present a related case study that also uses the Quake multiplayer game server. They discuss the process of porting an existing parallel version of the game server that relies on lock-based synchronization to use transactional memory instead. Even though they take a similar approach to ours in investigating how transactional memory compares with lock-based synchronization, they use a different testbed to do so. The Quake multiplayer game server is an application built for the very specific purpose of serving game clients, its parallel version makes a distinctive use of concurrency according to the game's needs and tests must be carried out in accordance to the game's logic. Our testbed, luaproc, is a concurrent programming library which allows us to experiment with a broader range of concurrency usage patterns. Additionally, luaproc is small enough (1,100 lines of code in 4 files) when compared with Quake (27,400 lines of code in 56 files) so that it allows us to report our findings more thoroughly, yet complex enough (5 mutual exclusions, 3 condition variables) so that it lends itself to a case study. Furthermore, the growing importance of concurrency and interest in transactional memory, as well as the lack of exhaustive comparison studies with locks, demonstrate that there is still need for more such studies.

In yet another work based on the Quake multiplayer game server, Lupei et al. [17] present a case study to evaluate the parallelization of SynQuake, a 2D version of Quake 3 which they developed to use as a game benchmark. They compare a lock-based parallel version of Quake (ported to SynQuake) with a transactional memory parallel version of SynQuake, developed using a software transactional memory library (libTM). Besides using a different testbed from ours, other aspects in which their work differs from ours is that they do not explore conditional variables, while we do, and they use the libTM library, developed by one of the authors, while we use "off-the-shelf" GNU C Compiler Collection (GCC) and the GNU Transactional Memory Library (libitm). The libTM library is highly-customizable, it supports different conflict detection and conflict resolution methods, which are explored in the transactional memory version of SynQuake. The libitm library, on the other hand, does not allow programmers to use different conflict detection or resolution methods.

Finally, Volos et al. [28] discuss the usage of transactional memory to fix concurrency bugs in programs that use traditional lock-based synchronization. Their research is based on a study that picked some popular open-source software (the Mozilla web browser, the Apache HTTP server and the MySQL database) and tried to use transactional memory to fix existing concurrency bugs that had been reported and registered in public bug tracking systems. Despite the fact that real-world concurrent programs are used in the study, one of its premises is to use transactional memory to remedy chosen concurrency bugs that can benefit from transactional memory. Thus, different from our work, the focus is not to evaluate how transactional memory compares with lock-based synchronization when used as a replacement, but rather to carry out an empirical analysis on its use to fix chosen concurrency bugs on existing programs.

3. luaproc

The Lua programming language [13] supports coroutines, which enable collaborative multithreading. On the one hand, coroutines in Lua cannot natively exploit hardware parallelism, such as the one provided by multi-core processors, since they are confined to a single operating system thread. On the other hand, coroutines can be combined with system threads to create a combined concurrency model based on both user and system threads, like proposed in [12]. The luaproc programming library [24] uses the native concurrency support and the C API that Lua offers to implement that model.

User threads in luaproc are called *Lua processes*, which are execution flows of Lua code. Each Lua process uses a single Lua state, which defines the interpreter's state and keeps track of functions and global variables, among other interpreter related information. Thus, Lua processes have independent resources and do not share memory. When Lua code is loaded in a Lua state, its execution can be controlled just like a coroutine: it can be suspended, resumed and yielded. Also, it is possible to create multiple Lua states in C. Therefore, multiple Lua processes can be created and have their execution controlled from C. The library offers an optional feature that allows Lua states from finished Lua processes to be recycled, or in other words, to be used to host a new Lua process instead of being destroyed.

Communication between Lua processes is only possible by means of *message passing*. Each message can carry a tuple of Lua values of the following types: number, string, boolean or nil. Messages are addressed to *communication channels*, which are identified by (string) names and are completely decoupled from Lua processes. Sending a message in luaproc is a synchronous operation. A call to the send function only returns once the message has been received or the channel has been destroyed; if there is no Lua process waiting to receive a message on the same channel, the calling Lua process is blocked and its execution is suspended. Receiving a message in luaproc, however, can be a synchronous or asynchronous operation. A synchronous call to the receive operation works similarly to the send operation: if there is no Lua process waiting to send a message on the channel, the calling Lua process is blocked and its execution is suspended. An asynchronous call to the receive operation always returns immediately and indicates if it was able to receive a message or not.

System threads in luaproc are called *workers* and are implemented with the POSIX Threads Library (pthreads). Workers handle the execution of Lua processes and manage a single ready queue that holds Lua processes ready for execution. Workers have no direct relation with Lua processes and the same Lua process can be executed by multiple workers along its lifetime.

