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

Trilok Vyas, Yujie Liu, and Michael Spear
Lehigh University
{trv211, yul510, spear}@cse.lehigh.edu

Abstract

The addition of transactional memory (TM) support to existing languages provides the opportunity to create new software from scratch using transactions, and also to simplify or extend legacy code by replacing existing synchronization with language-level transactions. In this paper, we describe our experiences transactionalizing the memcached application through the use of the GCC implementation of the Draft C++ TM Specification. We present experiences and recommendations that we hope will guide the effort to integrate TM into languages, and that may also contribute to the growing collective knowledge about how programmers can begin to exploit TM in existing production-quality software.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Design, Performance

Keywords Transactional Memory, memcached, GCC, C++, specification

1. Introduction

For over a decade, Transactional Memory (TM) [6, 18] has been promoted as an efficient programming idiom for simplifying the creation of concurrent software. In recent years, the C/C++ community has made great strides toward integrating TM into mainstream products, and today there is both a draft specification [1] and a set of reference compiler implementations (GCC, Intel CC, LLVM, and xlC).

The integration of TM into C++ enables its use in two directions. First, it allows programmers to create new software from scratch that is designed around transactional constructs. Second, it enables existing software to be retrofitted with transactions, either to simplify the creation of new features, or to improve (from a performance or maintenance perspective) existing code.

In the former category, studies by Rossbach et al. [16] and Pankratius and Adl-Tabatabai [14] have shown that TM can simplify the creation of new software. Similarly, there are several microbenchmark and benchmark suites that demonstrate the use of TM, most notably STAMP [13], EigenBench [7], Atomic Quake [21], Lee-TM [2], SynQuake [11], and RMS-TM [9].

While these benchmarks and studies collectively provide a good platform for evaluating TM implementations, they largely treat the STM interface as a constant. Our effort in this paper is complementary: we are interested in evaluating the proposed C++ TM interface, with an eye towards (a) what common programming patterns and idioms are not well supported, and (b) what features are cumbersome or difficult to use.

* This work was supported in part by the National Science Foundation through grants CNS-1016828 and CCF-1218530.

For the purpose of this study, we used the open-source GCC compiler to replace locks with transactions in the popular memcached in-memory web cache. We chose GCC version 4.7.1, which implements the draft C++ TM specification, for two reasons. First, it is widely available and easy to modify, which suggested that if we encountered bugs, we would be able to remedy them quickly. Second, its relatively transparent TM library should ultimately enable us to investigate the behavior of different TM algorithms on this workload.

Similarly, we chose memcached for two reasons. First, it is a popular and realistic application, but one that is still of a tractable size. It was not unreasonable to analyze all language-level locks within the application, or to conduct whole-program reasoning about the correctness of code transformations. Second, it is sufficiently complex to represent a challenge for TM: locks either spin or block; locks and condition variables are intertwined; there is a statically defined order in which locks must be acquired; and there is a reference counting mechanism that employs atomic operations and volatile variables (i.e., C++11 `atomics`). In our study, we used memcached version 1.4.13, which contains recent scalability enhancements. While our work is similar to a recent effort by Pohlack and Diestelhorst [15], it differs in that we are using memcached to analyze the C++ specification, rather than to assess the utility of a particular TM mechanism.

The remainder of this paper is organized as follows. Section 2 describes the key features of the Draft C++ TM Specification that we evaluate in this paper. Section 3 introduces core themes and challenges when transactionalizing memcached, and Section 4 describes our experience attempting to achieve the most complete transactionalization of the `cache_lock`. Section 5 discusses issues that arise when replacing multiple locks with transactions. We present preliminary performance results in Section 6, share our thoughts about the central constructs of relaxed and atomic transactions in Section 7, and then conclude in Section 8.

2. The Draft C++ TM Specification

The Draft C++ TM Specification [1] integrates TM into C++ through the addition of several new keywords. These keywords can be roughly broken down into three categories.

Transaction Declarations: Most substantially, the specification introduces the `__transaction_atomic` and `__transaction_relaxed` keywords. These keywords are used to indicate that the following statement or lexically scoped block of code is intended to execute as a transaction. There are numerous interpretations of the difference between these two keywords. For our purposes, *atomic* transactions can be thought of as being statically checked to ensure that they contain no unsafe operations. While the specification does not guarantee that the meaning of *unsafe* will not evolve over time, a reasonable approximation is to assume that atomic transactions

cannot perform I/O, access volatile variables (i.e., C++ atomics), or call any function (to include inline assembly) that the compiler cannot prove will be safely rolled back by the TM library if the calling transaction aborts.

