Software Transactional Memory in the Linux Kernel

Garrett Bressler
Department of Computer Science
Brown University
garrett_bressler@brown.edu

May 2012

Abstract

Transactional memory (TM) is a synchronization paradigm which is an alternative to locking. Instead of relying on the programmer to manage entry into critical sections, TM realizes automatic conflict resolution through conceptually atomic *transactions* and attempts to avoid many of the classical problems associated with locks. Traditionally, TM is thought of as a system implemented at the processor level, though implementation in software is also possible. This paper examines the latter approach, by studying the effects of replacing locks with software transactions in FUSE (Filesystem in Userspace), which in part is implemented as a module bundled with the Linux kernel. Results show varying performance in different cases, with performance penalties for TM at 20% or lower under many scenarios, but over 1000% in a few instances, and show that 4-core transactionalized FUSE is frequently more efficient than its 2-core counterpart.

1 Introduction

Transactional memory, or *TM*, provides an alternative to the use of mutual exclusion, which dominates synchronization in multithreaded systems today. Instead of creating separate locks to manage access to separate shared data structures, the programmer specifies *transactions* covering sequences of instructions that may access any shared memory. They have the properties of *linearizability* (separate transactions take effect in their real-time order, and there is a canonical ordering for the transactions) and *atomicity* (a transaction either succeeds completely or does nothing at all). [10] In this way (except perhaps in critical parts of the TM implementation itself) there is no explicit locking, but rather the TM system detects and resolves conflicts during runtime.

Special measures must be taken to enforce the theoretical guarantees of TM discussed earlier, and to ensure that in all likelihood some thread will be able to make progress. First, an implementation must ensure linearizability and atomicity: this is usually done through a versioning system which makes distinctions between the current and tentative versions of shared memory, and atomically starts and commits transactions (likely through a machine compare-and-swap operation). Beyond atomicity and linearizability, an implementation might offer a *non-blocking* guarantee. Roughly speaking, "Non-blocking" means that, at least when certain conditions are met, some thread is guaranteed to make progress. There are several kinds of non-blocking guarantees, some stronger than others. One of these is *obstruction-freedom*, which ensures that if a thread runs long enough on its own, it will eventually make progress. An implementation might also satisfy the strictly stronger guarantees of *lock-freedom*, meaning that at any time *some* thread will eventually make progress, or *wait-freedom*, meaning that at any time all threads will eventually make progress. [9] Herlihy et al. [10] argue that settling for obstruction-freedom, though theoretically inadequate for avoiding starvation, in practice is usually

sufficient and greatly simplifies implementation. To accomplish this goal, an implementation should provide one or more *contention managers* to handle behavior for when two transactions conflict. Such behavior could be as simple as aborting the thread that caused the conflict, or could involve using techniques such as adaptive backoff and prioritization to forgive the conflicting thread until some threshold is met. [10]

Some authors, however, argue that even obstruction-freedom is not necessary, at least in STM. Ennals [6] essentially argues that obstruction-freedom is not a strong enough guarantee to matter. For example, it does not guarantee that a given transaction will be able to complete, but only that *some* transaction will be able to complete. Ennals gives the example of a year-long transaction operating on an object, which either must have exclusive access to the object for a year, or can never complete. He also notes that, even if in theory a task switching out in the middle of a transaction can block others from running, in reality this is temporary and its occurrence can be minimized. Most importantly, Ennals makes the case that implementing obstruction-freedom requires foregoing many optimizations involving cache locality and processor sharing. With all this in mind, the system we use, TL2-x86, is not obstruction-free.

Herlihy and Moss [11] describe a TM interface similar to the following. The body may include special load/store instructions that refer to or depend upon shared memory — these are the instructions managed by TM. The LT and ST are the transactional load and store operations, respectively. They are *tentative* updates and reads for the values on which they operate. Note that these are special instructions and must be used explicitly: a processor would also support regular load and store, which have no effect on the transaction. The data locations that a transaction accesses through LT or ST are its *read set* and *write set*, respectively; the combination of these two is the *data set*. Two concurrent transactions *conflict* if and only if the write set of one has nonempty intersection with the data set of the other.

Upon reaching a COMMIT, all tentative updates are applied if no conflicts occurred during the transaction, returning *True*; otherwise the transaction *aborts*, discarding all tentative updates, and returns *False*. [11] also describes a VALIDATE instruction, which aborts a transaction early if there have been conflicts. Finally, a program can decide to unconditionally abort a transaction with an ABORT instruction.

In the system just described, an aborted transaction simply fails to commit — the programmer must decide how to deal with a failed transaction. However, in the systems we shall discuss, a transaction is normally retried automatically. Therefore it is helpful to think of the inclusion of a START operation, which commences a transaction. When a transaction aborts, it jumps back to the START instruction and retries the transaction. From the programmer's perspective the code between START and COMMIT is executed atomically, and the code that comes after is what happens after the transaction has (successfully) completed.

1.1 Motivation

The most popular method of synchronization, locking, has been the dominant method for realizing synchronized systems since the dawn of multithreading. But despite its persistence, locking suffers from numerous well-known drawbacks. From the programmer's perspective, two of the greatest pitfalls are *race conditions*, in which improperly non-synchronized code leads to illogical behavior, and *deadlock*, in which cyclical lock dependencies cause two or more threads to be blocked forever. Another problem, *priority inversion*, occurs when a high-priority process is blocked waiting on a lock held by a lower-priority process. Worse, the process holding the lock may unexpectedly abort, in which case this becomes a deadlock. A programmer must anticipate such problems ahead of time ahead of time and deal with those accordingly. There is usually a tradeoff between parallelism and ease of correct implementation. At one extreme, the program may be single-threaded and as such require no synchronization at all, and at the other individual components of a data structure (such as nodes in a tree) may each have their own locks, requiring careful consideration of possible conflicts.

The potential number of errors rises rapidly with the number of locks, so in many cases one settles for no synchronization or for a few coarse-grained locks, and the potential benefits of synchronization are never fully realized. [17]

Transactional memory aims to relieve the programmer from the burden of making this tradeoff. Because of the dynamic nature of transactions, the programmer does not have to determine ahead of time which object accesses may potentially conflict. Furthermore, the automatic handling of conflict resolution, and (if present) the presence of guarantees such as obstruction-freedom, allow the contention manager to ensure with high probability that a thread will make progress. This paper examines the use of transactional memory in operating systems, a domain in which correct synchronization is of paramount importance but is oftentimes very difficult to achieve. Yet although operating systems and OS developers could potentially benefit substantially from TM, relatively little research to date has been conducted on this topic.

1.2 Methods

In order to be able to study TM in as close to a real-world situation as possible, we have decided to study STM in the Linux kernel. There are two parts to this endeavor. First, we needed to place an STM subsystem in the Linux kernel. For this, we use an adapted version of the TL2-x86 transactional memory system. [5] Second, we wanted to change some part of the Linux kernel in such a way that would model how a direct change to TM would affect behavior. For this purpose we have chosen to convert certain locked sections in FUSE to use transactional memory, creating "FUSE-TM". FUSE (Filesystem in Userspace) is an intermediary between the kernel FS subsystem and userspace. This allows filesystem writers to implement the entire filesystem at the user level, which could be useful for portability, security, or other reasons. [7] With this in mind, there are several filesystems layered above FUSE that can be used to test it. We have chosen to test using ZFS-FUSE, an implementation of ZFS on top of FUSE. [23]

FUSE is an ideal subject for several reasons: filesystems are easy to benchmark, it is a significant project in real-world use, its locking is relatively simple and straightforward, and it does not have too many sections that are problematic to convert. We have executed several tests intended to maximize the parallelization of FUSE. We compared performance between the original FUSE and FUSE-TM and between two and four cores, and also in FUSE-TM measured the extent to which transaction conflicts were actually occurring.