Although luaproc provides Lua programmers with a concurrency model in which there is no shared memory and both communication and synchronization rely on message passing, its implementation in C relies on shared data and on standard lock-based synchronization constructs that are widely used in programs to control concurrency. In particular, it uses mutual exclusion (`pthread_mutex_t`) and condition variables (`pthread_cond_t`) to control access to shared global variables and to coordinate workers and communication on channels, as follows:

ready queue mutex (mutex_sched) Controls access to the single (FIFO) queue that holds the Lua processes that are ready to be executed. Workers continuously remove Lua processes from the queue to execute them. They also insert Lua processes in the queue when processes are created or when they explicitly yield. Also used with the **awake worker condition variable**.

active Lua process count mutex (mutex_lp_count) Controls access to the active Lua process count, which is incremented when processes are created and decremented when they termi-

nate their execution. Also used with the **no active Lua process condition variable**.

channel list mutex (mutex_channel) Controls access to the list that holds channels used by Lua processes to communicate. The channel list is updated every time a communication channel is created or destroyed; it must be traversed every time a Lua process tries to send or receive a message. Also used with the **channel free condition variable**.

recycle mutex (mutex_recycle_list) Controls access to the list of Lua states that were used by finished Lua processes and that can be recycled, as well as to the recycle limit, which determines the maximum list size (or how many states can be recycled). The recycle limit is updated when a function for that purpose, included in the luaproc API, is called. The recycle list is accessed, and potentially updated, every time a Lua process is created or finishes executing.

channel mutex (channel→mutex) Controls access to individual channels, i.e., each channel has its own mutual exclusion variable. Channels hold two (FIFO) queues that cannot be non-empty at the same time: a queue of Lua processes blocked trying to send a message to the channel and a queue of Lua processes blocked trying to receive a message from the channel. The channel mutex, in fact, controls access to these queues.

awake worker condition (cond_wakeup_worker) Determines that a worker must be awaken. This condition is used when Lua processes are inserted in the ready queue and when workers are destroyed.

no active Lua process condition (cond_no_active_lp) Determines that there are no more active Lua processes, i.e., that all Lua processes have finished executing and the ready queue is empty. This condition is used for synchronization purposes, to prevent the main thread from exiting before all Lua processes have finished their execution.

channel free condition (channel→can_be_used) Determines that an individual channel may be used. Each channel has its own channel free condition variable. This condition is used to coordinate multiple operations, by different Lua processes, on the same channel. If two Lua processes try to send a message on a channel at the same time, one of them will be held back while trying to access the channel until the channel free condition is signaled.

4. Transactional Memory Support in the GNU Compiler Collection (GCC)

The GNU Compiler Collection (GCC) includes experimental transactional memory support since version 4.7.0. The support relies both on the compiler itself and on a runtime library, the GNU Transactional Memory Library (libitm); it can be enabled with the `-fgnu-tm` command line option. The code generated by the compiler when transactional memory support is enabled is compatible with the Linux variant of Intel's Transactional Memory ABI specification [14]. The language extensions implemented in GCC to support transactional memory are described in [27], which builds on the C++11 specification.

Transactions in C and C++ can be defined by *transaction statements*, *transaction expressions*, and *function transactions*. A *transaction statement*, or a statement that executes as a transaction, contains either the `__transaction_relaxed` keyword or the `__transaction_atomic` keyword, followed by a compound statement. The former keyword specifies a *relaxed transaction*, while the latter specifies an *atomic transaction*.

A compound statement executed as a relaxed transaction does not observe changes made by other transactions, nor do other transactions observe its partial results before it completes; its execution does not interleave with other transactions (atomic or relaxed) executing concurrently. Relaxed transactions do not impose restrictions on the code they execute, thus even code that might seem unsuitable for transactions can be wrapped in a relaxed transaction. The lack of restrictions allows relaxed transactions to support communication and synchronization with other threads by using constructs such as locks, C++11 atomic variables, volatile variables and I/O operations, i.e., it allows for interoperability with existing synchronization and communication constructs while providing isolation among transactions. This flexibility comes at a price, though: isolation cannot be guaranteed for relaxed transactions that include such communication and synchronization constructs, therefore their execution can appear interleaved with non-transactional code executed concurrently by other threads. The granularity of the potential interleaving, however, is not specified. Additionally, relaxed transactions may execute statements with side effects that cannot be reverted, such as I/O operations. These statements, referred to as *irrevocable actions*, prevent relaxed transactions from being canceled (or rolled back).

A compound statement executed as an atomic transaction, conversely, has stronger isolation guarantees. It does not observe changes made by other threads, nor do other threads observe its partial results before it completes; its execution does not interleave with other threads executing concurrently. So, while relaxed transactions guarantee isolation from concurrent (atomic or relaxed) transactional code, atomic transactions guarantee isolation from concurrent transactional and non-transactional code. Moreover, atomic transactions appear to execute atomically, in an all-or-nothing fashion, and can be explicitly canceled (with the `__transaction_cancel` statement) to void the effects of their statements. However, just as the flexibility of relaxed transactions comes at a price, so does the stronger isolation guarantees of atomic transactions: only safe statements, or statements that have no side effects that become visible before the transaction completes, are allowed in atomic transactions. This restriction ensures that atomic transactions can be canceled. Since atomic transactions can include function calls, function declarations may use the keywords `transaction_safe` and `transaction_unsafe` to specify whether they are considered safe or not — it is up to programmers to use these keywords to appropriately classify their functions. Functions declared as safe cannot contain statements that are not safe (such as I/O or lock-based synchronization operations).

