Transactionalizing Legacy Code: an Experience Report Using GCC and Memcached [*]

Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear
Lehigh University
{wer210, 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

Keywords Transactional Memory; memcached; GCC; C++

1. Introduction

For over a decade, Transactional Memory (TM) [13, 31] 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. [28] and Pankratius and Adl-Tabatabai [24] 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 [23], EigenBench [15], Atomic Quake [37], Lee-TM [2], SynQuake [20], and RMS-TM [16].

While these benchmarks and studies collectively provide a good platform for evaluating TM implementations, they treat the STM interface as a constant. Our effort in this paper is complementary: we are interested in evaluating the proposed C++ TM interface. Our approach is to transactionalize a real-world application using the Draft C++ TM Specification, with an eye towards identifying (a) what common programming patterns and idioms are not well supported, (b) what features are cumbersome or difficult to use, and (c) what performance implications the specification carries.

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 used the developmental GCC version 4.9.0, 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 enabled us to investigate the behavior of different TM algorithms and low-level design decisions.

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 and condition variables are intertwined; there is a statically defined order in which locks must be acquired; there is a reference counting mechanism that employs in-line assembly and volatile variables (i.e.,

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

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.
Copyright © 2014 ACM 978-1-4503-2305-5/14/03...$15.00.
http://dx.doi.org/10.1145/2541940.2541960

C++11 `atomics`); and it uses high-performance external libraries, such as a worklist implemented via libevent [21]. Furthermore, memcached uses some of these primitives in unconventional ways. For example, while there is a strict locking order (item, cache, slab, and then stats locks), this order is sometimes violated by using `trylock` to acquire an item lock while holding the cache lock. Similarly, in some cases a pthread lock is used as a spinlock, by acquiring it using a trylock inside of a while loop.

In our study, we used memcached version 1.4.15, which contains recent scalability enhancements. While our work is similar to a recent effort by Pohlack and Diestelhorst [26], it differs in that we are using memcached to analyze the Draft C++ TM Specification, rather than to assess the utility of a particular hardware 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 then chronicles the steps we took to transactionalize memcached, and reports the performance impact of our efforts. In Section 4, we explore changes to GCC that would affect our results. Section 5 presents recommendations for designers and implementors of TM libraries, and Section 6 makes recommendations for programmers intending to use the Draft C++ TM Specification. Section 7 concludes.

2. The Draft C++ TM Specification

The Draft C++ TM Specification [1] integrates transactional memory support into C++ through the addition of several new keywords and annotations. The extensions 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 do not carry the same restrictions as atomic transactions; they are allowed to perform I/O and other unsafe operations. This is achieved by enabling a relaxed transaction to become irrevocable [34, 36], or perhaps to run in isolation, starting at the point where it attempts an instruction 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 serial and irrevocable, they cannot run concurrently). However, in the absence of such unsafe code, the two types of transactions are indistinguishable; both 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 serial and irrevocable.

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. As an example, consider the following code segment:

```
__transaction_relaxed {
  ...
  if (verbose)
    fprintf(stderr, message);
  ...
}
```

When `verbose` is false, this transaction need not become irrevocable. When it is true, the inability of the TM system to undo writes to stderr necessitates that it become irrevocable before calling fprintf. Regardless of the value of verbose, it must be relaxed: the compiler cannot guarantee that it will never require irrevocability.

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.

Extensions For the purposes of our work, it is useful to consider two extensions to the Draft C++ TM Specification, both of which are supported in GCC. The first is the transaction_pure annotation, which allows programmers to indicate that certain functions are transaction safe without their memory accesses being instrumented. While this is intended as a performance optimization, its implementation is not checked: for example, one could annotate printf as being transaction_pure, and then the compiler would allow calls to printf from within a transaction.

The second extension allows registration of functions to run after a transaction commits or aborts. Functions registered as onCommit handlers run after a transaction commits. Functions registered via onAbort handlers run after an aborted transaction had undone any memory effects, but before it retries. Both take a single untyped parameter.

3. Transactionalizing Memcached

The Draft C++ TM Specification appears to offer a simple route to transactionalizing legacy code: one need only replace all lock-based critical sections with relaxed transactions. Since the existing lock-based code does not require compiler support to undo effects, there is no need for __transaction_cancel. Consequently, it would seem that atomic transactions are unnecessary, and transaction_callable annotations optional.

There are two flaws in this line of thinking. First, many programs contain condition variables, which require the use of an associated mutex. Dealing with condition synchronization currently requires ad-hoc solutions, and we discuss our solution for memcached below. Secondly, relaxed transactions are prone to serialization when they encounter an unsafe operation. Currently, there are no tools for easily identifying serialization points in relaxed transactions. Thus we claim that programmers should think of relaxed and atomic transactions as differing in terms of the performance model they carry: relaxed transactions have no performance guarantees, but atomic transactions guarantee that serialization will be avoided wherever possible.

Given this performance model, programmers can replace relaxed transactions with atomic transactions to gain a static guarantee that those transactions will not cause unnecessary serialization. To guide this effort, the programmer can use error messages from incorrect uses of atomic transactions and transaction_safe attributes as a tool for identifying unsafe operations.

We eliminated all contended locks in memcached, without requiring relaxed transactions in the final code. We first identified the locks that should be replaced, and then modified any condition synchronization built atop those locks. Next, we applied the Draft C++ TM Specification to the fullest extent possible to replace locks with atomic or relaxed transactions. We then developed transaction-safe alternatives to unsafe standard libraries. Finally, we applied onCommit handlers to eliminate all remaining relaxed transactions, resulting in a program in which no transaction required serialization to complete.

3.1 Identifying Locks

Our first step was to identify those locks that indeed were worthy of removal. There are four categories of locks in memcached, which are acquired in the following order:

1. item locks: These locks protect individual elements in the hash table that serves as the central data structure in the application.
2. cache_lock: This lock prevents concurrent modifications to the structure of the hash table (i.e., resizing).
3. slabs_lock: This lock protects the slab allocator. Slabs are memory blocks that store sets of objects of the same maximum size.
4. 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, some remain as global variables.

