Document number: N4438
Date: 2015-04-09
Project: Programming Language C++, Transactional Memory Working Group
Reply to: Brett Hall <bretthall@fastmail.com>

Industrial Experience with Transactional Memory at Wyatt Technology

With work progressing on the ISO C++ Transactional Memory TS, we offer a study on our use of a library-based transactional memory system in C++ at Wyatt Technology. Our results have been extremely positive and can perhaps offer some insight into what can be done to improve the transactional memory support in C++ as it evolves.

Introduction

While there has been much research into transactional memory (TM) there have been few examples of its use in industry that we know of. There are some examples from the high-performance computing and super computing sectors. There are also examples in the Haskell programming community. But all of those examples are from rather specialized niches. For the past three years at Wyatt Technology we've been using TM in a somewhat different context: a data collection and analysis application for use with our light scattering instruments called Dynamics.

In this paper we detail our transactional memory system and how we've applied it. We'll talk about what went well, what problems we've run into, and what features of the system turned out to be useful. Hopefully this can give some insight to the committee about how to standardize TM and what features to include in the system that is standardized.

Rationale

Why did we use transactional memory? Answering that question requires looking at the history of Dynamics. Over the years that Dynamics has been in use the amount of data that it has been tasked with handling has increased by orders of magnitude (due to improvements in both the instruments that it interfaces with and its own automation factilities). Unfortunately, thread-safety of the data had been originally handled by only allowing the data to be used in the main GUI thread of the program. Other threads were used to communicate with instruments, but once data was received from an instrument it was sent to the GUI thread where it was put into the data store and then only accessible from that thread. This precluded the parallelization of calculations, a big problem since the multi-core era was dawning. Even worse, since the data could only be processed on the GUI thread, the GUI would suffer from freezes and stuttering while analyses were being carried out. For common file sizes this could go on for minutes.

There were many attempts to solve the thread-safety issues but they all lead to a cascading series of changes that were going to require much of the program to be rewritten. Given competitive pressures for new features this issue was shelved for a few years as other functionality was worked on.

During this interim period, I had begun experimenting with the Haskell programming language and its implementation of transactional memory. As an exercise I implemented a system similar to theirs in C++. Also during this time period lots of planning was being done for the day when we would have time to attack the Dynamics thread-safety issue again. As this planning was going on we kept finding cases where transactional memory seemed to be a good, if not perfect, fit. Doing some proof of concept work and performance testing of the transactional memory system that I had built showed that it could fit our purposes. When the time came to attack the thread-safety issue, which was going to require rewriting much of the core of the program, the decision was made to use transactional memory in the data layer. It quickly expanded to fill many other

niches in the program as well. About the only place we don't use it, due to performance concerns, is in the low-level instrument communication code.

Dynamics

Dynamics is a data collection and analysis application for Wyatt Technology's dynamic and phase analysis light scattering instruments. Since much of our success with transactional memory can probably be attributed to the nature of this application it would be good to have some details on the structure of our data and the nature of the calculations being done. We'll limit our discussion to the dynamic light scattering data, the phase analysis data is similar.

The main piece of data used in dynamic light scattering is the *auto-correlation function* (ACF) which measures how well a signal correlates with itself over time (for our instruments the signal is the intensity of the light scattered by the sample). In Dynamics the data is structured as a series of *measurements* where each measurement is further broken down into a series of *acquisitions*. Each acquisition captures an ACF. The number of acquisitions in a measurement and how long the signal is correlated for in each acquisition are controlled by the user.

Once we have a set of acquisitions we perform a variety of calculations on the ACFs to tease out the properties of the samples being analyzed. Once the results for the acquisitions are calculated we apply a set of data filters and the ACFs of the acquisitions whose results pass the filters are averaged to create an ACF for the measurement (thus reducing the amount of noise in the measurement's data). The analyses are then repeated on this average ACF to determine the results for the measurement. The thing to keep in mind about all this is that the calculations for the acquisitions are independent and can thus be carried out in parallel. The data filters create a dependency between the acquisition results and the measurement's calculations, but the calculations of each measurement in a data file are independent. Thus the system has a lot of easily exploitable parallelism.