Transaction expressions and *function transactions* are simply syntactic features. Transaction expressions use the same keywords that define transaction statements (`__transaction_relaxed` and `__transaction_atomic`); however, instead of being followed by a compound statement, they are followed by a parenthesized expression. Function transactions also use the same keywords that define transaction statements; still, they are included in function definitions to indicate that a function's body is executed inside a transaction.

The semantics of transactional memory in GCC is based on the C++11 standard memory model [3], which defines some ordering constraints for multithreaded programs. The transactional memory constructs implemented in GCC are only guaranteed to behave properly, and thus to provide transaction isolation, in programs that do not contain data race conditions.

5. From Locks to Transactional Memory

We took up the challenge of porting the luaproc concurrent programming library for the Lua programming language, a real-world application that relies on lock-based constructs provided by the

POSIX Threads Library (pthreads), to use transactional memory instead. Our goal was to replace all pthreads lock-based constructs in luaproc, such as mutual exclusion (`pthread_mutex_t`) and condition variables (`pthread_cond_t`), with transactional memory constructs supported by the GNU Compiler Collection (GCC). We intentionally avoid mixing locks and transactions, as we believe this complicates reasoning about synchronization and our main interest lies in evaluating transactional memory as a replacement for locks.

The first decision we had to make when porting luaproc was whether we would use atomic transactions, relaxed transactions or both atomic and relaxed transactions on a case-by-case basis. Choosing atomic transactions would provide us with stronger isolation and the ability to explicitly cancel transactions, but it would require us to specify which of the functions implemented in luaproc are transaction safe and would restrict statements allowed in transactions, as well as in functions called within transactions. Choosing relaxed transactions, on the other hand, would provide us with greater flexibility to place statements and function calls in transactions, but would not allow us to explicitly cancel transactions and would provide us with weaker isolation.

Since luaproc already used lock-based constructs, i.e., its critical regions were already defined and we assumed it did not have race conditions, our isolation requirement was just that statements that were already within critical regions would not have their execution interleaved. Also, we did not anticipate the need to explicitly cancel transactions. Therefore, we chose to use relaxed transactions, with the added benefit of having no restrictions to place statements and function calls within transactions — this was especially useful for critical sections that include functions with side effects that cannot be rolled-back (such as `pthread_create`, for example). Just like we intentionally avoided mixing locks with transactions, we avoid mixing atomic and relaxed transactions, since we believe it complicates reasoning about synchronization.

Having decided to use relaxed transactions, we started the actual porting process. We worked our way through the critical regions in each of the luaproc two modules: the scheduler (lpsched.c) and the API implementation (luaproc.c). In each module we started with the easier mutual exclusion variables, notably those that were not used with condition variables, and then moved on to the more elaborate synchronization statements, notably those that were used with condition variables or that had a coarser granularity.

Replacing lock-based constructs with transactions was sometimes just a matter of exchanging a `pthread_mutex_lock` call for a `__transaction_relaxed` statement and the corresponding `pthread_mutex_unlock` call for a closing brace (to mark the end of the transaction statement). This was the case for critical regions where there were no calls to operations on condition variables, like the example presented in figures 1 and 2, which shows the code to increase the count of active Lua processes in a shared variable (`lpcount`). This simple pattern closely relates to the notion that the semantics of transactions can be understood as a *single global lock* [10]. We estimate we were able to port roughly half of the critical regions in luaproc by using this pattern, i.e., by replacing lock and unlock function calls with transaction statements and without any additional changes. Nevertheless, the effort and time consumed to port such critical regions represent only a small part of the overall effort and time required to port luaproc as a whole.

Porting critical regions that included calls to operations on condition variables, however, was not as straightforward as replacing lock and unlock function calls. We already expected this hurdle, since the transactional memory support in GCC does not include an alternative or equivalent construct to condition variables. As a matter of fact, most transactional memory systems do not include such functionality, and although condition variable designs for transactional memory have been proposed [6, 16], no standards have been set.

```
void sched_inc_lpcount( void ) {
  pthread_mutex_lock( &mutex_lp_count );
  lpcount++;
  pthread_mutex_unlock( &mutex_lp_count );
}
```

Figure 1. Increasing the active Lua process count with locks.

```
void sched_inc_lpcount( void ) {
  __transaction_relaxed {
    lpcount++;
  }
}
```

Figure 2. Increasing the active Lua process count with a transaction.