We profiled the contention on locks by using mutrace [25]. This revealed that the cache_lock and stats_lock were the only locks that threads frequently failed to acquire on their first attempt. Thus it was necessary to replace at least these two locks with transactions. However, several more locks ultimately required replacing:

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 serialize on account of the call to pthread_mutex_lock. Since the two locks are also released together at the end of the critical section, in these cases it is correct to change the lock order,

```
1    void func1a() {
2      __transaction_atomic {
3        if (tm_trylock(i.lock))
4          use_item(i);
5          tm_unlock(i.lock);
6        else
7          save_for_later(i);
8      }
9    }
10
11   void func2a() {
12     tm_lock(i.lock);
13     use_item(i);
14     tm_unlock(i.lock);
15   }
```

(a) `func2` privatizes `i`.

```
1    void func1b() {
2      __transaction_atomic {
3        // save_for_later isn't needed, since
4        // func2 no longer accesses i via
5        // privatization. Instead, this
6        // transaction and line 11 might conflict
7        use_item(i);
8      }
9    }
10
11   void func2b() {
12     __transaction_atomic {
13       use_item(i);
14     }
15   }
```

(b) `func2` does not privatize `i`.

Figure 1: Example of privatization that may arise under less aggressive approaches to transactionalization.

i.e., acquire the `slabs_lock` and then begin a transaction in place of acquiring the `cache_lock`. However, in other cases the `slabs_lock` was only acquired well after the `cache_lock`, and the only way to prevent `cache_lock` transactions from becoming serial and irrevocable was to also replace the `slabs_lock` with transactions.

Another challenge related to per-thread statistics. Over the past few years, many of the statistics counters in memcached have been transformed from global counts to per-thread counters, which are protected by per-thread locks. Unfortunately, *any* operation on a mutex lock is unsafe to perform in an atomic transaction, and thus, we had to replace these locks with transactions. This highlights a flaw with relaxed transactions: when an unsafe operation is performed in a context where conflicts are exceedingly rare, it still necessitates the serialization of all transactions. One solution to this problem would be to make lock operations transaction-safe. Barring such an option, we were forced to replace uncontended per-thread locks with transactions.

Indeed, "transaction-safe" locks were required when replacing the lock governing slab re-balancing. In memcached, a "slab rebalance" lock is used by the cache and slab maintenance threads to prevent concurrent maintenance of the cache and slab data structures. While holding other locks, these threads might use `trylock` calls to determine whether a concurrent maintenance operation is in-flight. To remedy this, we replaced the rebalance lock with a boolean that was modified via transactions. This allowed concurrent transactions to check the state of the lock. While the lock was primarily acquired via a spin loop and `trylock`, in one case it was acquired via a blocking call. Lacking any better alternative, we followed any failed blocking acquire with a call to `pthread_yield`.

Finally, and most unfortunately, we had to replace the item locks, even though they were never contended. While the locking order in memcached is item, cache, slabs, and then stats locks, there are cases in which a maintenance thread attempts to lock an item while holding other locks that come later in the locking order. In these cases, either a spin loop wraps a call to `trylock`, or else a `trylock` is used and failure to acquire results in a handler being registered so that missed items can be processed at a later time (see Line 7 of Figure 1a). While all `item_locks` are acquired using spin loops and `trylock`, we were still faced with an uncomfortable decision: as with the rebalance lock, we could make the lock acquire and release into mini-transactions on a boolean variable, or we could replace every item lock critical section with a transaction. The former choice immediately led to explicit *privatization* [22, 33]: some data protected by item locks would be accessed within transactions, and also outside of transactions (but with the item lock held). We ultimately chose both routes, and developed two branches of the code, one with privatization and the other without.

An illustration of the difference is provided in Figure 1. In branch 'b', where item lock critical sections are replaced with transactions, func1 and func2 may run concurrently and cause conflicts, but `i` is only accessed from within transactions. In branch 'a', where item locks were acquired and released with transactions, func1a is able to inspect the lock, and to use `i` within a transaction as long when the lock is not held, but `i` itself might be accessed outside of a transaction in func2a. Furthermore, the small transaction to acquire the lock in func2a will implicitly take priority over the larger transaction that reads the item's lock in func1a. Note that while branch 'b' is more aggressive with respect to replacing locks with transactions, both branches are correct, since the default TM algorithm in GCC is privatization safe, and this level of safety is a requirement of the Draft C++ TM Specification. Note too that neither option is clearly simpler: branch 'a' required fewer net lines of code to change, but branch 'b' enabled the removal of several corner cases (e.g., the `save_for_later` code path).

```
/* initially mx_can_run = true */

void worker() {
  lock(L);
  do_work();
  if (mx_needed())
    if (!mx_running)
      mx_running = true;
      cond_signal(C); //replace with sem_post(S);
  do_cleanup();
  unlock(L);
}

void halt_maintainer() {
  lock(L);
  mx_can_run = false;
  cond_signal(C);      //replace with sem_post(S);
  unlock(L);
}

void maintainer() {
  while(mx_can_run);
    lock(L);
    do_maintenance();
    unlock(L);
    lock(L);
    mx_running = false;
    cond_wait(L, C);  //replace with unlock(L);
    unlock(L);        //replace with sem_wait(S);
}
```

Figure 2: Comments depict a transformation for removing condition variables used to wake maintenance threads.

3.2 Refactoring Condition Synchronization

The `slabs_lock` and `cache_lock` are used by memcached both to protect critical sections and as the lock object for condition synchronization using `pthread_cond_t` variables. The current Draft C++ TM Specification does not support condition variables, and thus we were required to manually transform all condition synchronization.

