# Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached

Trilok Vyas, Yujie Liu, and Michael Spear

Lehigh University

# High-Level Overview

- Much work toward standardizing TM in C++
  - Draft specification v1.1 Feb 2012
  - Draft specification v1.0 from 2009
- Two implicit goals for the specification:
  - Simplify creation of new code
  - Enable transactionalization of existing code
- We focused on the latter problem
  - Goal: provide feedback on what works, what doesn't work, and what could be better
- The paper and talk are a collection of observations, experiences, and findings

# Why Memcached?

- It's important in its own right
  - Lots of users
- Expectation of scalability
  - Sets a high bar
- Mature code
  - Representative of good legacy lock-based programs
- It is nuanced
  - I/O, reference counting, function pointers, etc
- It's not too big
  - Just a hash table… <10 KLOC

# GCC and the C++ TM Specification

- Transaction blocks and expressions
  - `__transaction_relaxed` and `__transaction_atomic`
  - Can wrap a block of code (in '`{}`') or an expression (e.g., `if (__transaction_atomic(x++))`)
- Annotations
  - `[[transaction_callable]]` and `[[transaction_safe]]`
  - Instruct compiler to create instrumented clone of function
  - Identify code that can/can't be called from atomic transactions
- Exception-related stuff
  - E.g., to abort *and not retry* a transaction that encounters an exception
  - Not evaluated in this work… memcached is a C program

# Meta-Issues

- Transactionalizing is easy… optimizing is hard
- Difference between atomic and relaxed is confusing
  - Our approach: use relaxed only when you can't use atomic
    - E.g., I/O, library functions that aren't marked safe
  - We expected this to give a mental performance model
    - Relaxed transactions are likely to serialize (though not always)
    - Atomic transactions should scale unless there are conflicts (no hazards)
  - Our attitude probably would have been different if we were using cancellation and/or exceptions

# Issue #1: Collateral Transactionalization

- Using mutrace, we determined that the **`cache_lock`** and **`stats_lock`** are highly contended
  - **`cache_lock`** protects the main hash table
  - **`stats_lock`** protects a bunch of counters
  - Clearly, contention is for different reasons
- It's not enough to focus on these two locks
  - Locking order is **`cache_lock`**, **`slabs_lock`**, **`stats_lock`**
  - Once a (relaxed) transaction acquires a lock, it must be serial and irrevocable
  - Ultimately, needed to transactionalize **`slabs_lock`** too
    - Otherwise most transactions would be relaxed, might serialize
  - Note: transactionalizing any lock is an all-or-nothing proposition

# Issue #2: Condition Synchronization

- Memcached uses **`pthread_cond_t`**, which is tied to a mutex (e.g., **`cache_lock`**)
  - C++ specification has no support for condition synchronization
  - Short-term solution: decouple the condition variable from the lock that we are replacing
  - Correct for this work, probably not in general

- Some open questions:
  - How should TM support condition synchronization?
    - Signaling from a relaxed transaction is OK… waiting isn't
  - What does it mean to signal in the middle of a critical section?
    - Analogy to Hoare and Mesa monitors
  - What does it mean to signal in the middle of a relaxed transaction that might communicate?

# Issue #3: Atomic/Volatile Variables

- C++ **`atomic<>`** variables and C **`volatile`** variables cannot be accessed from an atomic transaction
  - And they force serialization in relaxed transactions
- We replaced all volatiles with transaction expressions
  - No changes to lines of code
  - Would be hard in bigger program
  - Would be nice if compiler optimized the resulting transactions
- Are semantics of transactions strong enough?

# Issue #4: Semantics

- We used a privatization-safe algorithm
  - In-place update, quiescence at commit time
- Is in-place update OK in C++?
  - If program has data races, this could lead to out of thin air reads, otherwise it's OK
- Is privatization safety necessary? Could we use polling instead of quiescence?
- Tools needed to answer these questions

# Issue #5: Annotations

- "Viral" **`transaction_safe`** annotations are *annoying*
  - Necessary for 'atomic' transactions, or code won't compile
  - Why atomic? We think it's a performance model… good to know what forces serialization
- Non-viral **`transaction_callable`** are *hard*
  - I'll take annoying over hard any day ☺
  - Another opportunity for tool designers
- A big surprise
  - While many standard library functions are **`transaction_safe`**, some important ones are not
  - Any un-marked function can only be used in a relaxed transaction, and *will* force serialization *if executed*

# Annotations: Some Unsafe Functions

- Calls to pthread library:
  - **`pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_trylock, pthread_cond_signal`**
- Exceptional behaviors and debugging messages:
  - **`assert_fail, abort, builtin_fwrite, fprintf, perror`**
- String functions:
  - **`builtin_strncpy, memcmp, snprintf, strlen, vsnprintf`**
- Allocation and variable arguments:
  - **`realloc, builtin_va_end, builtin_va_start`**

# Issue #6: Safe Library Functions

- How should these unsafe functions be handled?
  - Some require redesign of whole libraries (e.g., pthread locks)
  - Could rewrite others without unsafe features (e.g., no inline asm)
    - This would slow the common (nontransactional) case
- There should be a way to generate *only* a transactional version of a function
  - Risky from a maintenance perspective
  - Hard to standardize, since mechanism likely to be compiler-specific