Relaxed transactions can be thought of as not carrying the same restrictions as atomic transactions. That is, they are allowed to perform I/O and other unsafe operations. This is achieved by enabling a relaxed transaction to become irrevocable [19, 20], or perhaps to run in isolation, starting at the point where it attempts to do something that the compiler is not certain can be undone. Such transitions are invisible, though they may present a scalability bottleneck and can introduce the possibility of deadlock (e.g., if two relaxed transactions attempt to communicate with each other through atomic variables, then since both must become isolated and irrevocable, they cannot run concurrently). However, in the absence of such unsafe code, the two types of transactions are indistinguishable; both types of transactions should scale equally well.

In addition, the specification supports *transaction expressions*. Transaction expressions are syntactic sugar to simplify code such as using a transaction to initialize a variable, or using a transaction to evaluate a conditional.

Function Annotations: The specification supports two function annotations, transaction_safe and transaction_callable. A *safe* function is one that contains no *unsafe* operations. That is, it can be called from an atomic transaction. The compiler statically checks that safe functions only call safe functions, and that atomic transactions only call safe functions. In addition, the compiler must generate two versions of any safe function: the first is intended for use outside of transactions; the second contains instrumentation on every load and store, so that the function can be called from within a transaction (and safely unwound upon abort). The compiler generates an error if it encounters a function that is marked *safe* but contains unsafe operations.

The *callable* annotation indicates to the compiler that a function will be called from a transactional context, but is not safe. We interpreted this as indicating that it is possible, but not guaranteed, that the function will call unsafe code; this, in turn, means the function can only be called from relaxed transactions. In contrast to the transaction_safe annotation, transaction_callable is strictly a performance optimization. An implementation is free to execute all relaxed transactions serially, or to try to execute them concurrently as long as no running transaction requires irrevocability in order to perform an unsafe operation. If a relaxed transaction attempts to call a function that the programmer has not annotated as callable or safe, then unless the compiler has inferred the safety of that function, the transaction must become irrevocable and serial.

Based on this interpretation, we concluded that if a function cannot be marked safe, on account of possibly performing I/O or some other unsafe operation, then it should be marked callable to ensure that calls to that function from a relaxed transaction do not cause serialization in those instances where the function does not perform an unsafe operation. The canonical example for this feature is assertions: they should not force a relaxed transaction to become irrevocable unless the asserted predicate evaluates to false.

Exception Support: The third category of extensions in the Draft C++ TM Specification pertain to exceptions. When a transaction encounters a throw statement, failure atomicity may require the transaction to undo all of its effects; in other cases it may be desirable for the transaction to commit its partial state. To express this difference, the specification provides the __transaction_cancel statement.

Clearly, an irrevocable relaxed transaction cannot undo its effects, and indeed it is not correct in the current specification for any relaxed transaction to cancel itself. Atomic transactions may explicitly cancel themselves, and may even do so in the absence of exceptions. This, however, creates a challenge: with separate compilation, the compiler may not be able to determine whether a transaction_safe function called by a relaxed transaction will attempt to cancel. To remedy the problem, an additional may_cancel_outer annotation is required on some safe functions.

3. Memcached Analysis and Base Transformations

We began by using mutrace to identify highly contended locks within memcached. The tool identified two pthread_mutex locks as having enough contention to justify our attention:

- cache_lock: This lock prevents concurrent access to the underlying hashtable that serves as the central data structure in the application.
- stats_lock: This lock protects a set of counters for program-wide statistics. While much effort has gone into moving these counters into per-thread structures, several remain as global variables.

Finding #1: Collateral Transactionalization Memcached also contains a global slabs_lock. This lock protects the slab allocator. Slabs are memory blocks that store sets of objects of the same maximum size. Mutrace did not identify this lock as highly contended. However, the default locking order in memcached is cache_lock, slabs_lock, stats_lock. This means that in some cases, replacing the cache_lock with a transaction will require a relaxed transaction (and typically a serial irrevocable one), since the transaction will then acquire the slabs_lock. While it would be ideal to only replace contended locks with transactions, it is unwise to replace locks with transactions that are known to immediately become irrevocable. When transactionalizing an application, programmers should expect that they will need to modify more critical sections than just those protected by the locks they have profiled and identified as highly contended.