A simplifying factor is that condition synchronization in memcached follows two simple patterns. First, there are condition variables for notifying threads when work arrives on a network connection. These are not associated with contended locks, and we did not consider them. The second pattern is for coordinating data structure maintenance. In this pattern, many worker threads can attempt to wake a maintenance thread, which will then modify a data structure. A simplified version of this pattern is depicted in Figure 2. This pattern appears twice, for re-balancing the hash table (via `cache_lock`) and maintaining slabs (via `slabs_lock`).

One flag exists for thread shutdown, and another for determining if the maintainer is currently active. Every attempt to wait on a condition is immediately followed by a lock release, a back edge in the control flow graph, and a lock re-acquire. Furthermore, the code already ensures that there are no spurious wake-ups. Consequently, one can replace the condition variables with semaphores. The changes are trivial, and appear as comments in Figure 2. Rather than call `cond_wait` within a lock-based critical section, maintenance threads call `sem_wait` immediately after completing the critical section. At this stage of the transactionalization, worker threads called `sem_post` from within a lock-based critical section. In stage 3, these critical sections became relaxed transactions, and in stage 5, these calls were moved to `onCommit` handlers within atomic transactions.

```
  lock(stats_lock);
  increment(counter1);
  unlock(stats_lock);
  if (unlikely_condition) {
    lock(stats_lock);
    increment(counter2);
    unlock(stats_lock);
  }
```

Figure 3: Rapid re-locking in memcached.

This transformation resulted in changes to 10 lines of code, not counting comments, and had no impact on performance. While the transformation suffices for memcached, it may still be beneficial for TM to explicitly support condition synchronization (possibilities include conditional critical regions/retry[3, 12], punctuated transactions [32], communicators [19], and transactional condvars [8]). In particular, when `pthread_cond_wait` is not the last instruction in a critical section, and is in a deeper lexical scope, compiler support will be needed to transform the two “halves” of the critical section into separate transactions.

3.3 Maximally Applying the Specification

Having identified the locks requiring replacement, and having developed alternative condition synchronization mechanisms for those locks also associated with condition variables, we were at last able to begin using transactions. We replaced all targeted critical sections with relaxed transactions, maximally applied the callable attribute, and then systematically replaced unsafe operations with safe operations, so that transactions could be marked atomic. Recall that in the absence of transaction cancellation, it would have been correct to leave transactions relaxed after making their operations safe, but that the absence of relaxed transactions guarantees the absence of mandatory serialization.

Replacing Locks, Adding Annotations We began by developing two branches, corresponding to the two approaches to item locks discussed above, in which the `cache_lock`, `slabs_lock`, and `stats_lock` were also transactionalized. This resulted in 51 relaxed transactions in each branch. We then traced all function calls from within relaxed transactions, and marked all functions for which we had the source as callable. This led to 38 annotations in the privatizing itemlocks (IP) branch, and 49 annotations in the transactional itemlocks (IT) branch.

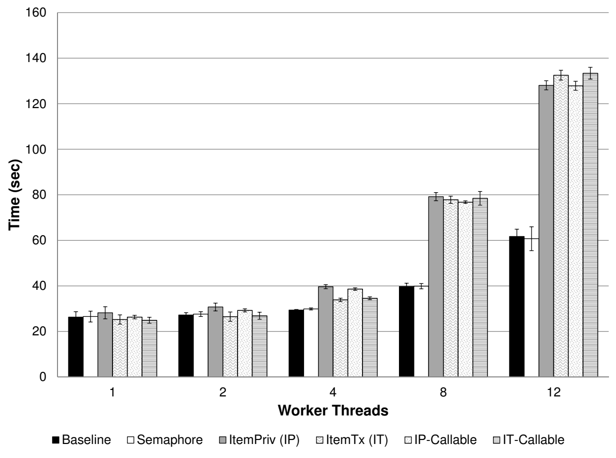

Figure 4: Performance of baseline transactional memcached.

Note that on a few occasions, a function would acquire a lock multiple times within a short region of code (see Figure 3). This pattern does not match with a mental model in which lock acquisitions are expensive. Furthermore, when this entire code region appears within an atomic transaction, transforming each critical section into a nested transaction seems unnecessary. This, in turn, implies that under some circumstances, using TM will encourage programmers to enlarge critical sections.

Figure 4 presents the performance of memcached with semaphores in place of condition variables, and then for each branch with (a) relaxed transactions but no callable annotations, and (b) relaxed transactions and callable annotations. Experiments were performed 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 13.04, kernel version 3.8.0-21, and an experimental GCC version 4.9.0. All code was compiled for 64-bit execution, and results are the average of 5 trials. Error bars depict a range of +/- one standard deviation.

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. We ran memslap with parameters `--concurrency=x --execute-number=625000 --binary`. We varied the memslap concurrency parameter (x)from 1 to 12 and matched memcached runs with the same number of worker threads plus an additional two maintenance threads. Note that perfect scaling corresponds to an execution time that remains constant at higher thread counts, since each thread executes 625K operations.

In Figure 4, we see that the switch from condition variables to semaphores is negligible, and that privatization (IP) appears to scale better than using transactions in place of item lock critical sections (IT). However, there is no evidence to suggest that the callable attribute (IP-Callable / IT-Callable) improves performance.

Branch	Trans-actions	In-Flight Switch	Start Serial	Abort Serial
ItemPriv (IP)	11201538	625K (5.6%)	625K (5.6%)	10
ItemTx (IT)	3462735	625K (18.0%)	1.25M (36.1%)	0
IP-Callable	10511717	625K (5.9%)	625K (5.9%)	10
IT-Callable	3467927	625K (18.0%)	1.25M (36.0%)	0

Table 1: Frequency and cause of serialized transactions for a 4-thread execution.

Table 1 presents the serialization rates for each branch. Naturally, there are fewer total transactions when transactions replace item locks, since each item lock acquire and release is a separate transaction in the IP branch. However, IP and IT have practically the same number of relaxed transactions that encounter unsafe code on a branch, and must become serial (In-Flight Switch), and that encounter unsafe code on every code path, and must begin in serial mode (Start Serial). A trivial number of transactions abort 100 times in a row, and serialize for the sake of progress.

