Transactional Memory in Practice

Brett Hall
Principal Software Engineer
WYATT TECHNOLOGY

“May You Live in Interesting Times”

- often said to be an ancient chinese curse

“May You Live in Interesting Times”

- actually not an ancient Chinese curse

1936: British diplomat hears it, writes it in memoir
"Better to be a dog in a peaceful time, than to be a man in a chaotic period."

“May You Live in Interesting Times”

- actually not an ancient Chinese curse

The multi-core era is our
“interesting times”

- Multi-core era is an “interesting time”,
- forcing us to figure out concurrency
- when dealing with concurrency the Best course would be to share nothing
    - hard to avoid sharing state in most instances though
    - need to synchronize memory access…

Current Synchronization Constructs

- atomics
- mutexes

Atomics

- Too fiddly for programming in the large
- Too fiddly for programming in the small as well except for limited circumstances…

easy to get wrong with disastrous effects

Atomics

- Too fiddly for programming in the large
- Too fiddly for programming in the small as well except for limited circumstances…by people who know what they're doing…

easy to get wrong with disastrous effects

Atomics

- Too fiddly for programming in the large
- Too fiddly for programming in the small as well except for limited circumstances…by people who know what they're doing…who still have a hard time getting it right

easy to get a "really subtle disaster"

Mutexes

- Not too fiddly for programming in the small
- Can work for programming in the large…but don't compose

9

- Work OK for small systems where you can get them to play nicely with each other and avoid deadlock
- But deadlock destroys the ability to compose solutions
- you can try to come up with all sort of ways to avoid it
    - but eventually there's a decent chance that you'll run into something like…

```
void some_function ()
{
    auto lock = std::lock_guard<std::mutex>(some_mutex);
    some_other_function ();
}
```


what does some_other_function do?
need to know to avoid deadlock

```
void some_function ()
{
   auto lock = std::lock_guard<std::mutex>(some_mutex);
   some_library_function_that_you_dont_have_src_for ();
}
```


Have to hope that the documentation tells you whether a lock is taken or not, and hope that the documentation is correct

```
template <typename F>
void some_function (F no_way_to_know_what_this_func_does)
{
    auto lock = std::lock_guard<std::mutex>(some_mutex);
    no_way_to_know_what_this_func_does ();
}
```


All bets are off
have to hope that caller respects any requests you make

```cpp
void some_function ()
{
    auto lock = std::lock_guard<std::mutex>(some_mutex);
    some_other_function ();
}

void some_function ()
{
    auto lock = std::lock_guard<std::mutex>(some_mutex);
    some_library_function_that_you_dont_have_src_for ();
}

template <typename F>
void some_function (F no_way_to_know_what_this_func_does)
{
    auto lock = std::lock_guard<std::mutex>(some_mutex);
    no_way_to_know_what_this_func_does ();
}
```

The “asking nicely” pattern doesn’t scale

- All cases require non-local reasoning about lock order
- All rely on the “asking nicely” pattern, and that doesn’t scale

We could use some better synchronization options

- Actors (aka CSP)
- Transactional Memory

Actors = share very little
actors require more code restructuring
will say what TM is in a minute
transactional memory is more flexible
actors can be implemented on top of TM, but going the other way is difficult

What is Transactional Memory?

- Start here since interview candidates give blank stares when asked about TM

Transactions

```
transaction
{
    const auto a = var_a;
    const auto b = var_b;
    const auto var_c = a + b;
}
```


- Somehow we start a transaction
    - I'll get into more detail in a bit
- transactions behave "atomically"…

“Atomically”, part one

```
//Thread 1
transaction
{
   const auto a = var_a;
   const auto b = var_b;
   const auto var_c = a + b;
}
```

```
//Thread 2

transaction
{
   var_a = 5;
}
```

We see a consistent view of memory in our transaction

- First part of “atomically”
- We need to see a consistent view of memory in our transaction
- transaction behaves as though there’s no other threads running around messing with stuff
    - could implement with one lock, but that would be a fancy way of writing single threaded code