Finding #2: Condition Synchronization Support is Needed The slabs_lock and cache_lock are used by memcached both for protecting critical sections, and as the lock object for condition synchronization using pthread_cond_t variables. The current TM specification does not have support for condition variables, and thus we were required to manually transform all condition synchronization.

In the case of memcached, condition synchronization is simple: whenever a thread wakes from a pthread_cond_wait() call, it immediately releases the associated lock, takes a back edge in a while loop, and then attempts to re-acquire the lock. Consequently, it is safe to add a new lock to the program, used only for condition synchronization. This change added 24 lines of code, not counting comments. After verifying that this transformation did not affect the performance or correctness of the code, we were able to progress with our effort to replace the cache and stats locks. However, it is unclear whether this technique generalizes to most pthread-based programs. If the programmer expects for a thread to wake *in the middle of a critical section*, the lack of support for condition variables will be a significant obstacle.

Finding #3: Volatile/Atomic Variables Many critical sections in memcached access volatile variables as part of periodic maintenance operations or condition synchronization. However, the released version of GCC 4.7.1 contained a bug that prevented access to volatiles even from within relaxed transactions. While the GCC maintainers fixed this bug within a day, we realized that these accesses would force most transactions to serialize, and therefore we

opted to replace all volatile variable accesses with transactional accesses to non-volatile variables.[1]

This transformation was straightforward. We simply renamed volatile variables, traced all compilation bugs, and replaced all errors with transactions accessing the new non-volatile variables. In all, we only changed three variables, and through transaction expressions, the total lines-of-code count did not change. However, this change raises two questions. First, memcached only consists of 7400 lines of code, and it is not clear that such a change would be realistic to push through a larger program. Second, if such a transformation is going to be frequent when transactionalizing legacy code, then the specification must guarantee that the semantics of transactions is no weaker than the semantics of C++ atomic variable accesses. That is, a transaction expression for reading a variable must have the same ordering guarantees as a read of an atomic variable, and a transaction that sets a variable's value must have at least the same ordering guarantees as a store to a C++ atomic.

Finding #4: Semantics One final issue that cannot be overlooked is that our effort entailed replacing locks with transactions. Strictly speaking, the default TM algorithm in GCC is privatization safe, achieving single global lock semantics [12] through the use of a quiescence mechanism at commit time. This mechanism is strong enough to permit transactions to immediately `free` memory upon commit, without using an epoch-based deferred memory reclamation policy [4, 8].

Considering that memcached uses timestamps to age data in a cache, and thus already has some manner of delayed reclamation, it is possible that a deferred reclamation policy would be acceptable, even if it led to greater heap consumption. More significantly, in our analysis of memcached we did not observe any instances of privatization. Taken together, these observations suggest that a much more scalable STM implementation might be viable for this workload, such as LSA [3]. However, a production system cannot offer such algorithms as run-time options unless it also delivers tools that can identify instances of privatization. To make the use of such algorithms legal, the specification will need to provide complementary linguistic mechanisms.

4. Replacing `cache_lock`

Given the lock contention in memcached, `cache_lock` appeared to be the most profitable to replace with transactions. We began by simply eliminating the lock, and replacing all critical sections with transactions. The biggest challenge we faced was in determining which transactions could be atomic, and which must be relaxed.

We began by trying to use atomic transactions everywhere. This immediately led to the compiler generating errors whenever a transaction attempted to call a function that was not marked as `transaction_safe`. Marking these functions, in turn, led to new errors when the marked functions called unsafe functions. After tracing all such calls, we arrived at a program with 10 atomic transactions and 14 relaxed transactions. Along the way, we had to annotate one function pointer that called a `transaction_safe` function, and three functions.

Finding #5: Callable Annotations It was much more challenging to identify functions that should be marked as `transaction_callable`. We ultimately added 23 annotations. Unlike `transaction_safe`, there was no compiler support for this task. Furthermore, this analysis is likely to change over time. For example, if a relaxed transaction deterministically calls unsafe function `foo()` before calling `bar()`, then it appears unnecessary to annotate `bar()`: the calling transaction will be serial and irrevocable, and thus instrumentation of `bar()` serves no purpose. However, should `foo()` one day become safe, then the failure to annotate `bar()` could become a performance bug. In general, tools should provide better support for determining which functions should be annotated as callable.