Handling Volatiles and Reference Counts Many critical sections in memcached access volatile variables as part of periodic maintenance operations or condition synchronization. Strictly speaking, the semantics of such accesses are not defined, though in practice one can expect these volatile variable accesses to be an approximation of C++11's `atomic` types. Accesses to `volatile` and `atomic` variables are both considered unsafe, can not be performed in an atomic transaction, and force transactions to serialize. Therefore, our only hope of achieving scalability was to replace all volatile variable accesses with transactional accesses to non-volatile variables.

This transformation was straightforward. We renamed volatile variables, traced all compilation bugs, and resolved all errors by adding transactions that accessed the new non-volatile variables. In all, we only changed three variables, and the availability of transaction expressions meant that the total lines-of-code count did not change.

However, this transformation raises many questions. First, this transformation was only correct because we manually inspected the code to ensure that concurrent critical sections never interact via volatile variables. Second, 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. Third, GCC currently does not optimize single-location transactions, and thus this change could have a significant impact on performance. Finally, the specification must guarantee that the semantics of transactions are 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

```
    if (volatile_var == 1)
        block 1;
    else if (volatile_var == 2)
        block 2;
    else
        block 3;
```

Figure 5: Re-reading a volatile within a conditional.

value must have at least the same ordering guarantees as a store to a C++ atomic, where in both cases the access uses `std::memory_order_seq_cst`. This is already the case in the Draft C++ TM Specification.

Surprisingly, we found that in some places within memcached, nested `if` statements will re-read the same volatile variable (Figure 5). Based on the assumption that memcached is correct, we did not rewrite these codes.

Memcached uses atomic read-modify-write operations (e.g., `lock incr` on the x86) for reference counting. As with volatile variables, replacing these accesses with transactions was straightforward: we replaced every increment and decrement with a transaction, and replaced every read of these variables with a transaction expression. There was no change to the total lines of code. The main concern with such a transformation is that these assembly operations have memory fence semantics, which must then be guaranteed at transaction boundaries. There is also a question of optimality, since many critical sections increment the reference count, access a datum, and then decrement the reference count. With transactions, it might be possible to replace the modifications of the reference count with a simple read [7].

Performance for this "maximal" transactionalization is presented in Figure 6 and Table 2. For reference, the baseline memcached and "Callable" results are reproduced. At all thread counts, performance degrades, with significant slowdown at high thread counts. Even worse, serialization increased for both branches. In the IP branch, our transformation reduced the number of transactions that started in serial mode, but virtually all of these transactions still ultimately serialized. Such an outcome will consistently hurt performance, since code executes in an instrumented slow path up until the point where serialization is necessary, at which point GCC aborts the transaction and restarts it serially but with less instrumentation. By delaying the point of serialization, we actually hurt performance.

In the IT-Max branch, the transformation was less profitable, and the slowdown more pronounced. Two contributing factors are the increased number of transactions (each of which has higher latency than the code it replaced), and the dramatic increase in transactions that fall back to serial mode due to high abort rates. Since GCC's TM uses direct update [9, 29], the cost of aborts is expected to be high.

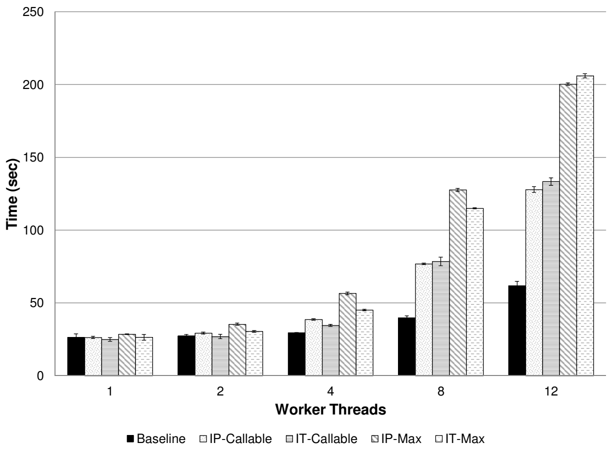

Figure 6: Performance of maximally transactionalized memcached.

Branch	Trans-actions	In-Flight Switch	Start Serial	Abort Serial
IP-Callable	10511717	625K (5.9%)	625K (5.9%)	10
IT-Callable	3467927	625K (18.0%)	1.25M (36.0%)	0
IP-Max	24085893	1162K (4.8%)	0	87K
IT-Max	6370739	559K (8.8%)	1.25M (19.6%)	66K

Table 2: Frequency and cause of serialized transactions for a 4-thread execution.

3.4 Making Libraries Safe

The most prominent remaining cause of serialization was calls to unsafe standard library functions. These calls fell into a few categories. First, there were operations on untyped memory: `memcmp`, `memcpy`, and `realloc`. Second, there were basic string functions: `strlen`, `strncmp`, `strchr`, `strncpy`, `isspace`, `strtol`, `strtoull`, `atoi`, and `snprintf`. Third, there were calls related to variable arguments: `vsnprintf`, `va_start`, and `va_end`. Lastly, there was a single call to `htons`.

Safety via Reimplementation Most of these functions are simple to transactionalize: we re-implemented the function and marked it transaction safe. This addressed calls to `memcmp`, `memcpy`, `strlen`, `strncmp`, `strncpy`, and `strchr`. Similarly, we re-implemented `realloc` in the naive way, by always allocating a new buffer and using `memcpy`.[1] Unfortunately, the Draft C++ TM Specification requires the transactional and nontransactional versions of a function to be generated from the same source. Thus to make our functions safe, we could not use custom assembly (e.g., vector instructions in memcpy), and thus we had to slow down the non-transactional code path by replacing calls to optimized standard library functions with calls to our naive implementations.