Our TM System

The transactional memory system that we have been using is entirely library based — no compiler modifications were possible as we use Microsoft Visual Studio as our primary development platform. As such it is an explicit system where any value that is to be transacted needs to be declared as a `stm::var<value_type>` where `value_type` is the type of the value that is being transacted. The main interface of `stm::var` is:

```
namespace stm
{
   template<typename value_type>
   class var
   {
   public:
      value_type get () const;
      const value_type& get (atomic& at) const;

      void set (const value_type& value);
      void set (const value_type& value, atomic& at);
   };
}
```

Objects of type `atomic` represent transactions. Functions that take one of these objects as an argument can only be called from within a transaction. The versions of `get` and `set` above that do not take `atomic` arguments are convenience functions that start a transaction and then call the versions that do take an `atomic` argument. From the interface it should be obvious that `value_type` must be copyable — we currently

do not support the storage of *move only* types in `stm::var`[1]. It is possible for the transacted version of `get` to return a reference since `get` will copy the value into the transaction and thus we can guarantee that the reference will be good for the length of the transaction (one just has to be sure to copy the object if it is returned out of the transaction).

Transactions are started using the `atomically` function that is effectively[2] defined as

```
namespace stm
{
   template <typename func>
   auto atomically (const func& f,
                    const atomic_params& args = atomic_params ())
      -> decltype (f (std::declval<atomic>()));
}
```

After starting the transaction and creating an `atomic` object, `atomically` calls `f` (the function passed in) passing it the `atomic` object that has been created. When `f` returns, `atomically` handles validating any reads and then commits any writes if the validation succeeds. Here *validation* means checking the read variables for modifications by other threads. If any other thread has committed a transaction that writes to a variable that has been read in our transaction, and we read that variable before the transaction in the other thread committed (thus getting an older value), then our transaction has a conflict and must restart. If there are no conflicts then our writes are made visible to other threads in an atomic fashion (i.e. if one write is seen then all writes from the transaction must be visible).

It is possible to manually validate during a transaction by calling the `validate` method on the `atomic` object – failed validation causes the transaction to be restarted immediately. If a long running calculation is done inside of a transaction then manual validation is used to avoid doing extra work if the transaction becomes invalid due to changes made in another thread. It is also possible to validate individual `stm::var` objects which proves useful when one knows that there is a high chance for some subset of the read variables to have conflicts and one wants to avoid having to pay for a validation of all the reads. There are also cases where manual validation is needed to ensure consistency of reads of variables that are used together in a transaction (this will be discussed in more detail below).

Throwing an exception from `f` will cause the transaction to be rolled back with the exception being allowed to propagate out of the call to `atomically`. It is the responsibility of code outside of the call to `atomically` to catch any exceptions thrown by functions that are run inside of `atomically`.

The first call of `get` on a `stm::var` within a given transaction yields the most recently committed value, even if that commit happened after the start of the transaction. Every call to `get` for a given `stm::var` in a given transaction will yield that same value, unless `set` is called. After a call to `set` the `stm::var` will have the set value for the remainder of the transaction, unless `set` is called again.

Starvation, i.e. short transactions repeatedly causing conflicts in another long running transaction preventing the long running transaction from ever committing, is a possibility in any transactional system. In our system we address this by allowing the caller of `atomically` to specify a limit on how many conflicts are acceptable and what to do if this limit is reached. The two options are to throw an exception out of `atomically` or to *run locked*. The latter option prevents any other threads from committing transactions until the transaction that is *running locked* has committed, thus preventing it from being starved any further.

[1] Transaction rollback precludes the transparent use of move semantics in our TM system. In order to support move semantics we will probably need to resort to some sort of wrapper objects around the values to be moved into or out of `Vars`. These objects would be responsible for restoring the *unmoved* state in the event of transaction rollback as different action needs to be taken depending on whether the moved object originated from inside or outside the transaction.

[2] There are some details being glossed over here. We need to apply some meta-programming in order to transparently handle functions that return `void` being one example.

Transactions can be nested to arbitrary levels[3]. When a nested transaction commits, its reads and writes are merged into the parent transaction. Any reads done in a child transaction first check the parent transaction to see if a value for the variable has already been read. If it has, the value that was read in the parent transaction is used instead of reading the value out of the variable itself. If an exception is thrown out of the child transaction that transaction is rolled back and this continues as the stack is unwound until the exception is caught. In other words, if a stack frame for a call to `atomically` is unwound then the transaction that was started by that call to `atomically` will be rolled back. Any transactions which were started by calls to `atomically` before the `try/catch` block that caught the exception was entered will not be rolled back and will continue from the point where the exception was caught. If a child transaction encounters a conflict then all transactions on the stack are rolled back and the root transaction is restarted[4].