Many built-in functions were already marked safe by GCC. However the following functions are currently not `safe`, and can only be used in serialized relaxed transactions:

- 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: `realloc`
- Variable arguments: `__builtin_va_end`, `__builtin_va_start`

In addition, the memcached functions for reference counting (`refcount_decr`, `refcount_incr`) required relaxed transactions, as they use GCC's built-in atomic increment/decrement intrinsics.

Finding #6: The Future of Safe Library Functions Clearly, some of these operations can be handled as special cases. Most notably, the use of atomic increment/decrement operations can be transformed into simple add and subtract operations when executed from within a transaction. Similarly, an assertion that leads to program termination could be a special-case. However, the specification does not handle other critical cases well. Consider the string functions and variable arguments: These functions are not safe because they use custom assembly code. However, they *could* be implemented in pure C code, at the expense of performance of non-transactional code.

To handle this, we believe the specification requires an extension: It is not enough to write code once, and expect the compiler to generate optimal transactional and nontransactional variants. Instead, it must be possible to create two implementations of a function with the same name, where one version is exclusively for non-transactional code, and the other is exclusively for use from a transactional context. In this manner, it would be possible for a single function (e.g., `snprintf`) to expose two symbols: one nontransactional symbol, corresponding to an implementation that uses inline assembly, and another transactional symbol, corresponding to an implementation that is `transaction_safe`.

Finding #7: Commit Handlers In the spirit of preserving the maintainability of the memcached code, we opted not to perform control flow restructuring. In the case of the call to `pthread_cond_signal`, this prevented the use of an atomic transaction. The problem is that this call takes place in a deeply nested call stack. It would be correct to delay this call until after the transaction commits, but doing so would require changes to several function signatures, or to implement thread-local flags for leaking the need to signal a condition out of a transaction. A clean solution, in the form of an "onCommit" handler, would be ideal in this case.

Finding #8: Timing of Serialization Given that assertions never evaluate to false in a typical execution, we removed them from the code. This led to two more functions becoming *safe*, and one transaction becoming atomic. However, it was often the case that removing assertions merely delayed the point in a transaction's execution where it would become irrevocable. Unfortunately, GCC does not yet provide profiling feedback regarding the frequency of commits

[1] Strictly speaking, the use of `volatile` for this purpose does not have well-defined semantics. If memcached were a C++ program, these `volatile` variables could be replaced with C++ `atomics`. Of course, neither `atomic` nor `volatile` variables can be accessed from atomic transactions.

and aborts on a coarse or per-transaction basis, and also does not indicate how often transactions become serial and irrevocable. In particular, if a relaxed transaction will always become serial, but the unsafe call does not occur until late in the execution, then attempting to allow concurrency during the prefix of the transaction may simply increase latency. Tools for profiling these behaviors would be valuable.

Finding #9: Predicting Nesting Overheads Reference counts are another source of unnecessary serialization. We proceeded to replace all reference counts with transactions. However, in doing so we determined that some reference count updates were performed outside of transactions, necessitating that `refcount_incr()` and `refcount_decr()` be implemented as transactions themselves. In addition, at this point we determined that some transaction expressions had become nested within transactions. From a performance perspective, this raises two questions. First, do libraries provide optimized implementations for simple transaction expressions, or do these statements contain a full thread checkpoint and complex commit protocols? Second, what is the overhead of nesting? Without a model of the cost of nesting, programmers will be tempted to sacrifice maintainability and duplicate code in order to reduce nesting overheads.

A more broad concern related to this finding is that our transformation of all reference counts into transactions does not generalize. In large programs, making all reference counts (or other atomic read-modify-write operations) transactional may not be possible. The only recourse for the programmer is to use a relaxed transaction, and accept that under the current specification, this will result in serialization of all transactions.

Finding #10: Lock Orders Can Change There were three instances in which the first operation of a `cache_lock`-protected critical section was to acquire the `slabs_lock`. When `cache_lock` was replaced with a transaction, the transaction immediately would become serial on account of the call to `pthread_mutex_lock`. Since neither lock is released until the end of the critical section, in these cases it is correct to change the lock order, i.e., acquire the `slabs_lock` and then begin the transaction. Doing this allowed one relaxed transaction to become atomic. In another case, it led to the only unsafe operation in a relaxed transaction being on a rare code path, which should allow for good scalability in the common case. In one case, this transformation complicated an early return from the critical section, again suggesting that onCommit handlers would be valuable.