“Atomically”, part two

```
//Thread 1
transaction
{
   var_a = 10;
   //Do some other stuff
   //...
   //...
   //...
   //...
   //...
   //...
   var_b = 5;
}
```

```
//Thread 2



transaction
{
   assert (var_a != 10);
   assert (var_b != 5);
}


transaction
{
   assert (var_a == 10);
   assert (var_b == 5);
}
```

Our writes become visible to other threads all at once

- writes aren’t visible until thread 1 commits
- they all become visible at once
    - if you see one change you can see them all
    - assuming no other thread makes another change, then you might miss a change
- If this is implemented right there’s no way to get deadlock

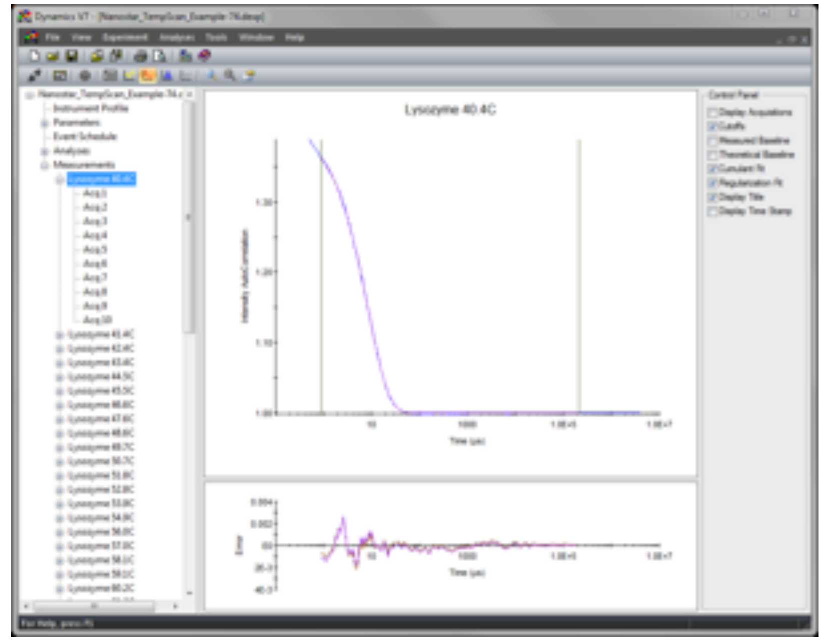
Why did we use
transactional memory?

20

- Go back 9 years to when I started at Wyatt
- when I inherited Dynamics it was in this state (took a year or so to realize this)
    - the architecture was never meant the amount of data it was now being tasked with
    - thread safety was through the “do it in the GUI thread” pattern
    - program would stutter and lock up for minutes at a time as calculations were done
    - impossible to use multiple cores for calculations
    - attempting to fix this lead to …

21

- fixing the threading issues led to a cascading series of changes that was going to require large parts of the program to be rewritten
- competitive pressures for features caused us to shelve fixing this for a few years
- In the interim was playing with Haskell STM
- wrote my own STM system in C++ for kicks
- it worked and TM seemed like a good fit for parts of Dynamics
    - originally only going to use it in a limited scope so that we could easily replace it if there were problems
    - ended up all over the place, more on that later
- The higher ups were more comfortable with using my TM system than with Haskell



- Before we get too much farther, should tell you a bit abut Dynamics
- 800K ~ 900K lines of code
  - not huge, but not trivial either
- files are broken down into measurements that are further broken down into acquisitions
  - calculations on acquisitions are independent
  - measurements depend on their acquisitions, but not on other measurements
  - lots of easy parallelism to exploit

Our System

Starting a transaction

```
template <typename Func>
auto atomically (Func f, const atomically_args& args = atomically_args ())
   -> decltype (f (std::declval<transaction>()));

void run_transaction ()
{
   const auto result = atomically ([](transaction& t) {return 0;});
}
```


- atomically creates the transaction object and passes it to the function that the caller passes in
    - transaction is not copyable and only atomically can create one
- atomically handles validating and committing or rolling back
- more on the "args" in a bit
- transaction has some methods, but we don't need to worry about that for now
- If the transaction function throws an exception then the transaction is rolled back
- kind of a boring transaction…