[1] We were able to optimize this slightly, since the initial size of the input is always known in memcached.

```
//wrap library function foo inside a
//pure function
[[transaction_pure]]
int pure_foo(char *in, char *out) {
  return foo(in, out);
}

...
//assume that max sizes of input and output
//strings are known otherwise, use malloc
//since alloca is not transaction-safe
int size = tm_strlen(shared_in_string);
char in[size1];
char out[size2];
//marshall data onto stack
for (int i = 0; i < size; ++i)
  in[i] = shared_in_string[i];
//invoke function with non-shared parameters
size = pure_foo(in, out);
//marshall data off of stack
for (int i = 0; i < size; ++i)
  shared_out_string[i] = out[i];
```

Figure 7: Example of marshaling shared memory onto the stack to invoke an unsafe library function `foo()`.

Safety via Marshaling To make the remaining functions safe, we relied on a novel but unsafe technique. GCC is aggressive about avoiding instrumentation, in particular by noticing when reads and writes are performed to the stack [27] and/or captured memory [6]. By combining this observation with the `transaction_pure` extension, we were able to implement a pattern in which data was marshaled from shared memory onto the stack, so that an unsafe library function could then be invoked using only thread-local parameters. As needed, the result would then be marshaled back into shared memory. A generic example of this approach appears in Figure 7.

Using this technique could be dangerous in buffered update STM algorithms [5], unless the programmer can be sure that the first marshaling operation does not buffer its writes. Although the GCC TM implementation does not use buffered update, we still verified that GCC does not instrument the writes to `in`, or the reads from `out`.

The `isspace`, `strtol`, `strtoull`, and `atoi` functions could all be made safe in this manner, by marshaling the input string onto the stack, calling a wrapped version of the function, and then using the scalar return value without any further marshaling. `htons` did not require any marshaling, since its input and return values are both integers. Similarly, `snprintf` required all its parameters to be marshaled onto the stack, and its output parameter to be marshaled back to shared memory.

Variable Arguments GCC does not yet support variable arguments within transaction-safe functions. Our solution was to manually clone and replace every variable-argument function with a unique version for every combination of parameters that appeared in the program. While this approach is tedious and does not generalize, we believe the effort is valid for a study such as this: there are no performance or correctness reasons why variable arguments will not eventually be transaction safe.

Branch	Trans-actions	In-Flight Switch	Start Serial	Abort Serial
IP-Callable	10511717	625K (5.9%)	625K (5.9%)	10
IT-Callable	3467927	625K (18.0%)	1.25M (36.0%)	0
IP-Max	24085893	1162K (4.8%)	0	87K
IT-Max	6370739	559K (8.8%)	1.25M (19.6%)	66K
IP-Lib	25658618	625K (2.4%)	0	15K
IT-Lib	8211858	0	625K (7.6%)	10K

Table 3: Frequency and cause of serialized transactions for a 4-thread execution.

Unfortunately, the techniques discussed in this subsection are not general. For example, while one might argue that even after standard libraries become transaction-safe, marshaling would still be useful for third-party libraries, there are many dangers. Chief among them are that a third-party library might at some point begin using transactions internally, resulting in erroneous behavior (especially with buffered update STM), and that marshaling requires, in all cases, that the assembly code be inspected to guarantee that on-stack buffers are being updated appropriately. Another concern is that one must predict the sizes of buffers. In memcached, maximum buffer sizes were easy to discern in all but one location, for which we used a generous 4KB buffer for the input, and 8KB for the output. Furthermore, when a transaction employs the marshaling technique multiple times within a single transaction, the illusion of atomicity may be violated. For example, `snprintf` relies on locale information when handling floating-point values. In a pathological case, the locale could be changed by an administrator between two of these marshaled calls by the same transaction. The output would clearly not appear to be *atomic*. Finally, the absence of side effects in a library is not always certain. A function's implementation might change, introducing static variables or logging that is invisible to the caller.

Figure 8 and Table 3 present the performance when standard library functions were made transaction safe. We see a notable improvement in performance, especially at higher thread counts, though still not as good as the earlier IP-Callable effort. Analyzing the frequency of serialization, we see a marked improvement in all areas: fewer transactions require serialization when they start, fewer become serialized during execution, and fewer abort at a high enough rate to require serialization to ensure progress.

3.5 Moving Code Out of Transactions

Only 6 unsafe functions remained in transactions: `event_get_version`, `assert`, `sem_post`, `fprintf`, `perror`, and `abort`. We began by handling `assert` and

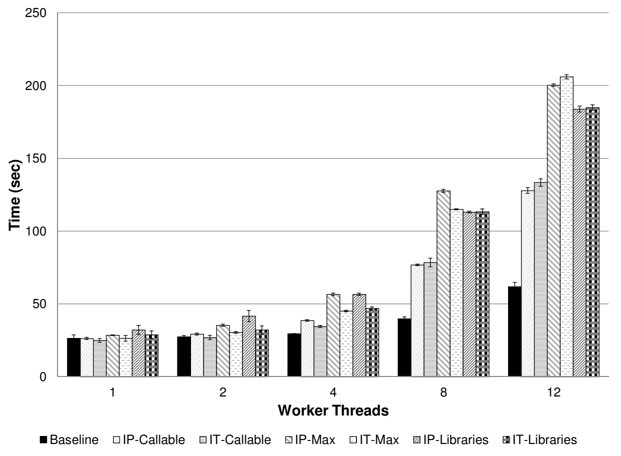

Figure 8: Performance with safe library functions.