Our approach for dealing with condition variables was twofold, according to what they were used to express. On the first case, when dealing with condition variables used to express the state of other existing variables, we started by removing the signal function calls (`pthread_cond_signal` and `pthread_cond_broadcast`). Then, we replaced the wait function calls (`pthread_cond_wait`) with a busy wait on the value of the existing variable that had its state represented by the condition variable. Using busy waits is an expected pattern [2, 11] in obstruction-free concurrency and is commonly observed when explicit `retry` [9] (or similar) operations are not supported. We used this approach, for instance, with the **no active Lua process** condition variable. This variable is used to allow a thread to wait until there are no more active Lua processes, an event which is signaled by other threads when the value of the `lpcount` shared variable used to keep track of active Lua processes reaches zero. In this case, we had to replace the wait function call with a busy wait on the value of the shared variable (`lpcount`). However, to avoid a livelock due to conflicting transactions, we placed the busy wait outside the transaction, which we used just to read the value of the shared variable. This particular example is presented in figures 3 and 4, which show the function used to wait until there are no more active Lua processes.

```
void sched_wait( void ) {
  pthread_mutex_lock( &mutex_lp_count );
  if( lpcount != 0 ) {
    pthread_cond_wait( &cond_no_active_lp,
                       &mutex_lp_count );
  }
  pthread_mutex_unlock( &mutex_lp_count );
}
```

Figure 3. Waiting until there are no more active Lua processes with locks.

```
void sched_wait( void ) {
  while ( __transaction_relaxed( lpcount )
          != 0 );
}
```

Figure 4. Waiting until there are no more active Lua processes with a transaction.

A `sched_yield` function call could be used in the code presented in figure 4, in the while body, to have the current thread relinquish the processor and allow another thread to run, in case the `no_active_lp` flag remained false. We explored that possibility by running the performance tests described in Section 6 with a modified version of luaproc that included a `sched_yield` function call as described; however, we did not observe significant differences in execution times and processor use, i.e., we did not observe a performance improvement. It is our understanding that including the function call does not necessarily increase performance, as yielding is likely to be advantageous when there are more workers than processor cores and plenty of Lua processes being actively executed to keep workers busy, a scenario not explored by the performance tests.

We applied the same pattern when porting the **awake worker condition variable**. This variable is used to allow workers to wait until there are Lua processes to be executed in the ready process queue, a condition signaled by functions that place processes in the queue. When using transactions, workers enter a busy wait to check the size of the ready Lua process queue, which is read from within a transaction and copied to a local variable, just like we did previously to check the active Lua process count. Apart from the condition variable replacement, it is also worth noting that by using transactions we were sometimes able to simply remove unlock function calls in critical regions, for instance when handling errors or particular conditions that would result in exiting a loop or returning from a function.

In the second case, when dealing with condition variables that, by themselves, were used to express a state, we had no choice but to replace them with regular (integer) variables, which we then used to implement busy waits similarly to the previously described porting pattern. This was the case with the **channel free condition variable** included in the structure that defines communication channels and is used to indicate whether a channel is currently being used or whether it is free to send/receive a message, i.e., a boolean state. Thus, we replaced it with an integer (flag) variable with the same name, as shown in figures 5 and 6.

The **channel free condition variable** was perhaps the condition variable involved in the most elaborate synchronization statements in luaproc. On the one hand, it provided us with an emblematic example of how transactional memory can make the semantics of synchronization easier to understand when the conditions are right. It allowed us to change the implementation of a function to unlock access to a channel which originally operated on two mutual exclusion variables and signaled a condition variable to simply execute a single statement within a transaction, as shown in figures 7 and 8. The easier semantics, though, come at the cost of using busy waits to check for conditions.

On the other hand, the **channel free condition variable** also provided us with the most intricate synchronization statement to port. At first it might seem as if this condition variable should be a mutual exclusion instead. However, the fact that channels can be destroyed while there are blocked threads waiting to use them makes this unfeasible, as we must be able to resume all blocked threads once the channel is destroyed. The statement shown in figure 9 is used in functions to access communication channels (called every time a Lua process sends or receives a message) and to destroy them. First, it tries to find a channel by using its name (`channel_unlocked_get`); whether the channel exists, its reference is returned, otherwise NULL is returned. If a channel is found, it tries to secure exclusive access to it by trying to lock the channel's mutual exclusion variable (`pthread_mutex_trylock`). In case it finds a channel but it isn't able to secure exclusive access, it waits until the corresponding **channel free condition variable** is signaled.

```
struct stchannel {
  list send;
  list recv;
  pthread_mutex_t mutex;
  pthread_cond_t can_be_used;
};

/* create a new channel and insert it into
   channels table */
static channel *chan_create( const char
                             *cname )
{

  channel *chan;

  /* get exclusive access to operate
     channels */
  pthread_mutex_lock( &mutex_channel );

  /* create new channel and register */
  lua_getglobal( chanls, LUAPROC_CHAN_TABLE );
  chan = (channel *)lua_newuserdata( chanls,
         sizeof( channel ));
  lua_setfield( chanls, -2, cname );
  /* remove channel table from stack */
  lua_pop( chanls, 1 );

  /* initialize channel struct */
  list_init( &chan->send );
  list_init( &chan->recv );

  pthread_mutex_init( &chan->mutex, NULL );
  pthread_cond_init( &chan->can_be_used,
                     NULL );

  /* release exclusive access to operate
     channels */
  pthread_mutex_unlock( &mutex_channel );

  return chan;
}
```

Figure 5. The communication channel's structure and creation function with locks.