Transactional Variables

```
template <typename Type>
class variable
{
public:
    const Type& get (transaction& t) const;
    void set (const Type& value, transaction& t);
};

variable<int> var1;
variable<std::string> var2;

void run_transaction_with_variables ()
{
    const auto result = atomically ([](transaction& t)
                                    {
                                        var2.set ("transaction run", t);
                                        return var.get (t);
                                    });
}
```


- … variables make transactions more interesting
- this is the skeletal version, also have some convenience functions for reading, setting outside a transaction
    - the convenience functions run a transaction that just does the get/set
- system is explicit since only variable's are transacted
    - anything else accessed in a transaction is not transacted
    - This makes our system explicit

Optimistic Concurrency

```cpp
variable<int> var_a (10);
variable<int> var_b (5);

//Thread 1
void transaction1 ()
{
   atomically ([](transaction& t)
               {
                  auto a = var_a.get (t);
                  auto b = var_b.get (t);
                  //Do stuff with a and b
                  //...
                  //...
                  //...
                  //...
                  //conflict on var_a
                  //=> roll back and try again
               });
}
```

```cpp
//Thread 2



void transaction2 ()
{
   atomically ([](transaction& t)
               {
                  var_a.set (20, t);
               });
}
```


- we journal the reads and writes
- at end of transaction we validate the reads
    - if a variable was changed by a transaction on another thread we roll back and start over
    - if no changes then we publish our writes to the rest of the system
- we don't validate writes
    - if there was no read of a variable then its old value was of no concern and if it changed it doesn't matter

Nested Transactions

```cpp
variable<int> var;

int not_a_child (transaction& t)
{
    return var.get (t);
}

int child ()
{
    return atomically ([](transaction& t)
                       {
                           return var.get (t);
                       });
}

void parent ()
{
    atomically ([](transaction& t)
                {
                    auto not_child_result = not_a_child (t);
                    auto child_result = child ();
                });
}
```


- have to allow nested transactions else we lose composability
    - library only system in C++ has no choice, Haskell doesn't allow nesting, a compiler based system could disallow it
- have to call atomically when in a transaction to get a child transaction
    - most of the time you want to call a function and pass it the transaction object instead (less overhead)
- when child commits its read and writes are merged into the parent
- more details to come when we get to some examples

SG5 Technical Specification

See N3919 and N4514

- ISO C++ committee study group for transactional memory
- N4514 voted out of committee as a TS at last meeting
- N3919 is more readable and gives motivation for what their doing

Starting a transaction

```
atomic_noexcept
{
   //throwing from here result in undefined behavior
}

atomic_cancel
{
   //throwing from here causes the transaction to rollback
}

atomic_commit
{
   //throwing from here causes the transaction to commit
}
```


- cancel requires the exception to be "transaction safe"
    - limited list of exceptions fits this description
    - supposed to prevent the observation of rolled back actions
- No requirements on implementations except that atomic blocks behave "atomically"
    - don't have to use optimistic concurrency

Transactional Variables

```
atomic_cancel
{
   ++i;
   x = y + 5;
}
```


- everything done in a transaction is transacted
- “implicit system” (the Wyatt system is “explicit”)
- might add “escape actions” later that allow one to marks part of the transaction as not being needed to be transacted
- have to be careful about not accessing the same data in and out of a transaction at the same time
    - leads to data races

Transaction Safety

```
void implicitly_safe ()
{
   //does only "safe" stuff
}

void explicitly_safe () transaction_safe
{
   //does only "safe" stuff, otherwise won't compile
}

void unsafe ()
{
   //does stuff that isn't safe
}

atomic_cancel
{
   implicitly_safe (); // this is OK
   explicitly_safe (); // this is OK
   unsafe (); //won't compile
}
```


- only functions “safe” can be called within a transaction
- “safe” functions only do “safe” stuff (see N4515)
    - generally I/O and volatile access is not allowed
    - things get complicated for virtual functions (transaction_safe_dynamic)
- can be explicitly marked safe, safety of implementation enforced by compiler
- calling other functions in a transaction that aren’t “transaction_safe” won’t compile
- only “safe” stuff can be done in a transaction
- wyatt system has a run-time check for functions that are explicitly marked “unsafe”
    - exception is thrown if unsafe function is called within a transaction
    - best we can do in a library