Since transactions are restarted in the event of conflicts we cannot have any non-idempotent side-effects take place during the transaction. There are cases where we need to be able to carry out side-effects based on the values read in the transaction though, and in the face of nested transactions we cannot simply carry out the side-effects based on the return value of the transaction (we might still be in a transaction at that point if there is nesting going on). So side-effects can be scheduled to run when the *root* transaction commits by calling the `after` method of `atomic` and passing it a function object. Side-effects scheduled in nested transactions are simply passed up through the parent transactions until the root transaction commits, at which point all the scheduled side-effect function objects are run. `atomic` also has an `on_fail` method that allows side-effects to be run in the event that the transaction has a conflict and fails to commit.

If it is known that a function has side-effects and needs to be prevented from being called within a transaction then the `NO_ATOMIC` macro can be used as one of the function's arguments. `NO_ATOMIC` is an argument with a default value that, when it is created, checks to see if a transaction is currently running. If a transaction is running then an exception is thrown – most likely ending the program. Unfortunately this is a run-time check, there is no way that we know of for our system to enforce this constraint at compile-time in C++.

Waiting for certain conditions among the transactional variables to prevail is accomplished with the `retry` function. When this function is called during a transaction, that transaction's thread is put to sleep until one of the variables that it has read is changed by another thread. At that point the transaction is restarted so that conditions can be checked again. `retry` can optionally be given a timeout that causes an exception to be thrown when the timeout is reached. It is up to the caller of `atomically` to catch this exception. When a child transaction is retried the retry is propagated all the way up to the root transaction which will be restarted when a change is made on another thread.

Another feature of our system related to `retry` is `or_else`. This function takes a list of transactional functions to run and runs the first function in a transaction. If that function completes then the result of that function is returned by `or_else`. If the function calls `retry` then, instead of going to sleep and waiting for a change to one of the variables read in the transaction, `or_else` will call the second function in the list. This continues until one of the functions is able to complete a transaction without retrying, with the result of the function that didn't retry becoming the result of `or_else`. If all the functions retry then the thread goes to sleep until there are changes made on another thread at which point the functions are run again starting at the front of the list.

There can be cases where we don't care about seeing a consistent view of memory. Instead, we only want to prevent races on reading `stm::var` objects. In this case one can use `inconsistently` instead of `atomically`. This function takes a function argument which it calls with an `inconsistent` object which can be used to safely read values from variables. Those values need not be consistent, multiple reads of the same variable can

[3]In order to maintain composibility in our system we have to allow the nesting of transactions. If nesting were not allowed then one would have to know that no transactions are started in every chain of function calls leading to a function that starts a transaction. This is a point where our system differs from Haskell's – nesting of transactions is not possible in their system due to type system constraints (in Haskell's system transactions can only be started from the IO monad and once in a transaction there is no way to safely return to the IO monad to start another transaction short of the current transaction ending).

[4]We have to roll back all the transactions that are ancestors of the child transaction in question to avoid the possibility of an endless loop. If the conflicting value was read in an ancestor transaction and we don't roll back that ancestor transaction then the child transaction will fail over and over again as it tries to validate the stale value that it is getting from the ancestor transaction.

yield different values within the same call to `inconsistently`. Setting variables and starting transactions are not allowed in this context. No validation is done on the reads when `inconsistently` finishes.

Results

Overall our experience with transactional memory has been very positive, the absence of locks makes it much easier to reason about program behavior. Since our system is explicit we trade having to make decisions about fine grained versus coarse grained locking schemes for decisions about what level to transact things at (e.g. store a whole object in a `stm::var` versus giving the object `stm::var` members). The latter seems to have a much lower cognitive load, probably because one doesn't have to constantly worry about introducing deadlock possibilities.

One concern we had when switching to transactional memory was the learning curve. Given the novel nature of this technology it is nearly impossible to hire anyone that has experience with it. It is rare to interview a candidate that has even heard of transactional memory, let alone used it. Luckily our fears proved to be unfounded: we've added two members to the Dynamics team since switching to TM (one right out of university, the other a senior level engineer). Both were able to come up to speed with TM in a reasonable amount of time – i.e. they both seemed to reach a decent comfort level and stop making easy mistakes within a month. There are still more subtle mistakes that one can make that require a more experienced hand to diagnose and fix, but those are rare and we'll touch on them later.