Once again, we recurred to a busy wait to replace a wait function call on a condition variable. In this case, however, we had to take a more resourceful approach, as shown in figure 10. We begin by trying to find a channel by its name, similarly to the original lock-based synchronization statement. If a channel is not found, the transaction exits. If a channel is found, it copies the value of the integer flag that indicates if the channel is free to use to a local variable. Then, if it finds that the channel is free to use, it changes the flag to indicate the channel is not free anymore. The busy wait that wraps the transaction ensures that it will be executed over again until it either determines that a channel does not exist or finds a channel, determines the channel is free to use and secures exclusive access to the channel by altering its free-to-use flag. As long as it finds a channel but the channel is not free to use, it will re-execute the transaction, in a behavior that mimics the behavior of the wait function call on a condition variable, but using a busy wait. In fact, all signal operations to the **channel free condition variable** were replaced by statements to set the free to use flag to true.

Overall, porting luaproc to use transactional memory instead of lock-based constructs was a fairly straightforward process, with two notable exceptions, which we believe highlight some of the present challenges of using transactional memory: choosing which transaction type(s) to use and dealing with condition variables. The

```
struct stchannel {
  list send;
  list recv;
  int can_be_used;
};

/* create a new channel and insert it into
   channels table */
static channel *chan_create( const char
                             *cname )
{

  channel *chan;

  __transaction_relaxed {

    /* create new channel and register */
    lua_getglobal( chanls,
                   LUAPROC_CHAN_TABLE );
    chan = (channel *)lua_newuserdata( chanls,
           sizeof( channel ));
    lua_setfield( chanls, -2, cname );
    /* remove channel table from stack */
    lua_pop( chanls, 1 );

    /* initialize channel struct */
    list_init( &chan->send );
    list_init( &chan->recv );
    chan->can_be_used = TRUE;

  }

  return chan;
}
```

Figure 6. The communication channel's structure and creation function with a transaction.

```
void luaproc_unlock_channel( channel *chan ) {
  /* get exclusive access to op. on channels */
  pthread_mutex_lock( &mutex_channel );
  /* release channel exclusive access */
  pthread_mutex_unlock( &chan->mutex );
  /* signal channel can be used */
  pthread_cond_signal( &chan->can_be_used );
  /* release exclusive access to op. on
     channels */
  pthread_mutex_unlock( &mutex_channel );
}
```

Figure 7. A simple function to unlock access to a channel with locks.

former required understanding how both atomic and relaxed transactions work, as well as reasoning about how each of them would blend with luaproc. The choice between transaction types is particularly important, as each type provides different isolation guarantees and the same program could behave differently depending on the atomicity semantics that are employed [18]. The latter required coming up with a pattern to replace condition variables (and their corresponding wait and signal operations). In our case this pattern turned out to be a busy wait, which is an extremely simple synchronization mechanism, yet causes a potentially wasteful increase in processor use.

```
void luaproc_unlock_channel( channel *chan ) {
  __transaction_relaxed {
    chan->can_be_used = TRUE;
  }
}
```

Figure 8. A simple function to unlock access to a channel with a transaction.

```
/*
   try to get channel and lock it; if lock
   fails, release external lock (mutex_channel)
   to try again when signaled -- this avoids
   keeping the external lock busy for too long.
   during the release, the channel may be
   destroyed, so it must try to get it again.
*/
while ((( chan = channel_unlocked_get(chname))
          != NULL ) &&
        ( pthread_mutex_trylock( &chan->mutex )
          != 0 )) {
  pthread_cond_wait( &chan->can_be_used,
                     &mutex_channel );
}
```

Figure 9. An elaborate synchronization statement used in functions to access and to destroy communication channels.

```
do {
  __transaction_relaxed {
    chan = channel_unlocked_get( chname );
    if ( chan != NULL ) {
      chan_free = chan->can_be_used;
      if ( chan->can_be_used == TRUE ) {
        chan->can_be_used = FALSE;
      }
    }
  }
} while (( chan != NULL ) &&
         ( chan_free == FALSE ));
```

Figure 10. A resourceful use of transactions to replace waiting on a condition variable in functions to access and destroy communication channels.

6. Performance

While porting luaproc to use transactional memory we replaced all lock-based constructs with transactions, sometimes wrapping them with busy waits. These changes, especially the introduction of busy waits, prompted us to assess how the performance of the lock-based version of luaproc would compare to the transactional memory version. We were especially interested in two metrics: total execution time and processor use.

We based this comparative performance evaluation on three tests. We executed the tests on a machine with an Intel Core i7-870 processor with four cores of 2.93GHz each, capable of running up to eight threads, with 4GB of DDR3 1333MHz RAM and running Ubuntu 12.04 LTS with kernel 3.2.0-38-generic #61-Ubuntu SMP. Each individual test execution was repeated ten times and the results we present in this section are the means of observed values. We measured both execution times and processor use with the GNU `time` utility.