Our tests were conducted on a 2-core and a 4-core x86-64 machine, the former a 2 GHz AMD Athlon 64 X2 Dual Core with 2 GB RAM, the latter a 2.4 GHz Intel Core 2 Quad with 4 GB RAM. The operating system was Debian 6.0.4 ("squeeze"), and our kernel is a modified version of Linux 3.1.0.

2 Related Work

Hardware transactional memory (*HTM*) and *software transactional memory* (*STM*) are the main two ways to realize transactional memory. HTM is the traditional conceptualization of transactional memory, and originally most research was conducted with this in mind. The first paper to articulate something similar to the concept of TM seems to be [2], which is based on page locks and relies on the hardware to manage per-process locking of transactional memory. However, [11] is the first paper to truly articulate the modern conceptualization, and as such most of the terminology and philosophy used to think about TM begins here. In this paper, a dual-cache system is used, with both the main cache and "transactional" caches private to the processors. Transactional operations are made in the executing processor's transactional caches, and when a transaction successfully commits, these lines are made available to other processors and to the main cache. If the transaction aborts, the cache lines are invalidated. Papers such as [14, 8, 3] extend this concept to allow for features such as unbounded transactional memory and implicit transactions, but we will not delve into these details any further.

Unfortunately, despite its potential usefulness for authors of software code, and the relative simplic-

ity of the architecture just described, only one physical architecture that supports transactional memory has been publicly released, the BlueGene / Q supercomputer from IBM. [12] Hence, most HTM research heretofore has taken place on conceptual or simulated architectures.

Of greater importance to us is STM. Obviously, STM has the advantage of portability: it can be architecture-dependent aside from machine instruction(s) to ensure atomicity, typically read-modify-write operations such as compare-and-swap. [20, 10] Additionally, STM is more flexible in that differing implementations and paradigms can be used in the same system or perhaps even in the same application. But it also has the obvious drawback of performance, with the conflict resolution implemented in software instead of at the processor level. Shavit and Touitou [20] were the first to propose implementing TM in software, and use an approach based on locking transactional locations in a predetermined order. The disadvantage of this approach is that the transactions must be defined *statically*; that is, the memory accesses in any transactions must be specified before compilation. A more intuitive approach is given by [10], which specifies a *dynamic* STM system implemented in Java and C++. This approach involves having a transactional object which points to "old" and "new" versions of the data, as well as to the last transaction to operate on the object. When a transaction first accesses an object, the object is copied and modifications are made to the "new version"; when a transaction succeeds, the transaction's status is atomically changed to "commit", which causes the new version of an object to be interpreted as the current version of the object and the old object version to be ignored.

There is also research into the application of STM towards existing problems. In recent years, papers examining the use of STM in application-level software have proliferated, [22, 1, 15, 4] being just a few examples. However, research on TM in *operating systems* is relatively sparse. [21] describes how to adapt TM to better support operating system events such as context switches and paging, but doesn't explicitly deal with how to implement OS facilities to take advantage of these features. TxLinux [18] is a system of significant interest to us, as it is the only other known implementation of TM in the Linux kernel. However, TxLinux's approach differs radically from ours. For one, it uses HTM via a simulator, MetaTM, which behaves much like the original system described in [11], with the added benefit of unbounded transactional memory. Furthermore, while we are replacing locks manually with transactions, TxLinux's approach is "automatic": critical sections protected by locks in the kernel are replaced by transactions. A major problem with doing this is the irreversibility of I/O operations: when such an operation takes place, it cannot simply be reverted or discarded. Whenever an I/O operation is encountered during a transaction, the transaction aborts and a spinlock is used to protect the section is instead. To minimize the number of times this is necessary, a transaction is suspended when an interrupt occurs, so I/O operations during an interrupt need not interfere. In this way, TxLinux uses a "best effort" approach to replacing locks, requiring essentially no restructuring on the programmer's part.

The last system we shall describe, "Transactional Xinu" [19], probably comes closest to what this paper attempts to accomplish. Transactional Xinu, like its parent Embedded Xinu, is a research kernel which stays compact while still making use of modern kernel facilities. Schultz uses the Intel 2PL library [13] to implement transactions in Xinu, adapting it to be compatible with the kernel. In his research, he adds transactions to certain device drivers to implement synchronization, and observes the effects of transactionalization. In particular, he compares the performance between a locking Ethernet driver and its transactionalized counterpart, and notes performance not much worse than the original version.

To our knowledge, this paper is the first to examine the implementation of transactional memory in a production-use kernel on a physical architecture. It is, in a sense, a combination of the research on TxLinux and Transactional Xinu.

3 Implementation

3.1 TM Subsystem

The STM system in use is adapted from TL2-x86 [16], a general-purpose C library for transactional memory on the x86 and x86-64 architectures. TL2-x86 itself is based on the TL2 system by Dice et al.[5] Several things in particular made TL2 an excellent candidate for porting to the Linux kernel. First, it is written entirely in C, which is a requirement for interfacing with the kernel. Second, it is very self-contained: most of the special facilities, even the CAS, are implemented as part of the library, and the ones that aren't are either relatively unimportant, or are procedures like `malloc()` that can readily be replaced by equivalent kernel facilities. Third, it is designed to be efficient — it foregoes obstruction-freedom so that it can make optimizations. Finally, TL2 does not require a special runtime environment to enforce safe behavior — it can be used in a normal C program.

The TL2 system deals with transactions as follows. There is a global logical clock which is used as a version number, and is updated through a CAS instruction whenever a write operation occurs. There is a spinlock for every memory location transactionally accessed, and each such spinlock has such a version number. There is also a version number rv associated with each transaction. The protocol for a transaction in TL2-x86 is as follows: [5]

- When doing an LT operation: before doing the load, read the associated location's lock's version number. After the LT is complete, check that a) the version number did not change, b) the lock's version number rv is less than the transaction's version number, and c) the lock is (and was) free. If any of these conditions does not hold, the transaction aborts.

- When doing an ST operation: If someone else owns the lock, wait until it is free for some bounded number of tries. If this fails, abort. Also, if we observe that some location read set is incoherent (that is, the version number of the lock for that location exceeds rv), abort. Otherwise, tentatively update the value by putting the new value in the log and add it to the write set (if it is not already there). Note that we never actually acquire the lock.

- When doing a COMMIT operation: Try to acquire all the locks for locations in the write set in some order. For each lock, use some bounded number of tries, and abort if this is exceeded. Check that the read set is coherent; if not, abort. Update the global version clock to a new value wv. Update the addresses in the write set with the new value. Set the version number for each lock to wv. Release the locks.