Beyond the basics of transactions the next most useful feature of our system has been `retry`. In fact, managing without it would be extremely difficult since the use of condition variables is problematic in transactional code (one cannot *undo* signaling a condition variable in the event of transaction rollback). `retry` is also much more flexible than condition variables as it is much easier to compose many conditions to wait on.

Given the usefulness of `retry` it's probably surprising that `or_else` has not seen any use in our code. This feature was implemented with the thought that it would be very useful, else why put the time into implementing it. I believe that it may be the case that it was simply forgotten and `retry` was used instead in cases where `or_else` may have yielded simpler code. So it is not the case that `or_else` is useless, it is just not as central to transactional programming as `retry` is.

Another very useful feature is the `after` method of `atomic`. In many cases it provides the only reliable way to interface transactional code to non-transactional code that contains side-effects. The `on_fail` method has less uses as RAII is the better option in most cases where resources need to be released in the event of rollback. It does see some use though.

As detailed above our system is explicit, only values stored in `stm::var` objects are transacted. While this does impose a bit more of a cognitive load than an implicit system as we need to decide what to transact and what not to, it also has some advantages as well. For example, most of our calculations first grab some data and parameters (all values that are transacted). These values are then used in computations that are private to the thread and involve intermediate values that do not need to be transacted. Then the results are stored back into transactional variables. With an explicit system all of this can be done in one transaction as we only have to pay for the transactional reads at the start and the writes at the end. If the user changes the parameters during the calculation then our calculation will be automatically restarted since the parameters are transacted.

With an implicit system one would have to read the data and parameters in one transaction, carry out the calculations outside of a transaction, then start a new transaction where the data and parameters are read again and compared to those obtained in the first transaction. The results are only written if this final check show that there have been no changes to the data or parameters while the calculation was in progress. And if one wants to check for parameter changes during the calculations to avoid doing unnecessary work due to changes made in other threads, the explicit system again has a slight edge as one doesn't have to start new transactions every time the parameters are checked for changes. Instead, the current transaction just needs to be validated.

The implicit system does have the advantage of being easier to program as one doesn't have to make decisions about what to transact. And if a good enough privatization analysis could be done by the compiler then much of the overhead listed above could be done away with. But in this case it seems like a lot of the work has shifted from deciding what to transact to checking to make sure that what you hoped was privatized actually was (or making sure that you are minimizing the amount of work done in your transactions).

Our implementation is based on one global reader-writer mutex. The mutex is locked for reading when a value is read from a `stm::var` for the first time in a transaction (subsequent reads of the same variable simply read the value from the transaction itself which is local to the current thread). A write lock is obtained when committing so that only one commit can proceed at a time and no reads can be done while a commit is in progress. This has worked well, but testing easily shows the the effects of contention for locks on the mutex. Say we have a set of threads, with each thread having its own set of `stm::var` objects that are only accessible in that thread. In each thread we are continually starting transactions, modifying the variables, and committing. In this system we observe an exponential drop-off in the commit rate per thread as more threads are added and they contend for locks on the mutex. This hasn't been much of an issue since the number of threads that we run are limited by the number of cores that our customers have in their machines, two to four for the vast majority of our customers. It is something that will need to be addressed in the future are larger numbers of cores becomes more common place.

Our system is weakly atomic, values that are read from `stm::var` can be modified at will. Short of modifying the language itself I don't know how one would create a strongly atomic system in C++. If a type is stored *by value* in `stm::var` then there's no problem, there's no way for two threads to get a hold of the same instance of an object since objects are always copied into the `stm::var` on `set` and copied out on `get`. For types stored *by reference*[5] we have to enforce that the stored value is either immutable or *internally transacted* (the latter meaning that any mutable state in the object is also transacted). So while our system is weakly atomic we enforce strong atomicity by policy.

As mentioned above, *starvation* can be an issue in transactional memory systems. Fearing this, we implemented the option of *running locked* if a transaction has too many conflicts. We did have some areas of the program where it was suspected that starvation could be an issue so the *running locked* option was applied preemptively. Through much real world testing we've found that we never seem to run transactions locked. This doesn't mean that it could never happen, but our program doesn't seem to have the right kind of contention among transactions to cause starvation to be an issue. We've also never encountered any bugs where starvation was the cause either.

The main problems that we have run into have to do with non-idempotent side-effects in transactions. Most of the time these are mistakes that new-comers make as they ascend the learning curve. An example being a dialog that should be displayed once but ends up being displayed some random number of times as the transaction that it is displayed in has conflicts and is restarted. This type of issue is easily diagnosed and fixed. There have been a few cases of more subtle side-effect bugs which have required much more effort to fix. But these instances have been rare and generally are no more difficult to diagnose and fix than a moderately difficult deadlock.