On the first test, we evaluated the performance impact of the busy waits we introduced as a replacement for the **awake worker condition** used by workers and the **no active Lua process condition**. The busy wait that replaced the **awake worker condition** causes workers to continuously check whether there are Lua processes in the ready process queue. The busy wait that replaced the **no active Lua process condition** causes the main luaproc thread to continuously check whether there are active Lua processes. Therefore, in this test we create a number of workers and then create a single Lua process that simply executes a long `for` loop, i.e., we intentionally create a situation where there are more workers than actual Lua processes to be executed.

We experimented with 1, 2, 4 and 8 workers in the first test and started by measuring execution times. The lock-based version of luaproc had almost constant execution times of 5.35s, with a 0.01s standard deviation, independently of the number of workers. The transactional memory version had slightly increasing execution times, as we increased the number of workers, from 5.61s to 6.47s, from 4% to 20% longer than the lock-based version execution time for the same number of workers; standard deviations ranged from 0.01s to 0.17s.

Processor use in the first test, however, as shown in figure 11, increases significantly when more workers are used in the transactional memory version, while it stays constant in the lock-based version of luaproc. Variance was insignificant for the processor use of the lock-based version (0.00%) but the transactional memory version exhibited very small standard deviations (from 0.70% to 0.97%) for 1 and 2 workers, and more significant standard deviations for 4 workers (4.29%) and 8 workers (25.88%). The constant processor use in the lock-based version can be explained by the fact that only one worker remains active during the test's execution, namely the one that is executing the only active Lua process. Throughout execution, the main thread simply waits (on a condition variable) for the active Lua process to finish and any remaining workers simply wait (also on a condition variable) for other Lua processes to be inserted in the ready process queue; thus, they do not increase processor use.

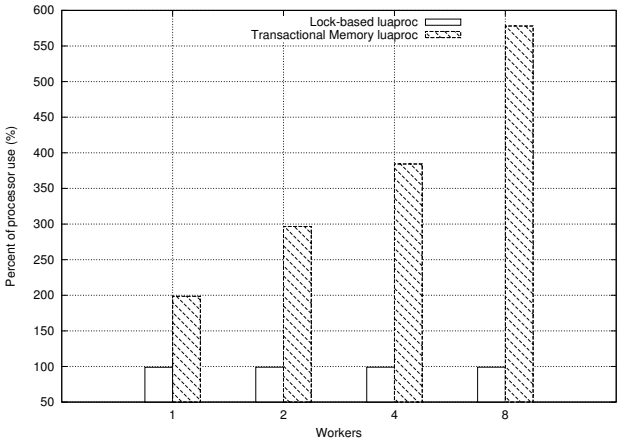

Figure 11. Processor use for the first performance test: multiple workers, a single Lua process.

In the transactional memory version of luaproc, the increased processor use can be explained by two reasons. The first reason is that luaproc uses its main thread just to wait until all Lua processes have finished executing, i.e., when a single worker is used there are actually two threads, the main one which simply waits and another one which actually executes Lua processes. On the one hand, in the lock-based version of luaproc, waiting for all Lua processes to finish executing was implemented with a condition variable and thus did not increase processor use. In the transactional memory version of luaproc, on the other hand, this waiting was implemented with a busy wait. This means that the main thread is constantly polling to check whether all Lua processes have finished executing and therefore it consumes a processor. That is why the results show near 200% processor use when a single worker is used. The second reason is that when porting luaproc, we replaced the main worker loop, which originally waited on a condition variable, to continuously poll the ready process queue. This means that, even when the ready process queue is empty, workers will poll it, consuming a processor. That is why, despite not increasing actual work to be done (for instance by creating more Lua processes), the processor use increases as we increase the number of workers. This suggests that in the transactional memory version of luaproc, the relation between workers and Lua processes is more important than in the lock-based version and perhaps this version should allow for dynamic adaptation during execution.

We also used the first test to further investigate where time was being spent when using each of luaproc's versions. For that matter, we employed the `strace` utility to produce a summary of system calls and relative time spent on each system call during the test's execution. Perhaps surprisingly, we found that both versions reportedly spent close to 100% of their time making calls to the `futex` [7] system call, a construct which can be used to implement basic locking or higher-level locking abstractions, such as the POSIX Threads Library (pthreads) mutual exclusion and condition variables. This means that, despite providing a transactional memory abstraction, the GNU Transactional Memory Library (libitm) relies on the same system call used by pthreads for its underlying synchronization. Most likely, libitm will switch to specific hardware supported system calls for transactions if (or once) they are available. Moreover, we noticed that the total number of calls to the `futex` system call was significantly higher in the transactional memory version of luaproc (around 200,000 versus around 8 in the lock-based version). We assume that difference is due to the busy waits we employed in the transactional memory version.