- When doing an ABORT operation: Release any held locks and jump back to the start of the transaction.

Most of the necessary modifications involved hooking the TL2 system up with the kernel. The basic unit of execution is modeled as a `TxThread`, which stores all transaction state. Normally, it is up to the programmer to handle initialization and destruction of this structure, but in the kernel it is not easy to decide when to create or destroy it. Nonetheless, this decision is important because the default size of a `TxThread` can be quite large, due to storage for the read and write sets. One option would be to create it in `fork()`, the time at which new a new `task_struct` (basically the kernel's concept of "thread") is created, and destroy it in `exit()`, when the `task_struct` is destroyed. The problem with this is that we might initialize a thread which creates a `TxThread` and never uses it. In fact, this would be by far the most common scenario, given that presently transactions occur only in a single module of the entire kernel. Our approach is to initialize a `TxThread` only at the first time a transaction is actually encountered, and then keep it around until the thread exits. The downside to waiting until `exit()` for destruction is that a thread might only encounter only a few transactions and then go on to do something else, at which point it is left over with a stale `TxThread` which it never needs. However, if a thread encountered a transaction at least once, it is a very good bet that it will use transactions often in the future, so keeping the `TxThread` around is probably a good

idea. Another issue is that initialization can fail (normally from running out of memory) and dealing with this at thread initialization. So suddenly starting a transaction is now an operation that can fail, and it is probably not clear what to do when this happens. On the other hand, if `TxThread` creation fails the system is probably so overloaded anyway that any hope of getting useful work done is lost.

The interface we will describe is in terms of simple wrappers we have defined around the TL2-x86 interface proper. Mostly they just ensure that the `TxThread` in use is the `TxThread` in the `task_struct`, and then call the corresponding `Tx*` function. The exception is `tm_start()`, which may have to perform `TxThread` initialization, as described above. What follows is a brief description of the main procedures used by modules that wish to make use of transactions.

- `tm_start(env, roflag)`: Starts a transaction. Before this, the user should call `setjmp` on a `jmp_buf` field. `env` is a pointer to the `jmp_buf`, and `roflag` is a pointer to an optional flag indicating if this transaction is read-only (We never make use of this, but TL2 does have special optimizations for this case — see [5]).

- `tm_commit()`: Tries to commit the current transaction.

- `tm_abort()`: Aborts the current transaction.

- `tm_load(l)`: Transactionally loads a 64-bit word from location `l`.

- `tm_store(l, v)`: Transactionally stores 64-bit word `v` in location `l`.

- `tm_store_local(l, v)`: Transactionally stores 64-bit word `v` in location `l`, where the programmer guarantees that `l` will only be accessed by the current transaction. This is not strictly necessary, but optimizes the store: the new data at `l` must be revoked on abort, but it need not appear in the write set. The reasoning is that `l` may need to be updated tentatively if it was used before the transaction started. In the case of an abort, we want `l` to have the value which it had at the time of `tm_start()`, so to ensure atomicity we need TM to restore its original value when the transaction aborts.

- `tm_alloc(sz, flags)`: Transactionally dynamically allocates `sz` bytes, returning the location or `NULL` on failure. That is, the allocation is revoked (freed) on abort. `flags` includes the kernel memory allocation flags `include/linux/`, which specify the memory pool and mode of allocation. (This should probably be `GFP_KERNEL`.)

- `tm_free(p)` tentatively frees the memory at `p` allocated with `tm_alloc()`.

In addition to this, there are macros `TM_{LOAD/STORE/STORE_LOCAL}{8/16/32/64}()`, which operate on various-size words. But there are still more operations needed to deal with the idiosyncrasies of kernel operations. Before we noted that certain kernel operations do not fit nicely in the TM model. Most of these operations cannot normally appear in a transaction. However, we might want critical sections containing them to be synchronized with transactions, because they might have to be executed atomically from other transactions. The solution is the `TM_LOCK()` protocol. `TM_LOCK()` allows one to define a spinlock to protect these operations. Once this happens, the kernel does whatever operations it needs to (as normal code, not under a transaction) until it frees the lock with `TM_UNLOCK()`. The waiting on and acquisition of the spinlock occur in one transaction. Other transactions that need to be exclusive from what is protected by the `TM_LOCK(p)` should honor it by including `TM_WEAK_LOCK(p)` at the beginning of the transaction. `TM_WEAK_LOCK()` works by spinning on the spinlock until it is free, but does not acquire it. Thus when some other thread calls `TM_LOCK()`, this thread is preempted and must abort its transaction. In summary, the lock operations are:

- `TM_LOCK(p)`: Spins while someone else holds the spinlock `p`, and then acquires exclusive ownership of `p`, which is a 64-bit word. (Called on its own, not in a transaction)

- `TM_UNLOCK(p)` Releases ownership of the spinlock `p`.
- `TM_WEAK_LOCK(p)` Called by transactions at the start to ensure mutual exclusivity from operations protected by `p`.

Note how this is not unlike the approach used by TxLinux, [18] except with our system the programmer must define statically which sections are protected by locks and which are protected by transactions. In this paper, we have simply left a critical section protected by a `TM_LOCK` rather than a transaction if it would execute some problematic operation.

3.2 FUSE-TM

The sections we have transactionalized are those protected by the spinlock `lock` in the `fuse_conn` struct. `fuse_conn` is the structure that gives a connection between the user and a mounted filesystem, and orders requests to the filesystem. Among other times, `fuse_conn->lock` is locked when modifying the attributes of an inode, when creating an inode, and at certain times when reading and writing pages. In our conversion, all instances of `fuse_conn->lock` have been removed and replaced with synchronization based on TL2-x86.

However, as we hinted at before, some of these locked sections are difficult to straightforwardly convert, because they modify state outside of FUSE or use a kernel facility that doesn't know about TM. For these, we have simply replaced the spinlock with a `TM_LOCK()` on a TM spinlock. For the transactionalized sections to remain synchronized with these parts, every transaction invokes `TM_WEAK_LOCK()` on this spinlock, as we described in the previous subsection. In essence, these sections are not transactionalized, but they are guaranteed not to interfere with the operation of the existing transactions. To understand how this can be used, it is probably easiest to study an example. The following is the code for `queue_request()` in `dev.c` (the function is called under lock)

```
static void queue_request(
    struct fuse_conn *fc,
    struct fuse_req *req)
{
    req->in.h.len = sizeof(struct fuse_in_header)
            + len_args(req->in.numargs,
                    (struct fuse_arg *)
                    req->in.args);
    list_add_tail(&req->list, &fc->pending);
    req->state = FUSE_REQ_PENDING;
    if (!req->waiting) {
        req->waiting = 1;
        atomic_inc(&fc->num_waiting);
    }
    wake_up(&fc->waitq);
    kill_fasync(&fc->fasync, SIGIO, POLL_IN);
}
```

The code that waits on `fc->waitq` is located in `request_wait()` in `dev.c`. The thread makes itself non-runnable, deschedules itself, then (after it has been woken up) continues assuming a new request is ready. If, after this, it sees there are no pending requests, it returns with the error code `-ERESTARTSYS`. The assumption is that if it was woken up, then (barring some system failure) a request must have been made available. However, this does not hold if the `wake_up()` occurred in a transaction which later aborted, or if the thread wakes up before the transaction commits — in either case, no request has really been queued. Other problems with wait queues are also possible. Suppose, for instance, that instead of continuing on after being woken up, a thread checks a condition variable to determine if the resource it was waiting on has become available. (This is the behavior in particular for `fi->page_waitq`.) This will prevent the problem wherein a thread executes code prematurely. We have another potential problem, however: the thread might wake up before the transaction commits, thus entering an eternal sleep.