5. Replacing Multiple Locks

Individually replacing the `slabs_lock` or `stats_lock` did not reveal any new insights relative to what was already gleaned through our work with the `cache_lock`. However, removing the three locks together did lead to new experiences and additional suggestions.

We began with the `stats_lock`, which revealed a surprising aspect of memcached code. On a few occasions, a function would acquire a lock multiple times within a short region of code. The pattern appears in Algorithm 1.

This pattern clearly does not match with a mental model in which lock acquisitions are expensive and incrementing a counter is not. Furthermore, when this entire code region appears within an atomic transaction, transforming each critical section into a distinct nested transaction does not make sense. This, in turn, implies that under some circumstances, using TM will encourage programmers to enlarge critical sections, which is somewhat contrary to the advice that transactions should be kept small.

Algorithm 1: Rapid re-locking in memcached.

```
LOCK(stats_lock)
INCREMENT(counter1)
UNLOCK(stats_lock)
if ⟨unlikely_condition⟩ then
    LOCK(stats_lock)
    INCREMENT(counter2)
    UNLOCK(stats_lock)
```

We also found that in some places within memcached, multiple arms of a conditional will re-read the same volatile variable. This pattern appears in Algorithm 2.

Algorithm 2: Re-reading a volatile within a conditional.

```
if volatile_var = 1 then
    // block 1
else if volatile_var = 2 then
    // block 2
else
    // block 3
```

As before, `volatile` is an approximation of the `atomic` feature of C++. That being the case, it is virtually impossible for an outsider to determine whether the lack of atomicity among repeated reads within the function is intentional, or represents a possible bug. Consequently, we could not determine whether this code *required* a relaxed transaction. Another question that emerges from this observation is that if the re-reads were intentional, then since we made all volatile variable writes in memcached occur inside of lock-based critical sections, our replacement of locks with transactions would not allow behaviors expected by this code. This effect is identical regardless of whether we used relaxed transactions to access the volatiles, or converted the volatiles into non-volatile variables that were only accessed with atomic transactions.

Finding #11: Granularity of Atomicity Must Be Documented To summarize, these two experiences suggest that it is not easy to discern the intended granularity of atomicity in legacy code, and that transactionalizing code risks altering behavior, even when the transformation preserves race freedom. In our examples above, we saw that a lock may be acquired and released multiple times, and that a shared memory location might be re-checked on each arm of an if-else conditional. In memcached, these idioms do not seem to be related to a subtle communication technique, and thus replacing each with a coarse transaction appears correct. Even so, it is not clear that the transformation is benign with respect to performance. Consider the case where `volatile_var` changes frequently, and the code within "block 1" is unlikely to conflict on data, but takes a long time to run. In such a case, changes to `volatile_var` will cause "block 1" to abort and roll back, wasting significant work. Our assumption that repeated reads of a variable in the conditions of a nested `if` statement required atomicity could result in a higher abort rate.

Finding #12: The Return of Abstract Nesting Our `mutrace` analysis of memcached showed that both the `cache_lock` and the `stats_lock` are highly contended. A crucial concern, though, is that the `stats_lock` protects a small amount of global data, such that converting to use transactions is likely to generate many conflicts. Furthermore, `stats_lock` operations are often nested within `cache_lock` operations. Thus at the point where we had

maximally employed atomic transactions, and replaced all remaining critical sections with relaxed transactions, it appeared that transactions might frequently abort due to global statistics updates performed in the middle of a large transaction. The nature of these transactions is compatible with abstract nested transactions (ANTs) [5]. Such features, whether employed automatically or manually, may be necessary to ensure scalability.

Finding #13: Cross Platform Concerns One final issue that warrants discussion is that committing to TM in the near term is likely to produce inscrutable and macro-laden code. Currently, memcached compiles on many operating systems and platforms, to include those that lack compiler support for atomic operations (e.g., `fetch-and-increment`), and macros hide platform-specific implementation differences. The need for cross-platform support can be an obstacle to progress: it is even unlikely that memcached `volatiles` will be replaced with `atomic` data types any time soon, and if such a conversion occurs, it will likely be handled via even more macros. Similarly, preserving multi-platform support while performing the modifications discussed in this paper to leverage TM would necessitate duplicating the codes that we restructured (e.g., those relating to lock order) and hiding new keywords behind macros. The resulting code would be significantly less maintainable.