A related type of issue is caused by our system's handling of transactional reads. Specifically, when a variable is read for the first time in a transaction that value is saved and will always be returned in that transaction (unless the variable is written to in the transaction). This opens the door to using inconsistent values in a transaction. Say we have two transactional variables and we are going to use their values in some calculation. There is a chance that we read the first variable and then another thread changes the value of both variables before we get a chance to read the second. This resultsin the values of the two variables being inconsistent in our transaction. Most of the time this doesn't matter, the validation at the end of the transaction will catch the inconsistency and restart the transaction. But sometimes we need to do something with the values during the transaction that will cause a problem if the values are inconsistent. Say we need to allocate memory where the amount to allocate is given by the difference of the two variables. If the variables are consistent then the first will always be greater than the second and we'll get a valid size for the memory allocation.

[5] In our system storing *by reference* really means storing by `shared_ptr` so the object lifetimes can be ensured between transactions in different threads.

But if the values are inconsistent then we could end up with the first value less than the second yielding a very large size for the memory allocation when the difference is taken, so large that the allocation will fail[6]. One solution would be to validate the transaction on every read, but this would be prohibitively expensive performance-wise. Instead we've opted to live with the possibility of these types of bugs and fix them when discovered by manually inserting a transaction validation to ensure consistency before the values that must be consistent are used. These bugs have proved to be rare and now that we know what to look for they are rapidly diagnosed and fixed. It would be better if this validation was automated though and we are currently looking into ways to do this without causing too large of a performance hit.

Conclusion

Our experience with transactional memory at Wyatt has been extremely positive. The two biggest issues that transactional memory can theoretically have, starvation and side-effects, have either not been an issue at all (the former) or not been that difficult to deal with given a bit of discipline (the latter). While it would be nice to be able to address the side-effect issue through the type system, that just isn't feasible with a library only solution in C++. And compared with the costs of lock-based programming it is a small price to pay. The only remaining issue, inconsistent reads, is less a general problem of transactional memory and more a quirk of how our system handles reads.

Outside of the basics of the reading and writing of transactional variables the most used feature of our TM system is `retry`. The system would be extremely difficult to use without this feature. The related feature of `or_else` turned out to not be used at all, probably due to neglecting to remember that it existed. That is not to say that it couldn't be useful, there could be sections of code that could be simplified through the application of this feature. Due to time constraints we have not attempted to identify such areas in our code at this time.

We also make a lot of use of `after` (to schedule side-effects to take place when the current transaction commits) in order to interface with non-transactional code. The related `on_fail` facility is used much less as RAII can handle most cases where resources need to be released when a transaction is rolled back.

One thing to keep in mind is that our application may be among those ideally suited to the use of transactional memory. For the most part we don't have tight performance constraints: we don't have real-time guarantees to meet, we're not trying achieve a high frame-rate in a game, or beat other stock traders to market. The only place where performance is paramount is in instrument communication where we need to be sure to keep up with the data that the instrument is sending us (in this case we use lock-free queues instead of transactional memory). But once the data gets into Dynamics it is managed via transactional memory from there on out.

We do a lot of non-trivial numerical calculations, but they are structured such that the overhead of the TM system is negligible compared to the calculations themselves. We do a few transactional reads at the start to get data and parameters, the calculations then run without doing anything transactionally, and then the results are written out transactionally. So the overhead consists only of a few reads, a few writes, validation of the small number of reads, and the commit of a small number of writes. Also, while the calculations can take minutes if a lot of data has been collected, by pushing all processor intensive tasks onto background threads and prioritizing calculations based on what the user is looking at in the GUI we can hide any performance issues. This is not to say that we are unconcerned with performance, it is just not at the forefront as it is in other application domains. In other words, we can afford to optimize for developer time at the expense of performance to some degree. Transactional memory has allowed us to do this without giving up too much performance compared to code that uses locks.

[6] Arguably we shouldn't be allocating memory like this in a transaction as it is a side-effect. It can be rendered idempotent through the use of RAII to reclaim the memory in the event of a rollback though. So we allow the allocation of memory in transactions.

Acknowledgments

I'd like to thank Michael Wong for suggesting that this paper be written and for reviewing drafts of the paper.