Nested Transactions

```
void child ()
{
   atomic_cancel
   {
      //...
   }
}

atomic_cancel
{
   child ();
}
```


- nesting transactions is allowed
- works similarly to the Wyatt system

The Canonical Synchronization Example

The Canonical Synchronization Example

Bank Accounts

Simple Transfer

```
//Wyatt
variable<int> account1 (10);
variable<int> account2 (0);

void transfer (int x)
{
   atomically ([](transaction& t)
              {
                 account1.set (acount1.get (t) - x, t);
                 account2.set (acount2.get (t) + x, t);
              });
}
```

```
//SG5 TS
int account1 = 10;
int account2 = 0;

void transfer (int x)
{
   atomic_cancel
   {
      account1 -= x;
      account2 += x;
   }
}
```


- Note: no chance of deadlock if another thread does the same transfer in the opposite order
- No other thread can observe either operation without observing the other
    - can't see debit of account1 without seeing credit of account 2
- No race: If either account balance changes while we're in our transaction then we start over and try again until we can do it without any conflicts

Transfer with notification

```
void send_email ()
{
   //send and email about a transfer
}

void transfer_with_notification (int x)
{
   atomically ([](transaction& t)
               {
                  account1.set (acount1.get (t) - x, t);
                  account2.set (acount2.get (t) + x, t);
                  t.after (send_email); //Sends after commit
               });
}
```


- Going to be sticking with the Wyatt system from now on, SG5 doesn't have some of these features yet
- "After" queues an action to take when the TOP-LEVEL commit finishes
- multiple actions can be queued
- needed to interface with non-transactional code
  - can't just wait for the transaction commit because we might be in a nested transaction
- SG5 is considering something similar

Waiting for a balance

```
void transfer_when_greater (int limit, int x)
{
   atomically ([](transaction& t)
               {
                  if (account1.get (t) <= limit)
                  {
                     retry (t);
                  }

                  account1.set (acount1.get (t) - x, t);
                  account2.set (acount2.get (t) + x, t);
               });
}
```


- can't use condition variables since we don't have mutexes
- retry sleeps until a variable that's been read changes
    - then transaction starts over
- SG5 also considering something along these lines
- the are some issue to look out for, more in a bit

Pitfalls

Pitfalls

Unfortunately it's not all unicorns and rainbows

- never reached "Rainbows and Unicorns" status though
    - see urban dictionary (this one isn't good)

Side-effects

```
void transfer_with_notification_broken (int x)
{
   atomically ([](transaction& t)
               {
                  account1.set (acount1.get (t) - x, t);
                  account2.set (acount2.get (t) + x, t);
                  send_email (); //Sends before commit, maybe more than once
               });
}
```


- Need to avoid side-effects in transactions
    - SendEmail will be called more that once for the same transfer is we have a conflict
    - could be called ANY number of times
- Most of the time its this obvious
    - every once in a while its harder to track down, but not any worse than a difficult to find deadlock
- in SG5 system this couldn't happen since SendEmail wouldn't be transaction safe and the compiler would catch it
- in our case we have to use something else…

NO_ATOMIC

```
void send_email (NO_ATOMIC)
{
   //send and email about a transfer
}

void transfer_with_notification_broken_but_not_as_bad (int x)
{
   atomically ([](transaction& t)
               {
                  account1.set (acount1.get (t) - x, t);
                  account2.set (acount2.get (t) + x, t);
                  send_email (); //crashes here, normally caught in testing
               });
}
```


- declares an arg of type “no_atomic”
  - “no_atomic” constructor throws if it is called within a transaction
- Get a run-time error when SendEmail is called in transaction
- caught be testers most of the time