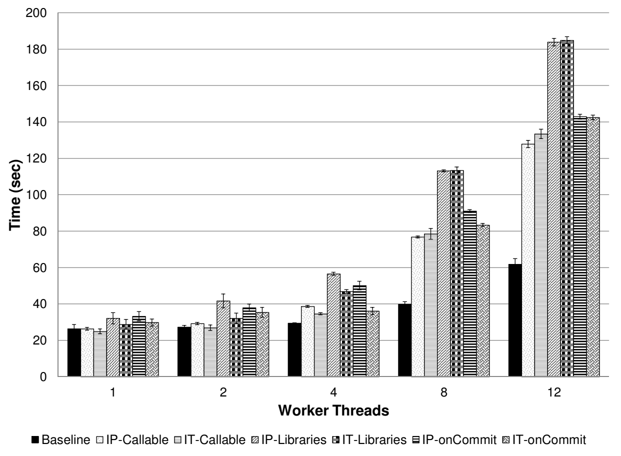

Figure 9: Performance with onCommit handlers.

Branch	Trans-actions	In-Flight Switch	Start Serial	Abort Serial
IP-Callable	10511717	625K (5.9%)	625K (5.9%)	10
IT-Callable	3467927	625K (18.0%)	1.25M (36.0%)	0
IP-Lib	25658618	625K (2.4%)	0	15K
IT-Lib	8211858	0	625K (7.6%)	10K
IP-onCommit	40305505	0	0	48K
IT-onCommit	8130896	0	0	8K

Table 4: Frequency and cause of serialized transactions for a 4-thread execution.

`abort`, which can immediately terminate a program, without calling any `atexit` functions. If the underlying TM is opaque [11], then at the point where the assertion evaluates to false or the `abort` is encountered, there exists an equivalent lock-based execution in which termination would be justified. Furthermore, since `atexit` functions are not called, other threads should not be able to observe the intermediate state of the transaction calling `assert` or `abort`. Consequently, it is safe to simply terminate the program at these points. To achieve this effect, we wrapped the assert/abort code and accompanying I/O in a pure function. Note that the I/O only involved string literals. Note, too, that while this change allowed many relaxed transactions to be marked as atomic, there was no impact on the frequency of serialization, since the relevant code never runs in a correct execution of memcached.

To handle `event_get_version`, we assumed that the version of `libevent` would not change during program execution. We then called the function outside of a transaction, and used the stored value in place of a function call.

Finally, we removed remaining `fprintf` (which log certain events to `stderr` when a program-wide flag is set), `perror`, and `sem_post` calls by registering `onCommit` handlers. In GCC, these handlers all run after the respective transaction commits *and releases all locks*. That being the case, our use of `onCommit` handlers has the potential to produce output in a different order than a lock-based program, since I/O performed by the handler does not complete atomically with the associated critical section. Furthermore, in the case of `perror`, we could not simply delay the function, but instead saved the `errno` and then called `strerror_r` in the commit handler. Unlike I/O, there are no concerns about ordering when delaying `sem_post` via an `onCommit` handler, since the only uses of condition synchronization are to wake up maintenance threads.

The only other challenge when using commit handlers was that some code could be called from both transactional and nontransactional contexts. GCC's TM does not expose the function for checking if a thread is in a transaction. We made this function visible to the program, in order to check for cases when the onCommit handler should be registered, versus those times when the handler should run immediately.

The performance following these changes took a remarkable turn. As shown in Figure 9, running times dropped to almost as low as the previous best, the simple IP-Callable branch. More importantly, when there is no mandatory serialization, transactionalization of item locks performs better than the use of privatization. Table 4 verifies that transactions no longer serialize at begin time, or due to an unsafe call during their execution. With all unsafe operations removed, the impact of serialization due to aborts becomes the dominant factor distinguishing the two approaches to item locks. The transactional version, which does not prioritize the small lock acquire/release transactions of the IP branches, has fewer serializations, leading to lower execution times.

4. Modifying GCC-TM

In the previous section, we eliminated *all* relaxed transactions, to include those that performed logging (via `fprintf` and `perror`) and those that ultimately would terminate the program (via `assert` and `abort`). Throughout this paper, we have claimed that atomic transactions provide a perfor-

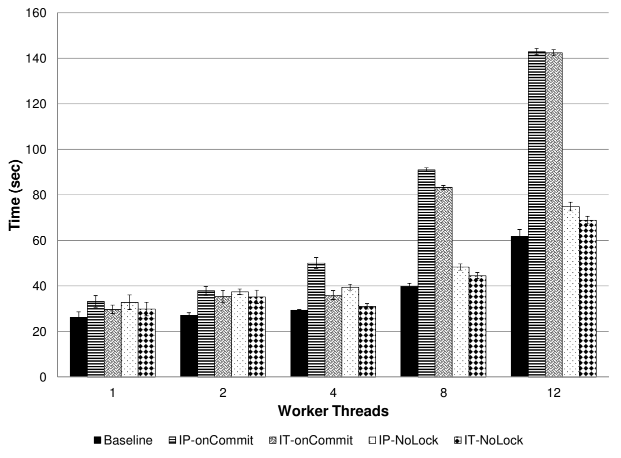

Figure 10: Performance without the readers/writer lock.

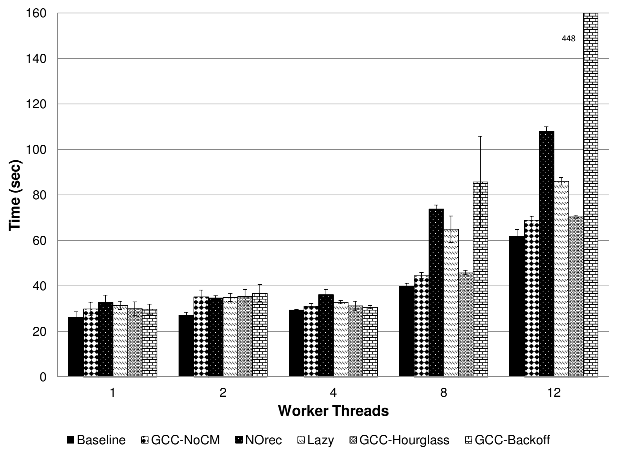

Figure 11: Comparison to other TM algorithms and contention managers.

mance model to the programmer. To illustrate the significance of this model, we made two further changes.