To correct this, one option might be to have the woken thread not permanently go back to sleep after it re-checks its condition variable — after all, if it's been woken up there is a high probably a transaction will complete very soon (or already has completed) and it doesn't have to wait much longer. A more general solution is to defer the processing of the wakeup to the point that the transaction actu-

ally commits. One issue with these approaches is that we don't observe the "tentative" consequences of the wakeup. That is, if the transaction that does the wakeup ends up depending on the woken-up thread in some way, we will not see the expected behavior and might stall. Though this is probably a pretty unlikely problem, it's useful to keep in mind that different circumstances may call for different solutions.

`kill_fasync()` sends a signal to the process that sent the request, to notify it that its request has been sent out. This results in a similar problem, except that the external process (which is beyond FUSE's control) decides how to respond to the signal. It may ignore it, or it may execute processing that should be delayed until the commit happens. Or, it may receive a signal twice (once as a side effect of an aborted transaction), prompting the execution of the wrong signal handler the second time. Fortunately, however, signal processing usually isn't very time-sensitive, so using a general strategy of postponing the signal until commit would probably suffice in most cases.

In any case, to keep behavior as straightforward and as close to original FUSE as possible, we have taken the conservative approach of leaving any potentially problematic sections locked. Fortunately, as our tests shall show, the sections that were transactionalized still make up a great proportion of the most important parts of FUSE.

4 Benchmarks

In each test, we measured at several configurations the amount of time taken for the 2-core and 4-core machines to complete the test each using standard FUSE (hereafter "FUSE") or transactionalized FUSE (hereafter "FUSE-TM"). Also, when using transactionalized FUSE, we counted the total number of aborts that occurred during the run. The final results were taken from a 20% trimmed average of fifteen runs per configuration (except for the last test).

For each test, there are five graphs. The first two compare performance at each configuration between the different versions of FUSE on two and four cores, respectively. The next two give the average number of aborts at each configuration when using FUSE-TM. The final graph compares the percentage increase in time at each configuration between the 2- and 4-core runs.

A general analysis of the tests follows in the Analysis section.

4.1 Write File

The first test involved many threads appending to a single file at once. The test was intended to determine how FUSE-TM handles sequential accesses on a single resource. The size of the file was kept invariant, while the number of threads was allowed to vary. In this and the next two tests, we took measurements for 64, 128, 256, 512, 1024, 2048, and 4096 simultaneous threads. To ensure simultaneous execution, the threads were synchronized with a barrier.

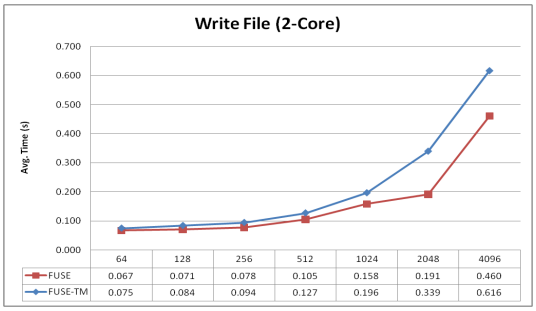

Figure 1: Performance for Write File, 2-core

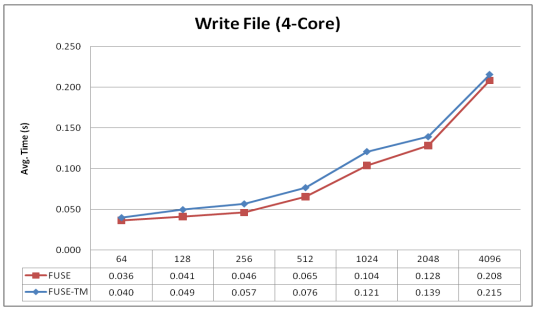

Figure 2: Performance for Write File, 4-core

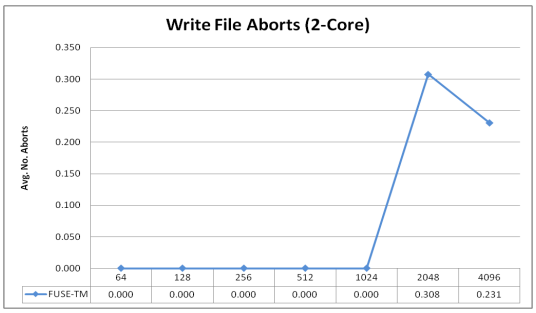

Figure 3: Aborts for Write File, 2-core

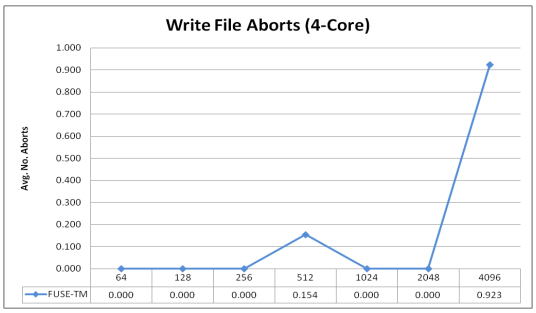

Figure 4: Aborts for Write File, 4-core

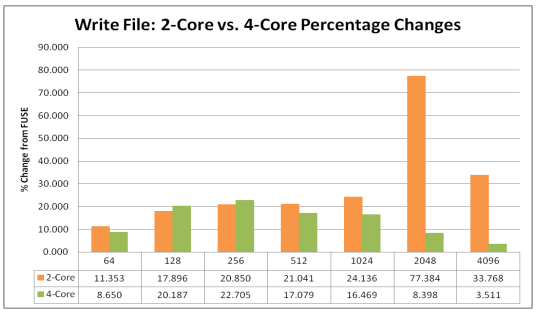

Figure 5: Comparative Performance for Write File

In every case, time is roughly proportional to the number of threads. In terms of performance compared with standard FUSE, at lower numbers of threads, the 2-core and 4-core performed roughly equivalently, but at the last two points showed opposite behavior, with a particularly noticeable spike at 2048 threads in the 2-core case. Aborts in either case were minimal.

4.2 Many Files

In this test, several threads each wrote their own 4-KB file to a single directory simultaneously. The test was intended to model the common scenario of several processes making use of the same directory, and to determine the behavior when several requests were executed in parallel. Only the number of threads was varied, and as in the last test, a barrier was used to coordinate the threads. The configurations used were the same as in the last test.