Starvation

```cpp
variable<double> value;

//thread 1
void long_transaction ()
{
   atomically ([&](transaction& t)
               {
                  auto val = value.get (t);

                  //do a bunch of stuff
                  //...
                  //...
                  //yep, still doing stuff
                  //...
                  //...
                  //...
                  //...
                  //...
                  //need to do anough stuff so
                  //that the transaction on
                  //thread 2 can commit and cause
                  //us to have a conflict every time
               });
}
```

```cpp
//thread 2



void short_transaction_that_causes_conflicts ()
{
   while (true)
   {
      atomically ([&](transaction& t)
                  {
                     auto old_value = value.get (t);
                     value.set (old_value + 5.0, t);
                  });
   }
}
```


- short transaction continually causes conflicts in a longer transaction
- until a couple of months ago we had never seen this happen
- One of the optional arguments to atoimically allows you to specify how may conflicts are acceptable
    - either throws an exception when the limit is hit or “runs locked”
        - “run locked” = no other thread can commit until our transaction does
        - controlled by another argument
    - had put this in preemptively in some spots, but had never seen the code fire
    - of course has a bug when we finally needed it (weird edge case we didn’t have a unit test for)
    - heavy-handed response, but works for us given how rare this is
        - other ways to handle, but much more complicated to implement

Retry Deadlock

```
variable<bool> value1 (false);
variable<bool> value2 (false);

//thread 1
void set_1_and_wait_for_2 ()
{
   atomically ([](transaction& t)
               {
                  value1.set (true, t);

                  if (!value2.get (t))
                  {
                     retry (t);
                  }
               });
}
```

```
//thread2
void set_2_and_wait_for_1 ()
{
   atomically ([](transaction& t)
               {
                  value2.set (true, t);

                  if (!value1.get (t))
                  {
                     retry (t);
                  }
               });
}
```


- retry is extremely useful, but has some issues
- two threads, each is waiting on the other to do something that will be done once the other thread commits, but neither can commit because they're waiting on the other
- never saw this in really code, but is theoretically possible
- really we need to split up the transaction…

Split the transactions

```
variable<bool> value1 (false);
variable<bool> value2 (false);

//thread 1
void set_1_and_wait_for_2 ()
{
    atomically ([](transaction& t)
                {
                    value1.set (true, t);
                });
    atomically ([](transaction& t)
                {
                    if (!value2.get (t))
                    {
                        retry (t);
                    }
                });
}
```

```
//thread2
void set_2_and_wait_for_1 ()
{
    atomically ([](transaction& t)
                {
                    value2.set (true, t);
                });
    atomically ([](transaction& t)
                {
                    if (!value1.get (t))
                    {
                        retry (t);
                    }
                });
}
```


Fixes things, but what if some bozo comes along and does this…

Nested, split transactions

```
variable<bool> value1 (false);
variable<bool> value2 (false);

//thread 1
void set_1_and_wait_for_2 ()
{
    atomically ([](transaction& t)
               {
                   value1.set (true, t);
               });
    atomically ([](transaction& t)
               {
                   if (!value2.get (t))
                   {
                       retry (t);
                   }
               });
}

atomically ([](transaction& t)
           {
               set_1_and_wait_for_2 ();
           });
```

```
//thread2
void set_2_and_wait_for_1 ()
{
    atomically ([](transaction& t)
               {
                   value2.set (true, t);
               });
    atomically ([](transaction& t)
               {
                   if (!value1.get (t))
                   {
                       retry (t);
                   }
               });
}

atomically ([](transaction& t)
           {
               set_2_and_wait_for_1 ();
           });
```


- broken again
- modified from example given by Hans Boehm to the SG5 mailing list
    - SG5 is wrestling with this now in discussions of retry in TS
- Fixed by applying NO_ATOMIC to the set_…. functions
    - relying on commit happening can be considered a side-effect
- Can be more subtle, the second transaction could be buried in library code

Inconsistent Values

```cpp
variable<size_t> value1 (10);
variable<size_t> value2 (2);

void thread1 ()
{
   atomically ([](transaction& t)
               {
                  auto a = value1.get (t); //a = 10
                  //do some stuff
                  //...
                  //...
                  //...
                  //...
                  //...
                  //...
                  //...
                  //until the other thread has committed
                  auto b = value2.get (t); //b = 11
                  auto values = std::vector<int>(a - b); //oops
               });
}
```

```cpp
void thread2 ()
{
   atomically ([](transaction& t)
               {
                  value1.set (15, t);
                  value2.set (11, t);
               });
}
```