The GCC implementation of TM assumes that serialization is a common occurrence. To make serialization cheap, all transactions acquire a single global readers/writer lock in read mode when beginning a transaction, and release it on commit or abort. Transactions that require serialization acquire the lock in write mode. The availability of this lock makes switching to serial mode cheap, simplifies corner cases related to thread creation and joining, and enables a simple contention management policy, wherein a transaction becomes serial after 100 consecutive aborts.

The single readers/writer lock is an obvious bottleneck, but for programs with even a single relaxed transaction, serialization may be required for correctness. This holds true even with other contention managers [30]. Once we removed the last relaxed transaction, we removed the readers/writer lock from the GCC TM library, and added a separate lock exclusively for thread creation/joining. We then added a variety of simple contention managers (exponential backoff [14], a modified form of serialization called "hourglass" [10, 18], or simply no contention management), as well as a "lazy" STM algorithm[2] and the NOrec STM algorithm [4].

Figure 10 repeats the IT and IP results with onCommit handlers, and adds two new curves for GCC without the readers/writer lock. In these curves, there is no contention management: transactions immediately retry until they succeed. At high thread counts, we now can see that contention on the readers/writer lock is the primary source of overhead. Furthermore, performance is within 30% of the baseline memcached. This is despite expensive instrumentation (every read and write of shared data involves a function call, and every transaction boundary involves the creation of a checkpoint), and is in comparison to a highly optimized lock-based baseline.

[2] This algorithm uses the same lock table as the default GCC algorithm, but buffers updates and acquires locks at commit time

In Figure 11, we consider only the best-performing of these algorithms, IP-NoLock, which we now call "GCC-NoCM". We compare it to the Lazy and NOrec STM algorithms, also configured without contention management, as well as versions of the GCC STM algorithm using backoff or the hourglass contention manager (configured to prevent new transactions after 128 consecutive aborts).

With regard to contention managers, we see that the hourglass performance best matches performance without any contention management, making it an interesting candidate for cases where livelock is not expected, but programmers desire some guarantee of forward progress. At high thread counts, backoff performs poorly due to preemption. Furthermore, since the workload is heterogeneous, with dozens of source-code transactions of varying sizes and access patterns, backoff is not an optimal choice even at lower thread counts.

Similarly, we found that the GCC algorithm, which does not have buffered update, had the lowest latency and the best scalability. This is despite extremely high abort rates: at 12 threads, NOrec worker threads aborted once per 5 commits, Lazy worker threads aborted 14 times per 1 commit, and GCC worker threads aborted 12.6 times per 1 commit. In the case of NOrec, the frequency of small writer transactions induced a bottleneck on internal NOrec metadata; for both NOrec and the Lazy algorithm, the need to buffer byte-by-byte stores in `memcpy` and then read them later as words necessitated an expensive logging mechanism. Additionally, the variance in abort rate among threads was an order of magnitude lower for GCC than for Lazy, suggesting that Lazy was more prone to bouts of starvation.

These results suggest that real workloads do exhibit sensitivity to contention management and STM algorithm decisions, that real workloads place stresses on TM algorithms that research prototypes often ignore, and that expert programmers should be able to tune these parameters.

5. Recommendations for System Designers

We believe that the most viable path forward is for programmers to strive to avoid serialization, and system developers to both (a) optimize for the serialization-free case, and (b) facilitate the creation of programs that do not require serialization. To this end, we offer the following recommendations to designers of transactional runtime libraries, and to the authors of future TM language specifications.

Implementors Should Assume Serialization is Rare The ultimate success or failure of transactional programming will depend on performance. Serialization of relaxed transactions is a dangerous obstacle. The GCC TM library assumes serialized relaxed transactions are common, and optimizes that case at the expense of workloads in which non-serializing atomic transactions abound.

To be clear, relaxed transactions are necessary: they are the only way to perform I/O using transactional data, and in the absence of I/O, relaxed transactions should perform on par with atomic transactions. When transactionalizing legacy C programs, the only justification for atomic transactions is the static guarantee that they will not force serialization. While it ought to be possible to approach the performance in Figure 10 with a different lock implementation [17], even a scalable lock will be a bottleneck: if many transactions must serialize, then the serial fraction of the program will be too high to achieve good performance. It is probably better to increase overhead for the serial transaction to avoid bottlenecks [35, 36]. Note, too, that the latest version of GCC requires every *hardware* transaction to use this lock, suggesting that hardware TM will not achieve its full potential as long as serialized transactions are the common case.

The Specification Must Address Condition Synchronization The C++ Draft TM Specification does not allow a lock to be replaced with a transaction if that lock is also used in conjunction with a condition variable. In this work, we were able to leverage the specific communication pattern between threads, and replace condition variables with semaphores. Our technique does not generalize, and relies on the availability of `onCommit` handlers. Given the widespread use of condition variables in real-world programs, it is essential that the specification provide a solution. Otherwise, TM adoption will remain limited.

Specify More Transaction-Safe Libraries Serialized relaxed transactions are currently the only mechanism by which calls to standard library functions are possible. Our ad-hoc work-arounds, while necessary for evaluating the specification, do not generalize, and are not safe in the common case. Programmers should not develop ad-hoc approaches, marshal parameters into private memory, or use `transaction_pure`. The programming environment should also provide support for managing terminating exceptions (e.g., asserts), so that programmers can continue to develop robust, self-diagnosing code while using transactions.

Furthermore, transaction safety should not inhibit nontransactional performance. In the current specification, a single function body is used to produce two code "clones" appropriate for transactional and nontransactional uses, and thus transaction-safe code cannot include inline assembly. In the long term, application programmers and system developers alike will benefit from the ability to provide different implementations of a function depending on the calling context.