6. Evaluation

We believe that it is too early to draw definitive conclusions about the performance of STM on memcached. First, the GCC implementation has high latency due to its implementation of STM as a shared library. Second, without additional modifications to GCC and the specification (to include safe versions of variable argument functions and onCommit handlers), the memcached code requires serialized relaxed transactions in too many cases. Likewise, we suspect that ANTs will be needed for the highly contended but nested `stats_lock` transactions; without such support, the STM algorithm used by GCC could livelock.

Nonetheless, some preliminary results are instructive, as they help to understand whether the performance model implied by transactions is acceptable. It is also useful to observe whether first-generation TM support suffices to deliver on the promise of better scalability than coarse locks. To these ends, we present results on a dual-chip Intel Xeon 5650 system with 12 GB of RAM. The Xeon 5650 presents 6 cores/12 threads, giving our system a total of 12 cores/24 hardware threads. The underlying software stack included Ubuntu Linux 12.04, kernel version 3.2.0-27, and GCC 4.7.1 (–O3 optimizations). All code was compiled for 64-bit execution, and results are the average of 10 trials.

We generated a workload for memcached using memslap v1.0, downloaded as part of the Ubuntu libmemcached-0.31_1 source package. To ensure that network overheads were not hiding the higher latency of transactions, we ran the memcached server and memslap on the same machine. In our experiments, 12 threads were dedicated to memslap, so that there was a constant amount of work. We varied the number of memcached threads between 1 and 12. All memslap threads were pinned to one processor, and all memcached threads to the other. The benchmark ran until all outstanding requests were satisfied, and thus the overall time to execute the benchmark indicates the effectiveness of our transactionalization. We ran memslap with parameters `--concurrency=12 --execute-number=416667 -s localhost:11211 --binary`.

Figure 1 presents the results of our experiment. There are a total of 7 bars. Baseline is a version of memcached 1.4.13 in which we modified condition variables to decouple condition synchronization from the locks used to protect data. As stated earlier, this modification did not change performance relative to the public memcached code. The remaining 6 bars correspond to several different degrees of transactionalization of memcached:

- CacheLockNaive: the `cache_lock` was replaced with transactions, but no optimizations from the paper were employed.
- CacheLock: again, only the `cache_lock` was replaced with transactions.
- SlabsLock: only the `slabs_lock` was replaced with transactions.
- StatsLock: only the `stats_lock` was replaced with transactions.
- Cache+Stats: `cache_lock` and `stats_lock` were replaced with transactions.
- Cache+Stats+Slabs: `cache_lock`, `stats_lock`, and `slabs_lock` were replaced with transactions.

In CacheLockNaive, the `transaction_callable` annotation was not used. In all other versions of memcached, we employed all techniques discussed in this paper in order to maximize the use of atomic transactions.

The dominant result is clearly a negative one: every attempt to transactionalize memcached caused a slowdown, and slowdowns increased as the threading level increased. This suggests that at the GCC level, or perhaps even at the specification level, it may be desirable to provide some interface to a contention manager. Whether contention management at transaction boundaries [10] is sufficient (which could be supported by an `onAbort()` handler), or per-access decisions [17] are required, is not of consequence to such a feature, and can remain an implementation issue.

Analyzing each effort independently, we see several trends. The slabs lock was not highly contended, and was transactionalized only collaterally. Thus an implementation that only transactionalized the slabs lock did not differ greatly from the baseline, and performance is reasonably close until higher thread counts. On the other hand, the stats lock is highly contended, but this contention is due to true conflicts over data. Transactionalizing only the stats lock led to higher latency within these critical sections, but no increased concurrency.

The next most apparent result is that performance results do not compose. When we transactionalized just the `cache_lock` or just the `stats_lock`, we did not observe nearly the slowdown that occurred when both were transactionalized. As discussed earlier, nesting a `stats_lock` transaction within a `cache_lock` transaction increases contention, since `stats_lock` transactions are small, frequent, and likely to conflict. By nesting these transactions, the larger parent transaction inherited the conflicts of the child, and thus became more likely to abort. This result suggests that either ANTs or a lazy STM algorithm should be employed. To a lesser degree, this result confirms that the specification should expose some form of contention management advice from the programmer.