- quirk of our system
- value1 > value2 is a class invariant that ensures we only allocate a sane about of memory
    - in this case we will probably try to allocate a 4gb vector
- normally our transaction would have a conflict and we'd restart and see the consistent values
    - memory allocation is a side-effect, but most of the time it's idempotent (especially if RAII is used) so it doesn't matter
- to solve this case use…

Validate

```
variable<size_t> value1 (10);
variable<size_t> value2 (2);

void thread1 ()
{
    atomically ([](transaction& t)
                {
                  auto a = value1.get (t); //a = 10
                  //do some stuff
                  //...
                  //...
                  //...
                  //...
                  //...
                  //...
                  //...
                  //until the other thread has committed
                  auto b = value2.get (t); //b = 11
                  validate (t); //restart here until no conflicts
                  auto values = std::vector<int>(a - b);
                });
}
```

```
void thread2 ()
{
    atomically ([](transaction& t)
                {
                  value1.set (15, t);
                  value2.set (11, t);
                });
}
```


- validate() runs transaction validation and restarts transaction if there's a conflict
    - can't get to the memory allocation if things are inconsistent
- has only happened a few times
- Looking into ways to avoid this problem without killing performance

Fine vs. Coarse Grained

```
struct fine_grained
{
   variable<int> m_member1;
   variable<int> m_member2;
   variable<int> m_member3;
   variable<int> m_member4;
};

struct coarse_grained
{
   int m_member1;
   int m_member2;
   int m_member3;
   int m_member4;
};
variable<std::shared_ptr<const coarse_grained>> m_coarse;
```


- anything shared between threads needs to be transacted
- decision is what level to transact at
  - fine grained: individual members are transactional variables
    - good for often updated structure
  - coarse grained: individual members not transactional, but whole structure is always stored immutably in a transactional variable
    - good for read lots, write rarely
    - can become point of contention if too much writing going on

Weak Atomicity

```
void exploiting_weakness ()
{
    exhaust_port* port = atomically ([](transaction& t)
                                     {
                                         return fly_down_the_trench (t);
                                     });
    shoot_proton_torpedo (port);
}
```


- our system is weakly atomic
- the exhaust port is stored in a transactional variable
    - our system can't stop you from mutating the object outside of a transaction
        - defeats the whole point of having the system
- to guard against this…

Require Immutable or “Internally Transacted”

```
//Immutable
variable<std::shared_ptr<const exhaust_port>> port;

//Internally Transacted
struct exhaust_port
{
   //all mutable state is stored in transactional variables...
   variable<ray_shielding> m_shielding;

   //...or acessed through transactional methods
   void shoot_with_proton_torpedo (transaction& t);
};
```


-… as a policy we required stuff stored in transactional variables to be either immutable or “internally transacted”
- “internally transacted” = all mutable state is in transactional variables
- enforced through code review

Invasive

- original plan was to only use TM in data store and maybe experiment parameters
    - it sort of crept like the kudzu above to take over a lot more
    - this was due to TM usefulness and complication of interfacing transactional and non-transactional code
- Used TM for everything except low-level instrument communication (data reception)
- GUI isn't transactional (MFC), but calls transactional code

No one's heard of it

- When I ask about TM in job interviews usually just get blank stares in response
    - Hasn't really mattered, people come up to speed quick
- But there's also not a lot of places to turn for "best practices"
    - lots of academic papers on implementing TM systems
    - but you're mostly on your own when it comes to figuring out how to apply the system
- that's it for pitfalls…

Performance?