OnCommit Handlers Should Be Part of the Specification Our experience greatly benefited from `onCommit` handlers, which were removed from the 1.1 Draft Specification. Their return to the specification will allow programmers to avoid serialization, and to move non-critical code (i.e., logging) out of critical sections. We are less convinced that `onAbort` handlers are needed: the only role we envisioned for them in memcached was to employ backoff after a failed transaction. Specifying a simple contention manager interface is likely more appropriate.

Ignore Further Ease of Use Concerns Finally, we believe that the developers of transactional environments should not worry too much about ease-of-use. In particular, we found that manual annotation of safe and callable functions, while tedious, was not difficult. Furthermore, it was not error-prone, since incorrect `transaction_safe` annotations immediately generated compiler errors. Similarly, being forced to re-implement volatile variables, locks, and condition synchronization was unpleasant, but ultimately rewarding. We discovered, for example, that replacing item locks with transactions removed corner cases, and indeed there are several optimizations to memcached that are now possible on account of transactional reference counts and the elimination of delayed maintenance code paths. If locks and volatiles were transaction-safe, we would not have identified these opportunities.

6. Recommendations for Programmers

The success of transactional programming will depend on cooperation between the programmer and system developer, where both work together to avoid serialization. Our recommendations to application developers appear below:

Performance Model While there are semantic differences between relaxed and atomic transactions, for legacy code 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. For this reason, we believe that programmers should rewrite code to avoid relaxed transactions whenever possible.

Our study entailed only one application, written in C, and 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 mandatory 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. 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. Relaxed transactions play a necessary role in allowing I/O with transactional data, calling libraries that are not yet instrumented for transactions, and allowing transactions to interact with nontransactional threads. Programmers must be savvy enough to know when a relaxed transaction is the right choice.

Incremental Transactionalization is a Myth Our initial goal was to replace `cache_lock`. However, we had no choice but to expand our scope to include an array of item locks, as well as every major lock in the program, whether it was contended or not. When transactionalizing legacy code, we expect this experience to be the norm: if the last lock in a lock hierarchy is the only contended lock, then either the program lacks concurrency, or there is a trivial technique for splitting that lock into a set of low-contention locks. It is far more likely that some intermediate lock in the hierarchy will be contended, requiring the programmer to replace many locks with transactions.

While any transactionalization will entail considerable effort, it resembles a refactor more closely than a redesign. While we may be overly optimistic, we found that the tedious process of annotating functions and replacing locks and volatile variables provided an opportunity to identify optimizations in a number of areas (reference counting, interaction with the maintenance threads, statistics reporting), which we intend to improve as future work.

Expect Limited Tool Support Manually diagnosing the causes of aborts and serialization in our program was challenging, and we eventually extended the GCC TM library to use the Linux `execinfo.h` infrastructure to provide more meaningful profiling data. While static information about serialization could be generated at compile time, and dynamic information about aborts at run time, we did not see the absence as unreasonable. These tools will arrive eventually, but system developers' effort should first be placed on making standard libraries safe, and condition synchronization possible.

Similarly, tools for automatically annotating functions do not seem particularly valuable. To achieve good performance, we had to modify most source files in memcached. That being the case, the additional task of annotating functions was not overly burdensome. We expect tools to eventually make this task easier, but in our opinion it is not urgent.

Expect to Fork the Code Ultimately, we believe that the annotations and transformations we made to memcached are too complex and widespread to hide behind macros. Currently, memcached compiles on many operating systems and platforms, to include those that lack compiler support for atomic operations (e.g., `lock inc`), with macros hiding 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 C++ `atomic` data types any time soon, and if such a conversion occurs, it will likely be handled via even more macros. Preserving multi-platform support while performing the modifications discussed in this paper would create inscrutable, un-maintainable code.

7. 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. Nonetheless, our version of memcached is a valuable benchmark, which we will release as open source code.

Our main findings related to the cost of unintended serialization, the effort required to avoid serialization, and the contract between system developer and programmer that is most likely to result in a viable transactional programming environment. 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 standardization 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. We also thank Justin Gottschlich, Victor Luchangco, Jens Maurer, and Torvald Riegel for many helpful conversations about the Draft C++ TM Specification.

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. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, Oct. 2005.

[4] L. Dalessandro, M. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[6] A. Dragojevic, Y. Ni, and A.-R. Adl-Tabatabai. Optimizing Transactions for Captured Memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[7] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On The Power of Hardware Transactional Memory to Simplify Memory Management. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing*, San Jose, CA, June 2011.

[8] P. Dudnik and M. M. Swift. Condition Variables and Transactional Memory: Problem or Opportunity? In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[9] 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.

[10] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free Algorithms Can Be Practically Wait-free. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sept. 2005.

[11] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[12] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[13] 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.

[14] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, Boston, MA, July 2003.

[15] 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.

[16] 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.

[17] Y. Lev, V. Luchangco, and M. Olszewsk. Scalable Reader-Writer Locks. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

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

[19] V. Luchangco and V. Marathe. Transaction Communicators: Enabling Cooperation Among Concurrent Transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, San Antonio, TX, Feb. 2011.

[20] 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.

[21] N. Mathewson and N. Provos. Libevent – An Event Notification Library, 2011–2013. http://libevent.org/.

[22] 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.

[23] 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.

[24] 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.

[25] L. Poettering. Measuring Lock Contention, 2009–2013. http://0pointer.de/blog/projects/mutrace.html.

[26] 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.

[27] T. Riegel, C. Fetzer, and P. Felber. Automatic Data Partitioning in Software Transactional Memories. In *Proceedings of*

the 20th ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 2008.

[28] 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.

[29] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[30] 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.

[31] 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.

[32] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with Isolation and Cooperation. In *Proceedings of the 22nd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Montreal, Quebec, Canada, Oct. 2007.

[33] M. Spear, V. Marathe, L. Dalessandro, and M. Scott. Privatization Techniques for Software Transactional Memory (POSTER). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[34] 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.

[35] M. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, Sept. 2008.

[36] 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.

[37] 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.