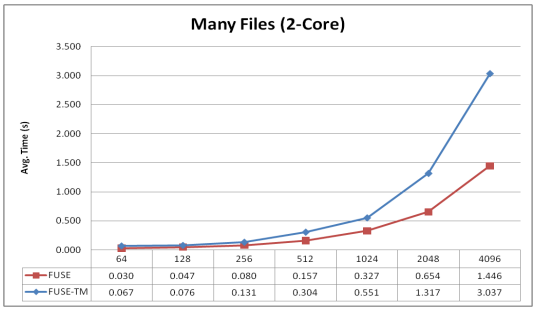

Figure 6: Performance for Many Files, 2-core

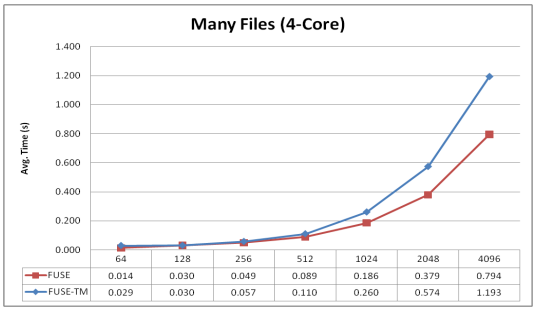

Figure 7: Performance for Many Files, 4-core

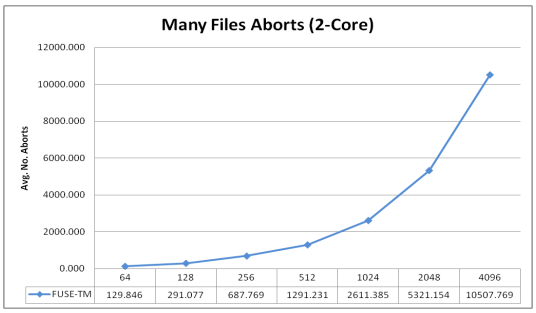

Figure 8: Aborts for Many Files, 2-core

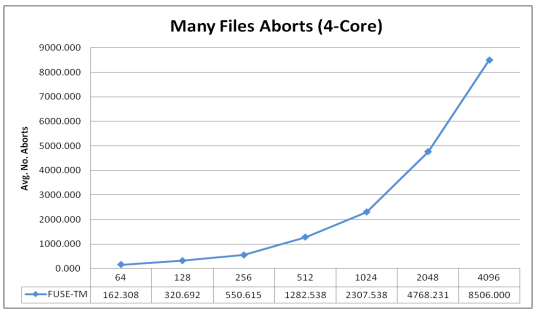

Figure 9: Aborts for Many Files, 4-core

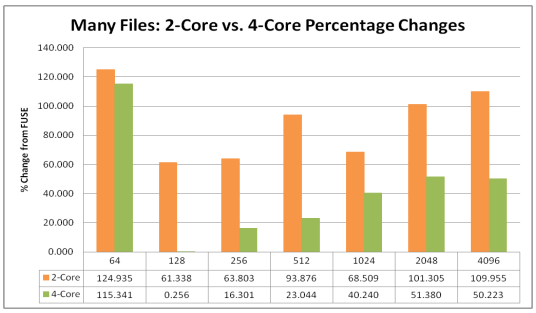

Figure 10: Comparative Performance for Many Files

Again we see a roughly proportional relationship between number of threads and time in each case, with the time about doubling at each point. Aborts exhibited similar behavior, and the number of aborts could get quite large with very many threads. In terms of time compared with standard FUSE, there was a general steadily-increasing trend, but with the 4-core performing far better in all except the first case. An exception was noted for the 64-thread case, in which both the 2-core and 4-core comparatively performed quite poorly.

4.3 Many Reads

In this test, many threads were reading from a single 1-MB file. Only the number of threads was varied, and as in the last two tests, a barrier was used to coordinate the threads. This test was mainly intended as the read-only counterpart of the first test, and models a very common use case for a filesystem. The configurations used were the same as in the last test.

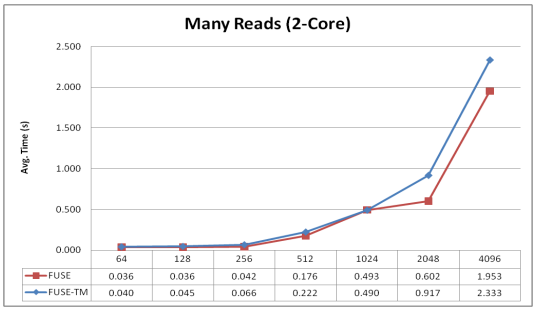

Figure 11: Performance for Many Reads, 2-core

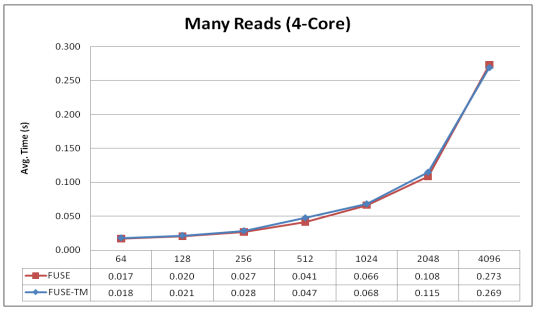

Figure 12: Performance for Many Reads, 4-core

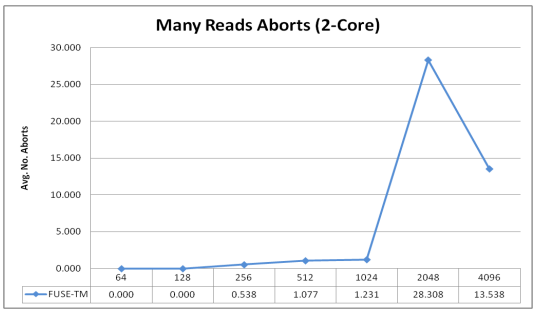

Figure 13: Aborts for Many Reads, 2-core

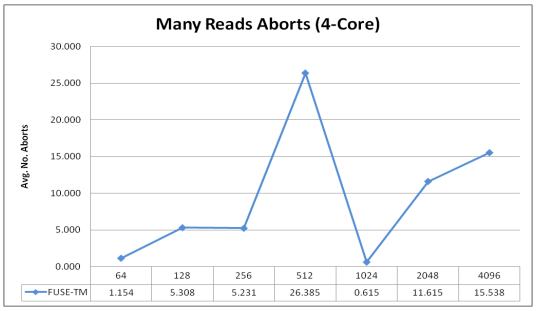

Figure 14: Aborts for Many Reads, 4-core

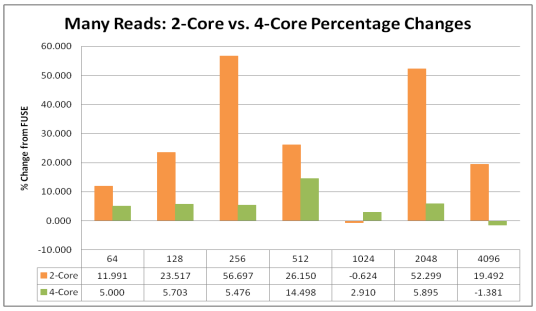

Figure 15: Comparative Performance for Many Reads

As in the last two tests, we note a generally proportional trend. Aborts did not follow this trend, seeming to spike unpredictably at certain points, and mostly being insignificant. However, they were more common than in the Write File test. Percentage changes also did not follow this trend, showing marked volatility in the 2-core case and staying relatively low in the 4-core case. It is noted, however, that relative performance was much better in the 4-core case.