- Much like this car: not the fastest, but fits our purposes well

Performance

- Dynamics is not a “low latency” application (for the most part)
  - Not a game or trading bot
  - We’re good at “hiding” our latencies
  - Calculations are almost ideally set-up for transactional memory
  - Some instrument communication is low latency, but doesn’t use TM
- With these caveats we get plenty of performance

- Dynamics isn’t a low latency app
- Calcs taking even 10 seconds extra (out of minutes) is acceptable
- push as much as possible onto background threads (TM facilitates this)
- prioritize calcs based on what the user is looking at
  - sometimes the user is looking at something that requires everything to be calculated, they just have to wait but the GUI stays responsive
- Data reception has latency requirements
  - have to keep up with data stream or get disconnected
  - don’t use TM here (lock free queues instead)
- Performance is good, even with our bone-headed implementation
  - as more cores become common place we will need to upgrade out implementation so it scales better

Calculation Structure

```
calculate ()
{
   params = getParams ();
   data = getData ();
   auto res = CalculateResult (params, data);
   StoreResult ();
}
```


- Calculations were the main thing we wanted to push into background threads
- Calculations are structured such that only a few transactional reads need to be done at the start and then some transactional writes at the end
- the transactional overhead negligible compared to what Calculate result is doing

Performance

- More overhead
- Bigger non-determinism price to pay than with mutexes
- Can have less contention

- TM May Still be using mutexes under the covers
    - otherwise using lock-free stuff
        - don't know how to do retry without a mutex and condition variables
    - either way still have to pay sync price
    - Also have to pay for journaling and validation (and maybe non-locality if system is indirect)
- Don't hold locks for long, so less contention possibility (especially for read only stuff), but…
- When there is contention you might have to pay higher price
    - repeated work when there's conflicts
    - an example…

Performance

```cpp
std::mutex m;
int a;
int b;

void use_a_and_b ()
{
   auto lock = std::lock_guard<std::mutex>(m);
   //do a bunch of read-only stuff with a and b
}
```

```cpp
variable<int> a;
variable<int> b;

void use_a_and_b ()
{
   atomically ([](transaction& t)
               {
                  //do a bunch of read-only stuff with a and b
               });
}
```


- Only one thread can call use_a_and_b at a time when using mutexes
- Any number of threads can use the transactional version
    - transactional version might have more overhead, but the contention win might weigh in its favor as long as writes are RARE
- the mutex version can probably be restructured to have less contention (or use a shared_mutex)
    - that has to be done by hand
    - in transactional case its automatic

Hardware Transactional Memory

- Exists today: Intel TSX
  - Transacts cache-lines
  - Implicit
  - Bounded

- haven't actaully used it, just read about it
- was in high-end Xeon Haswell chips
  - silicon had bug
  - disabled until later broad well chips, still high end only
  - in skylake, might be more evenly distributed
- cache-lines transacted => have to watch out for cache ping-ponging
- implicit like TS
  - all memory access is transacted while in a transaction
- Bounded
  - limited buffers for reads and write
  - have to fall back on software TM if cache spills
  - probably an expensive context switch

How’d it work out?

- TL;DR: Great!
  - Just keep in mind what our app is like
- New people get up to speed quickly
- More fun than mutexes
  - Optimizes for programmer time

- Had two new people join the team since we started using it
  - one junior, one senior
  - Neither had TM experience
- learning curve is short
  - Usually stop making obvious mistakes in less than a month
- mutexes set the bar pretty low
  - TM has lower cognitive load for the most part

Open Source?

- Always get asked about this
- last year the answer was “no” when I gave a lightning talk,
- this year the answer is…

Open Source?

Soon!

- Got the go ahead from the higher ups at Wyatt to do it a few weeks ago
- Haven't had time to clean it up
    - some embarrassing code that we've been living with that needs to be fixed
        - open sourcing it is already improving our code quality
- Other goodies along with TM system
    - DeferredResult, Channel, etc.
- watch my blog for details…

Resources

- My blog: https://backwardsincompatibilities.wordpress.com/
- Wyatt: http://www.wyatt.com
- SG5 papers:
  - N3919: motivation and details
  - N4514: standard-ese
  - N4438: my paper about our experience at Wyatt
- GCC experimental TM system

- blog url is in my conference profile
- No reference implementation for SG5 stuff yet, but plans for GCC and Clang implementations
- for gcc just google “gcc transactional memory”
  - has been in there since 4.7
- Intel had an experimental compiler
  - it’s listed as “retired” on their website