On the second test, we evaluated the performance impact of the busy waits we introduced as a replacement for the **channel free condition variable** used in communication channels. These busy waits were implemented in functions to destroy channels and to obtain references to channels (called every time a message is sent or received, as well as when a channel is created). Thus, in this test we first create a number of communication channels, then we create 100 Lua processes; half of the Lua processes continuously send a message and the other half continuously receive a message. Each sender/receiver repeats the send/receive cycle for 10,000 times. We ran this test with different numbers of workers and communication channels. Its results are presented in table 1.

As the results for the second test show, execution times for both versions increased as we increased the contention on communication channels. Still, the transactional memory version had considerably longer execution times than the lock-based version. In fact, it had execution times that ranged from 37% to 244% longer, or 125% longer on average, than the lock-based version. Standard deviation ranged from 0.01s to 0.89s for the lock-based version and from 0.13s to 2.75s for the transactional memory version. The longer execution times for the transactional memory version can be explained by the increased contention on communication channels, which in turn evidences the cost of the busy waits used to access them. As we increased the number of workers, the number of simultaneously active Lua processes also increased and so did the demand to access communication channels. Therefore, since channels cannot be accessed concurrently, more workers got stuck waiting for their turn to execute send or receive operations. This is a

Workers	Comm. Channels	Execution time (s)	
		Locks	Transactional Memory
1	1	1.00	2.10
	2	1.02	2.09
	4	1.02	2.13
2	1	7.08	9.72
	2	6.92	12.61
	4	6.50	12.14
4	1	15.41	27.36
	2	14.71	26.89
	4	14.43	26.35
8	1	22.26	76.61
	2	21.62	73.78
	4	20.42	69.49

Table 1. Execution times for the second performance test: multiple Lua processes sending and receiving messages, different numbers of workers and communication channels.

worst-case scenario for busy waits, as performance was clearly impacted. The overall processor use in this test followed the pattern observed in the first test, i.e., the transactional memory version of luaproc consumed more processors during the test's execution.

On the third test, we used a parallel string search application to evaluate how an application would perform with each version of luaproc. In this test we were interested in assessing the overall performance of an application and not in stressing specific aspects of luaproc. The string search application relies on three modules. The first (master) module creates workers and communication channels, as well as reads the name of the file with the strings (or patterns) that must be searched for and the names of the target files to search in. The second (coordinator) module handles work distribution and consolidates results. The third (searcher) module does the actual searching, i.e., it reads target files looking for the specified strings. Each searcher receives a single file name at a time; it searches the file for the specified strings and then it sends the matching lines back to the coordinator. If there are files left to search, the coordinator then sends a new file name to the searcher after it reports its results.

For this test, we used a sample strings file with 5 strings, one per line, and multiple copies of the same target file (an ASCII log file, with one entry per line). The target file had 8,599,067 lines and 2,172,914,027 bytes (around 2GB). We ensured that the there were always enough workers and target file copies to run searchers in parallel. Therefore, when working with 1 searcher we used a single target file copy and 2 workers (1 coordinator + 1 searcher), when working with 2 searchers we used two target file copies and 3 workers (1 coordinator + 2 searchers) and so forth. The results of the third test are presented in table 2.

As the results of the third test show, when we used a single worker, execution times increased almost linearly, in accordance with the number of target files; when we increased the number of workers, execution times only increased significantly when there were more target files than workers, which confirms we were able to exploit hardware parallelism to search target files in parallel. The only exception occurred when we ran the test with 5 workers and 4 target files, which resulted in longer execution times than when we ran it with 5 workers and 2 target files, although still shorter than with 3 workers and 4 target files. We suspect that was due to an I/O bottleneck when accessing the single hard disk drive where target files were stored. It is worth noting that in this test the execution times for the lock-based and the transactional memory luaproc versions do not differ significantly. This is a best-case scenario for busy waits, where we adapted the number of workers to the actual work to be done and there is no contention on communication channels. The overall processor use in this test followed the pattern observed in the first test, i.e., the transactional memory version of luaproc consumed more processors during the test's execution.

Workers	Target files	Execution time (s)	
		Locks	Transactional Memory
1	1	133.72	134.80
	2	269.47	271.68
	4	541.27	544.17
3	1	130.99	146.05
	2	141.53	148.81
	4	315.62	323.92
5	1	129.47	155.94
	2	139.64	170.60
	4	237.99	255.71

Table 2. Execution times of the third performance test: parallel string search.

7. Conclusion

In this paper we presented our experience, or lessons learned, from porting the luaproc concurrent programming library to use transactional memory instead of the lock-based constructs. In our case study, we relied solely on the transactional memory support included in the latest stable GNU Compiler Collection (GCC) version and on commonly available hardware without specific transactional memory support.

Porting luaproc to use transactions instead of locks was a mixed experience. On the one hand, replacing mutual exclusion variables (that were not used with condition variables) with transactions was very straightforward and sometimes even resulted in simpler code, i.e., less lines with better readability. These mutual exclusion locks that were easy to replace accounted for roughly half of the critical sections in luaproc.