Perhaps the most disappointing finding, however, comes from comparing CacheLockNaive to CacheLock. The majority of our effort was in minimizing the number of relaxed transactions in the source code, based on an expectation that atomic transactions would scale better. Instead, our results show that this effort always produced a longer running time. Without detailed profiling tools, we cannot tell whether this was due to new conflicts (e.g., on reference count updates, which could be resolved via ANTs), more instrumentation overhead, or something intrinsic to our code transformations. In any event, the result should serve as a warning: while STM may be a powerful tool for designing new applications, programmers should be cautious when using it to replace carefully-designed lock-based critical sections. There is no guarantee of improved performance.

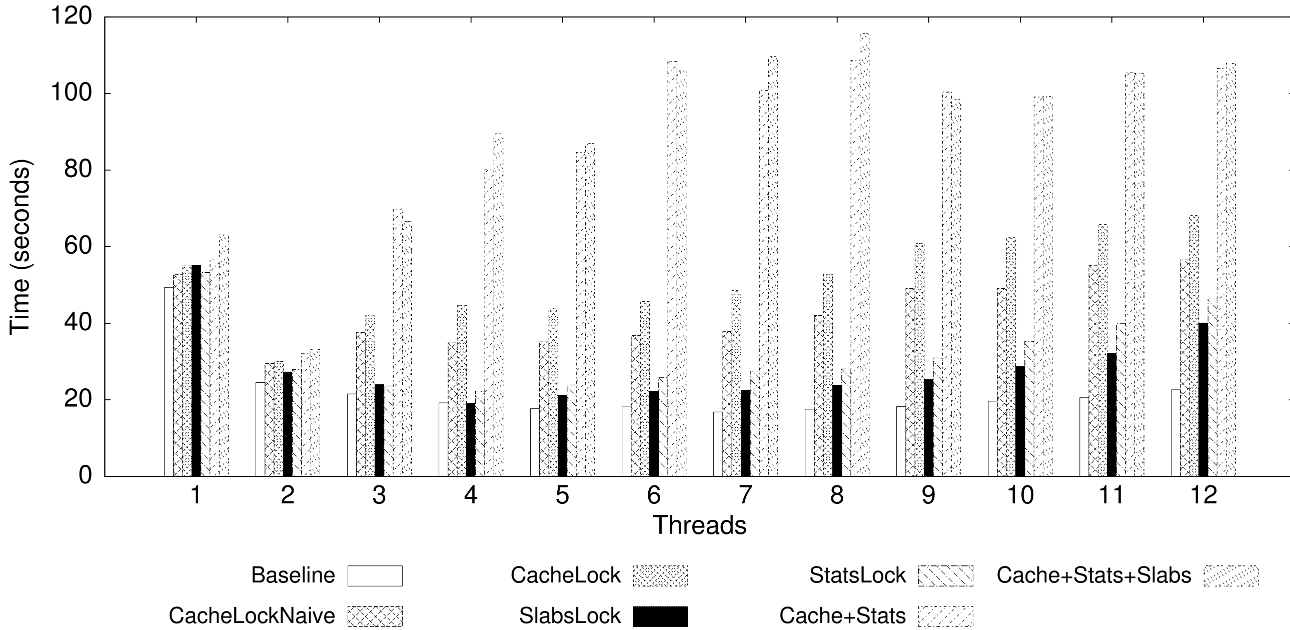

Figure 1: Running Time of memcached at Different Levels of Transactionalization (lower is better)

7. Relaxed or Atomic?

The overwhelming majority of our time and effort was spent on performance tuning. However, the nature of the C++ specification is such that most of this tuning can be done before executing code.

While there are noteworthy semantic differences between relaxed and atomic transactions, for the purposes of this work the key difference relates to the performance model exposed to the programmer. When a transaction is atomic, the programmer can be sure that there is no artificial obstacle to concurrency, whereas a relaxed transaction might contain some operation (to include a seemingly benign call to a variable argument function) that forces serialization.

In our experience, the annotation of functions as safe was tedious, but not particularly thought-intensive; compiler errors guide this process well. For memcached, the exact same performance and behavior could be achieved in a specification that lacked both atomic transactions and the `tm_safe` annotation, but achieving the same effect with relaxed transactions and the `tm_callable` annotation would have been substantially harder. The compiler would not have guided us, and we never would have known that so many library calls were unsafe.