4.4 Make Tree

This and the next test are both fairly dissimilar from the previous ones. We created a directory tree through a recursive procedure, using one thread per directory. With this test, we intended to determine the behavior of FUSE-TM under heavy load, as well as to see what happens when threads are being created asynchronously. We also wished to elucidate the consequences of having FUSE threads operate in separate directories. Instead of running several threads at the same time, having them perform the same task, and waiting for completion, we create threads dynamically. The test takes parameters (d, b), creating a directory tree b levels wide d levels deep. That is to say, the root directory, and every directory except the ones at the very bottom, contain b directories. Additionally, b files, of size $10000, 20000, ..., b \cdot 10000$ bytes, are created in each directory (again, except at the bottom). Including the top directory, the directories are nested d times, and the bottommost directories are empty.

The tree was created through the following recursive procedure. There is some number of threads, all performing the same function. Each thread is passed a path; this can be considered the "root" for the thread. If a thread notices that its root is at the dth level of the tree, it terminates immediately. Otherwise, it creates the b directories and b files in its root described in the last paragraph. For each of the b directories, the thread spawns a new thread whose root equals the path to the new directory (i.e., the name of the original root concatenated with the name of the new directory). Once it has started a thread for each new directory, this thread terminates, and the new threads start the procedure anew. Threads are detached immediately after they are created; that is, the parent thread does not wait for its child threads to run to completion.

In total $(1+b+b^2+...+b^{d-1})$ non-terminal threads are created. This is one greater than the total number

of files and directories created. Using the formula for a geometric series, this number is

$$\sum_{k=0}^{d-1} b^k = \frac{b^d - 1}{b - 1}$$

Note that in this test, it is harder to show a trend because the data points were varied on two axes, the breadth and the depth. Nonetheless, we have ordered the tests in terms of performance so that we can attempt to detect patterns.

The next three figures show the progression of Make Tree with breadth and depth equal to 3. Each black bar represents a thread, each rounded rectangle a directory, and each white square a file. The first thread "starts" at the (already-existing) top directory. In the next figure, it creates the files and subdirectories in that directory, and starts a thread for each subdirectory. Next, each of the new threads creates files in its own directory and three new directories each along with their threads. These new threads notice that their root has depth 3, so they terminate immediately.

Figure 16: Make Tree with depth = breadth = 3, start

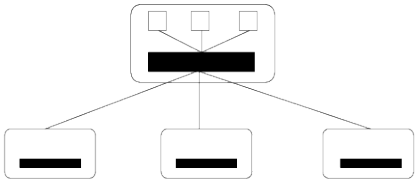

Figure 17: Make Tree with depth = breadth = 3, first thread has spawned three new threads

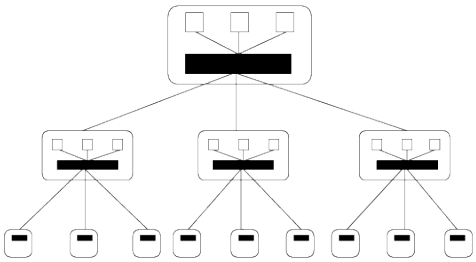

Figure 18: Make Tree with depth = breadth = 3, end

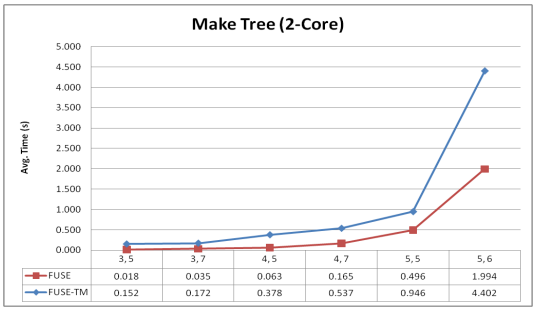

Figure 19: Performance for Make Tree, 2-core

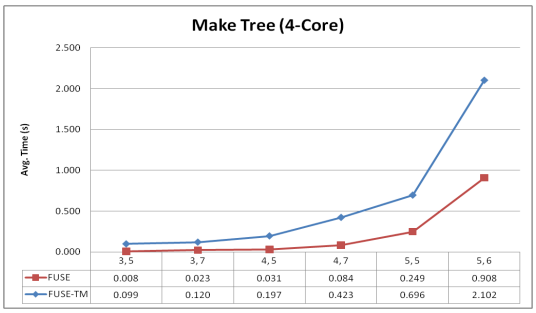

Figure 20: Performance for Make Tree, 4-core

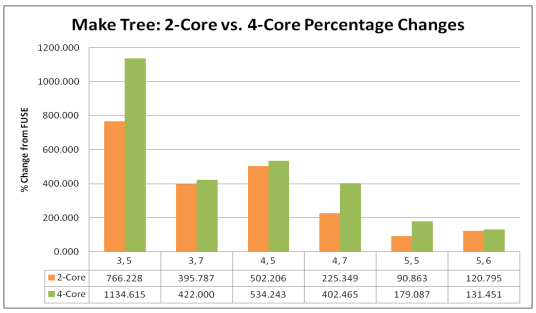

Figure 23: Comparative Performance for Make Tree

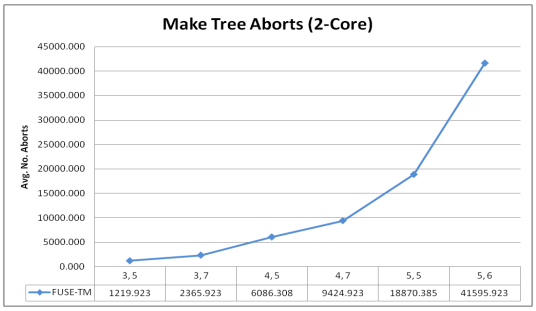

Figure 21: Aborts for Make Tree, 2-core

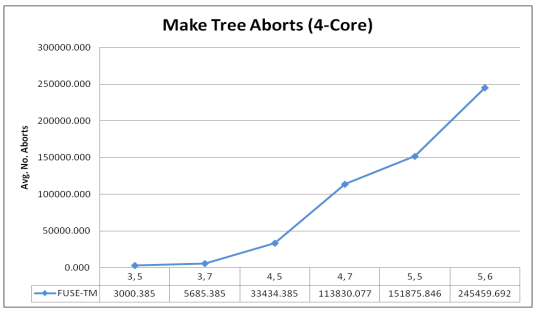

Figure 22: Aborts for Make Tree, 4-core

Overall, increasing either the breadth or depth could significantly impact performance. But depth seems to be more important than breadth in determining performance: namely, going from $(3, 7)$ to $(4, 5)$ and from $(4, 7)$ to $(5, 5)$ we are decreasing the breadth and number of threads (1093 to 341 and 5461 to 781, resp.), but this might increase time by as much as a factor of 3. The aborts followed a similar trend. Also note that the absolute number of aborts rapidly became very large.

In terms of percentage change comparisons, in general we actually have a decreasing trend, with "bigger" trees giving more favorable comparisons. Furthermore, the 4-core tests perform slightly worse than their 2-core counterparts. These are opposite trends from the first three tests. Frankly, in absolute terms the performance of FUSE-TM is quite poor, especially with "smaller" trees, where we see a 5- to 10-fold degradation in performance. Even in the best cases, performance was about halved by FUSE-TM.