On the other hand, replacing condition variables with transactions required more careful reasoning. Since the transactional memory support in GCC, like most transactional memory systems, does not include an alternative or equivalent construct to condition variables, we were forced to wrap transactions with busy waits and to adapt code accordingly. Sometimes this resulted in more lines of code and, in our perception, harder to understand synchronization statements. Replacing condition variables with busy waits and adapting the code accordingly accounted for most of the time we spent porting luaproc to use transactional memory. This suggests that the lack of a native, equivalent, functionality is still an open issue for transactional memory. Moreover, using busy waits has a performance impact on processor use and potentially on execution times. Even in the era of multi-core processors, it is not clear that spending processor cycles to continuously poll variables is a sound choice, especially on machines that host multiple servers or run multiple programs for different users.

Finally, many of the problems commonly associated with locks come down to relying on the programmers' discipline to ensure that: critical regions are protected with locks, locks are properly released, and deadlocks are avoided. However, transactional memory also relies, to a certain extent, on the programmers' discipline. It is up to the programmer to define which transaction type(s) will be employed (atomic and/or relaxed) and to reason about the consequences of that choice, to determine which code blocks must be

executed atomically, to ensure transactional data is not accessed outside transaction statements and, when using atomic transactions, to properly annotate function declarations to indicate whether they are transaction safe.

Acknowledgments

The authors would like to thank Simone Barbosa, from PUC-Rio, for her careful reading of the first draft and valuable suggestions, as well as the anonymous reviewers for their insightful comments. The authors would also like to thank CNPq for partially supporting this work (grant 305874/2010-1).

References

[1] J. Armstrong. Why I don't like shared memory, September 2006. URL `http://armstrongonsoftware.blogspot.com/2006/09/-why-i-dont-like-shared-memory.html`. Personal Blog – Armstrong on Software.

[2] H.-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, Berkeley, CA, USA, 2009. USENIX Association.

[3] H.-J. Boehm and S. V. Adve. You don't know jack about shared variables or memory models. *Communications of the ACM*, 55(2): 48–54, Feb. 2012. ISSN 0001-0782. doi: 10.1145/2076450.2076465.

[4] P. Bright. IBM's new transactional memory: make-or-break time for multithreaded revolution, August 2011. URL `http://arstechnica.com/hardware/news/2011/08/-ibms-new-transactional-memory-make-or-break-time--for-multithreaded-revolution.ars`.

[5] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4): 70–77, Apr. 2011. ISSN 0001-0782.

[6] P. Dudnik and M. Swift. Condition variables and transactional memory: Problem or opportunity? In *TRANSACT '09: 4th Workshop on Transactional Computing*, February 2009.

[7] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, June 2002.

[8] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 126–135, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-498-0. doi: 10.1145/1542275.1542298.

[9] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9.

[10] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*, chapter 2, pages 36–37. Morgan and Claypool Publishers, 2nd edition, 2010. ISBN 1608452352, 9781608452354.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9.

[12] R. Ierusalimschy. *Programming in Lua*, chapter 30. Lua.org, second edition, 2006. ISBN 8590379825.

[13] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua – an Extensible Extension Language. *Software – Practice and Experience*, 26(6):635–652, 1996.

[14] Intel. *Intel Transactional Memory Compiler and Runtime Application Binary Interface*. Intel Corporation, May 2009. Revision 1.1 (Draft).

[15] E. A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/-EECS-2006-1.html`. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

[16] V. Luchangco and V. J. Marathe. Revisiting condition variables and transactions. In *TRANSACT '11: 6th Workshop on Transactional Computing*, June 2011.

[17] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the 5th European Conference on Computer systems*, EuroSys '10, pages 41–54, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. doi: 10.1145/1755913.1755919.

[18] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), July 2006. ISSN 1556-6056.

[19] J. Ousterhout. Why threads are a bad idea (for most purposes). *Presentation given at the 1996 USENIX Annual Technical Conference*, January 1996.

[20] J. Reinders. Transactional synchronization in Haswell, February 2012. URL `http://software.intel.com/en-us/blogs/2012/02/07/-transactional-synchronization-in-haswell/`.

[21] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. Txlinux: using and managing hardware transactional memory in an operating system. *SIGOPS Operating Systems Review*, 41(6):87–102, October 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294271.

[22] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. doi: 10.1145/1693453.1693462.

[23] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3.

[24] A. Skyrme, N. Rodriguez, and R. Ierusalimschy. Exploring Lua for concurrent programming. *Journal of Universal Computer Science*, 14 (21):3556–3572, dec 2008.

[25] H. Sutter. The trouble with locks. *Dr. Dobb's Journal*, March 2005. URL `http://www.drdobbs.com/cpp/the-trouble-with-locks/-184401930`.

[26] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3:54–62, September 2005. ISSN 1542-7730.

[27] TM Specification Drafting Group. *Draft Specification of Transactional Language Constructs for C++*. Transactional Memory Specification Drafting Group, February 2012. Version 1.1.

[28] H. Volos, A. J. Tack, M. M. Swift, and S. Lu. Applying transactional memory to concurrency bugs. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 211–222, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150999.

[29] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic Quake: Using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 25–34, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: 10.1145/1504176.1504183.