Our study entailed only one application, written in C. The application did not use any of the exception-related features of the Draft C++ TM Specification, which apply only to atomic transactions. While it would be premature to draw any conclusions about the need for both atomic and relaxed transactions we nonetheless feel that both play an important role. In our view, atomic transactions carry a performance model that is statically checked, and about which the programmer can reason; even when performance does not match the model, the fact that atomic transactions were used guides the programmer's analysis: some potential problems, such as serialization, simply are not possible.

On the other hand, relaxed transactions provide a safety guarantee for operations that cannot be atomic, at the expense of a performance model. Certainly the best case performance for relaxed transactions should match atomic transactions, but without the compiler's guidance, we would not have been able to achieve any confidence that our use of transactions would avoid serialization. However, note that relaxed transactions are not simply a tool for transactionalizing code before the tools are mature, or a tool for getting a default transactionalization up and running. For I/O with transactional data, calls to libraries that cannot be instrumented, and interactions with other nontransactional threads, relaxed transactions preserve correctness while remaining relatively easy to understand.

We expect that over time, legacy programs will transition from containing mostly relaxed transactions to containing mostly atomic transactions. During this transition, the presence of relaxed transactions will be a useful guide. In particular, knowing which transactions remain relaxed, and why, should guide future work to both (a) fix some library functions and (b) rewrite portions of code to avoid unsafe calls. There will always remain transactions that must be relaxed, due to necessary but unsafe operations (e.g., I/O of shared data). Less certain is whether programs will always contain relaxed transactions that should scale. Consider, for example, a transaction that performs logging in the event of a failure. While this can be handled in a scalable fashion today through the use of relaxed transactions, we believe that some transaction-safe failure logging facility is the more desirable approach: it would present a cleaner performance model.

8. Conclusions

In this paper, we presented our experiences transactionalizing the memcached in-memory web cache using GCC. Our focus was analyzing the Draft C++ TM Specification, not on evaluating a particular TM mechanism or tool.

Throughout the paper we identified areas where the specification could be changed to improve programmability, transformations and analyses that programmers will need to perform, opportunities for tool creators, and maintenance and performance challenges that will arise during the piecemeal transactionalization of multi-platform programs. We hope that this study will assist standardiza-

tion efforts by providing further insight into the challenges that will be faced when transactionalizing legacy code, and that our source code will provide TM algorithm designers with a new workload for evaluating implementations.

Acknowledgments

We thank our anonymous reviewers, whose detailed suggestions greatly improved this paper.

References

[1] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft Specification of Transactional Language Constructs for C++, Feb. 2012. Version 1.1, `http://justingottschlich.com/tm-specification-for-c-v-1-1/`.

[2] M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson. Lee-TM: A Non-trivial Benchmark for Transactional Memory. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, June 2008.

[3] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[4] K. Fraser. *Practical Lock-Freedom*. PhD thesis, King's College, University of Cambridge, Sept. 2003.

[5] T. Harris and S. Stipic. Abstract Nested Transactions. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[6] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[7] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A Simple Exploration Tool for Orthogonal TM Characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Atlanta, GA, Dec. 2010.

[8] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. In *Proceedings of the International Symposium on Memory Management*, Ottawa, ON, Canada, June 2006.

[9] G. Kestor, S. Stipic, O. Unsal, A. Cristal, and M. Valero. RMS-TM: A Transactional Memory Benchmark for Recognition, Mining and Synthesis Applications. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[10] Y. Liu and M. Spear. Toxic Transactions. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[11] 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 EuroSys2010 Conference*, Paris, France, Apr. 2010.

[12] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[13] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Seattle, WA, Sept. 2008.

[14] V. Pankratius and A.-R. Adl-Tabatabai. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, June 2011.

[15] M. Pohlack and S. Diestelhorst. From Lightweight Hardware Transactional Memory to Lightweight Lock Elision. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[16] C. Rossbach, O. Hofmann, and E. Witchel. Is Transactional Programming Really Easier? In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[17] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[18] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, ON, Canada, Aug. 1995.

[19] M. Spear, M. M. Michael, and M. L. Scott. Inevitability Mechanisms for Software Transactional Memory. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Feb. 2008.

[20] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[21] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.