4.5 Scan Tree

This is very similar to the previous test, except we are not *creating* the tree, but traversing the tree created from the previous test. It was intended to determine whether traversing a tree would yield a similar effect to creating it. Wherever we would create a file or directory in the last test, in this test we `stat()` it instead. (Only the time of the traversal, not the time of creating the tree, was measured.) Results were measured at the same configurations as the last test.

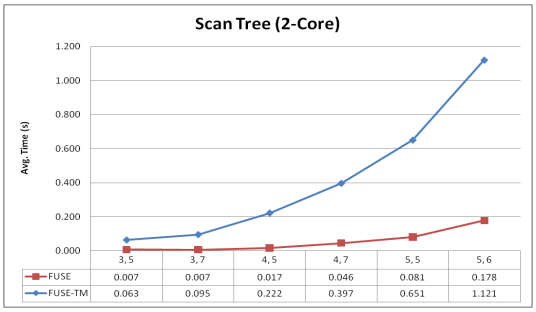

Figure 24: Performance for Scan Tree, 2-core

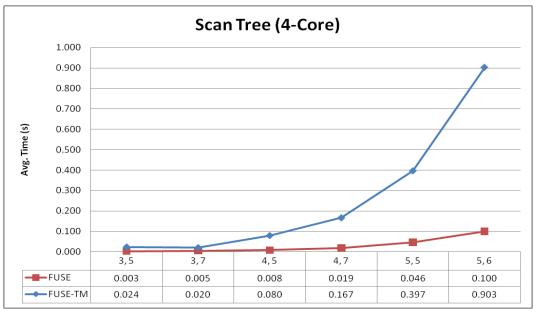

Figure 25: Performance for Scan Tree, 4-core

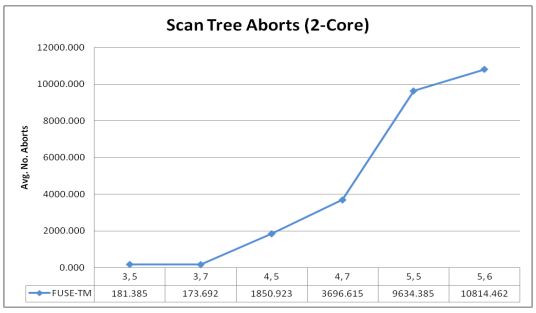

Figure 26: Aborts for Scan Tree, 2-core

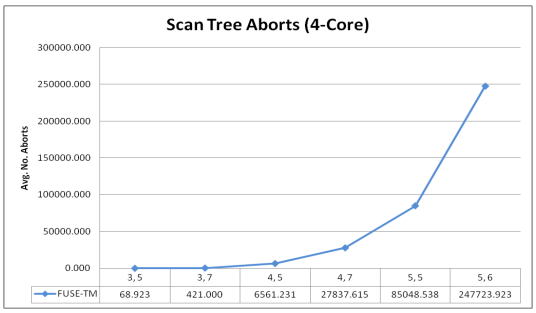

Figure 27: Aborts for Scan Tree, 4-core

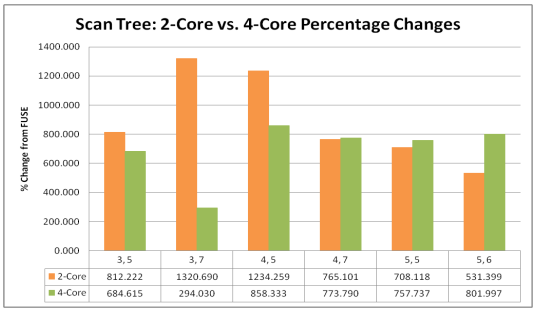

Figure 28: Comparative Performance for Scan Tree

In terms of trends, this is similar to what was seen with the Make Tree test. In absolute terms, FUSE performed very efficiently well, whereas FUSE-TM performed very poorly, markedly more so even than in the last test. However, we still see an improvement in terms of percentage increase in performance with "bigger" trees in the 2-core case; in the 4-core case, the percentage increase is roughly steady at just under 800 %.

4.6 Compiling

In this test, we complied the text editor Vim with parallel make. It was intended to show performance of FUSE-TM in a relatively realistic usage case. In attempting to keep the test realistic, we kept the total amount of parallelization fairly low. Our parallelization parameters to `make` were `-j 16`, `-j 32`, and `-j 64`. We also took just three measurements for each configuration, since the differences in results

from run to run were not significant.

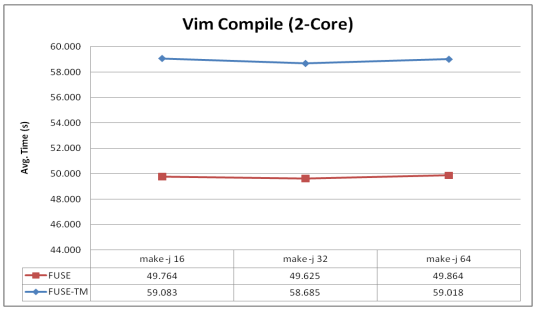

Figure 29: Performance for Vim Compile, 2-core

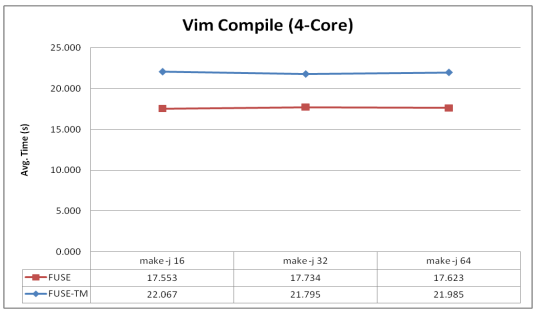

Figure 30: Performance for Vim Compile, 4-core

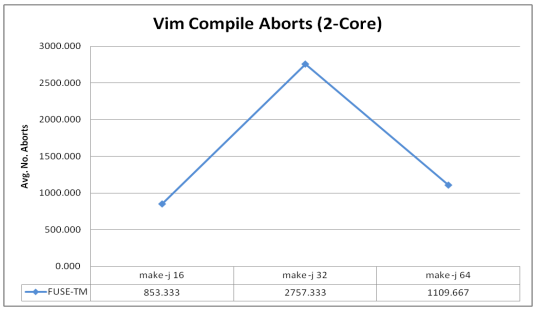

Figure 31: Aborts for Vim Compile, 2-core

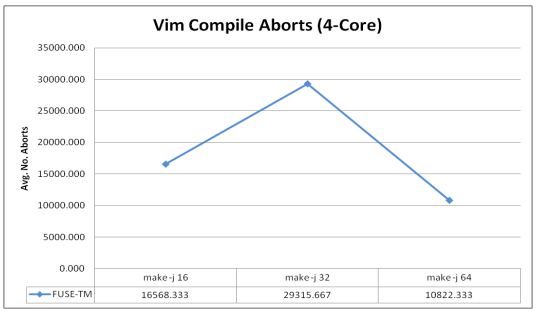

Figure 32: Aborts for Vim Compile, 4-core

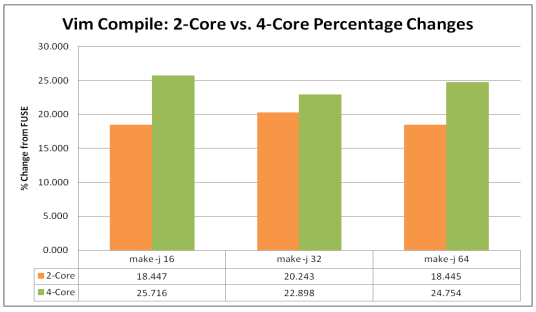

Figure 33: Comparative Performance for Vim Compile

Overall, we don't see much improvement in performance with more parallelization. In the 2-core case, FUSE-TM is roughly 20 % less efficient than FUSE, and about 25 % less efficient in the 4-core case. However, the amount of parallelization did affect the number of aborts, though not in a predictable way. The spike in aborts for `-j 32` in either case is notable, but it is not immediately explainable.