# Issue #7: Commit Handlers

- We didn't restructure control flow
  - **`pthread_cond_signal`** could have been moved to the end of a transaction, but it would have been ugly
  - A better solution is onCommit handlers
    - GCC already has them, but we didn’t use them
    - They should be added to the specification
- We did not see a need for onAbort handlers
  - Contention Management could be an onAbort hander
  - Otherwise, these seem to fall into the category of topics that only matter to *library* STM implementations

# Issue #8: Assertions & Serialization

- We removed all assertions to prevent reliance on relaxed transactions
  - Relaxed *should* scale as well as atomic
  - But we couldn't find other functions that needed to be marked **`transaction_callable`** otherwise

- It didn't help
  - Usually assertions weren't the only unsafe code

- Better run-time reporting of commits, aborts, and irrevocability would be good
  - Might be good to specify "always irrevocable" transactions
  - Too often, we knew a relaxed transaction would spend time on instrumentation, only to ultimately become irrevocable in a late function call

# Issue #9: Nesting

- Maximal transactionalization led to lots of nesting
  - Good from ease of programming perspective
  - But we're trained to believe that nesting has a cost… tempting to optimize
- Not clear: are simple transaction expressions optimized? If so, how would this affect nesting?
- Also not clear: should a programmer try to optimize to avoid nesting?
  - Copying code based on calling context
- What if there was true closed nesting? How would this advice change? Would it become more important?

# Issue #10: Lock Order Changes

- Consider the following critical section:

```
Lock(cache_lock);
Lock(slabs_lock);
...
Unlock(slabs_lock);
Unlock(cache_lock);
```

```
Lock(slabs_lock);
Lock(cache_lock);
...
Unlock(cache_lock);
Unlock(slabs_lock);
```

```
Lock(slabs_lock);
transaction {
...
}
Unlock(slabs_lock);
```

- Replacing **`cache_lock`** with a relaxed transaction could reduce concurrency
  - Collateral serialization of **`stats_lock`** transactions
- Reordering the locks makes **`slabs_lock`** accesses transactional *while holding the lock*
- Reordering also violates the program-wide lock order
- It's correct, but it feels strange to make changes like this

# Issue #11: Lock Granularity

- Consider these code blocks:

```
Lock(stats_lock);
stats.stat1++;
Unlock(stats_lock);
if (...)
  Lock(stats_lock);
  stats.stat2++;
  Unlock(stats_lock);
```

```
// x is volatile
if (x == 1)
  ...
else if (x == 2)
  ...
else
  ...
```

- On the left, should this be flattened into one transaction?
  - Aren't lock acquires expensive?
- On the right, this looks like a control flow bug
  - But it could be intentional, and necessitate true volatiles and relaxed transactions
- If nothing else, these problems illustrate the importance of documentation

# Issue #12: Abstract Nesting

- The **`stats_lock`** is highly contended but not scalable.
  - Nesting it inside of **`cache_lock`** could be a performance killer
  - Probably need something clever here, such as Abstract Nested Transactions (ANTs)
    - Can these be inferred automatically, or do we need language support?
- Perhaps it's time for a paraphrase?

"Every interesting problem in the systems community was solved 10 years ago by the database community"

(Butler Lampson)

"Every interesting problem in the TM community was solved 5 years ago by Tim Harris"

# Issue #13: Cross Platform Concerns

- Memcached is designed to run on all sorts of platforms
  - Even those without cheap atomic increment/decrement operations

- How can we retrofit TM into this code without doubling the code size or bloating everything with horrid macros?

# Evaluation

- Ran on dual-chip Xeon X5650 (12 cores/24 threads)
  - Generated workload with memslap
  - Pinned memcached threads to one CPU, memslap threads to the other

- Seven bars:
  - **Baseline**: only changed condition variables
  - **CacheLockNaive**: minimal effort, only replaced `cache_lock`
  - **CacheLock**: maximal effort, only replaced `cache_lock`
  - **SlabsLock**: maximal effort, only replaced `slabs_lock`
  - **StatsLock**: maximal effort, only replaced `stats_lock`
  - **Cache+Stats**: maximal effort, replaced two locks
  - **Cache+Stats+Slabs**: maximal effort, replaced all three locks

# Warning: Down is Good

Time (seconds)

120
100
80
60
40
20
0

1 2 3 4 5 6 7 8 9 10 11 12

Threads

Baseline
CacheLockNaive
CacheLock
SlabsLock
StatsLock
Cache+Stats
Cache+Stats+Slabs

- Every effort to improve performance failed
  - Sometimes quite dramatically

# Why the Bad Performance?

- We replaced carefully tuned locks with untuned transactions
  - Especially w.r.t. contention management
- Large, uncontended **`cache_lock`** transactions inherited **`stats_lock`** conflicts
  - Need ANTs?
- Latency
  - Transactions are slower at one thread: TM implemented as shared library, lots of function calls, …
  - Relaxed transactions that serialize late keep the high latency, don't add scalability

# Conclusions

- The C++ TM Spec is a leap in the right direction
  - Hopefully, feedback like this will help to improve it further
- It's a moving target, but don't let that discourage you
  - Likely that recent changes will necessitate a v2.0… changes to annotations, and to exceptions
  - Not sure when new spec will be implemented
- Huge opportunities for research, especially tools and new linguistic constructs

# Questions?

With special thanks to our reviewers, and to the GCC maintainers for their quick responses when we found bugs

Trilok just finished his MS degree, is on the job market

This code will be open-sourced soon… until then, please email me for a copy