5 Analysis

As has likely become evident, it is hard to generalize about the net results of using STM in the Linux kernel. For one, different tests yielded radically different behavior, and we have only studied a small subset of the possible use cases of FUSE, let alone Linux. Also, at this initial stage, it is difficult to prove exactly what makes FUSE-TM perform admirably in certain

circumstances and much worse in others. However, the relationship between aborts and performance as well as the differences we see between tests give us some clues.

First, in the first three tests, more aborts tend to indicate worse performance in comparison to standard FUSE. In the Write File and Read File tests, we see fewer aborts and less of a trend of comparative performance becoming worse and worse. In the Many Files case, there seems to be a decrease in comparative performance with an increase of threads (which correlates with an increase in aborts). Intuitively, this makes sense: aborting is costly. It means having to run the same code twice (or more) in one thread. On the other hand, with spinlocks the critical sections are guaranteed to execute only once in each thread. The benefits of TM normally come in when aborting *doesn't* happen: that is to say, when, in the non-transactional version, one lock would be waiting on another but doesn't need to be.

But this relationship does not always hold: the Scan Tree and Read Tree tests indicate no increasing trend in comparative performance, even though the aborts show a dramatic increasing trend. This demonstrates that aborts are not the only factor that matters. Despite the increase of aborting transactions, there is likely also an increase in transactions that run in parallel with no conflicts. If the proportion of non-conflicting transactions were increasing, this would explain why comparative performance improves. The following observation bolsters this hypothesis. Each thread is operating on a separate directory. Threads at separate directories may conflict at the inodes that correspond to common components in their pathnames. In fact, at least one inode access will occur for each component; these accesses are necessary for FUSE to check permissions. (These accesses are not read-only because FUSE unconditionally updates an inode when it gets its attributes). However, in FUSE-TM they will not conflict at separate inodes, because they are entirely different instances. However, in FUSE, all inodes are protected by the same spinlock, so contention is possible on different inodes.

We observe that 4-core performs better than 2-core in many cases, particularly in the first three tests. However, in the Tree tests, it performs about the same or even worse. This is probably due to the dramatic increase in aborts for the 4-core case. It is possible that that the 4-core tends to run all the threads spawned by another thread at about the same time (which could happen if nearly each thread is assigned its own processor), while in the 2-core case one such thread often finishes completely before another can be scheduled.

There were many results that are difficult to easily explain. For example, the 64-thread Many Files test has unusually poor comparative performance (for both 2- and 4-core), as does the 2048-thread 4-core Write File test. In the latter case, for instance, the data indicate that it is not so much that FUSE-TM performed unusually poorly, but that FUSE performed unusually well. Certain numbers of threads may simply be scheduled "just right", or "just wrong" at certain configurations. The scheduler, of course, is oblivious to transactions, and does nothing to try to prevent contention. This alludes to one of the difficulties in analyzing kernel behavior: the factors involved go far beyond just FUSE and TL2 and are very difficult to theoretically model. We also note the unusual bump in aborts for `-j 32` in the compilation test, which also could have occurred due to unfavorable scheduling.

6 Conclusion

To some extent, it is difficult to deduce the actual significance of these tests or draw general conclusions from them, since they measure the effects of changing only a very small part of the Linux kernel. Nonetheless, the results show that TM in the kernel holds promise in many ways. First of all, the success of simply converting non-transactional code to transactional code testifies to the possibility of adapting parts of an OS kernel to use transactional memory. Using "TM locks" in certain parts proved to be a workable compromise between using transactions and maintaining the integrity of special kernel operations. Whether other parts of the kernel are as readily convertible as FUSE remains to be seen, but one should keep in mind that straightforward conversion is not

necessarily the way TM would be introduced to the kernel. Though both [18] and this paper used the approach of conversion *in situ*, a module designed to use TM naturally could differ significantly from its locking counterpart. In particular, writing from scratch could allow one to isolate any operations that would be problematic under a transaction, or to minimize interactions with the rest of the kernel. Finally, the use of TM diminishes the temptation of fine-tuning algorithms to make critical sections as small as possible, and the flexibility of transactions might allow one to choose from a wider variety of algorithms. For instance, an RB-tree must be implemented very carefully in order to take advantage of parallelism through locking, but an RB-tree synchronized by simply transactionalizing the sequential version shows impressive performance.[10] The greater freedom allowed by the transactional approach could effect algorithmic improvements that surpass the advantages of optimizing existing algorithms for fine-grained locking. Hence, the traditional efficiency and simplicity of locking may become less of a selling point for mutual exclusion in the long run.

While a performance degradation of 25-50% (as seen in many of our tests) is not desirable, it is not necessarily a deal-breaker – software that is already very efficient may not suffer greatly from such a penalty. Also, in many cases using four cores instead of two yielded a significant performance increase. If machines with more cores become standard in the future, the observed penalty could be reduced. Although this isn't a certainty (our compilation test, for instance, yielded worse relative performance for the 4-core version), the improvements from using four cores in some of our more intensive tests are promising.

Of course, there were instances of our tests that exhibited admittedly unacceptable performance for FUSE-TM. But one could make the argument that most users will not use FUSE as intensively as most of our benchmarks. The worst case scenario is important for showing the limits of TM, but it is also important to keep in mind that it may be rare in practice. Even today, many parallelizable operations are implemented as a single thread, and operating on its own part of the filesystem. This use case is far removed from benchmarks involving thousands of threads manipulating inodes in the same directory. Further testing will be necessary to show how likely worst-case or near-worst-case behavior is in practice. Nonetheless, it is conceded that our research does not show any performance benefits from using STM in Linux.

What, then, is the payoff of our research? FUSE is probably one instance where TM was not necessary to improve code maintainability. Most importantly, we believe that this research shows the *plausibility* of using TM to replace locking in the kernel, thus opening the door for conversions or rewrites of more complicated synchronous systems. By doing research on a system as straightforward as FUSE, it was possible to easily study the effects and successfully implement the correct behavior. The relative simplicity of FUSE also provides a relatively non-biased way of comparing the experiment to the control, since more involved systems might require modifications that cause it to deviate in ways that interfere with the effects of TM itself. Furthermore, it gives us hints as to how TM might behave in an actual filesystem. If TM were used to implement a filesystem (for which one could easily make the case that manual locking is a great burden on the programmer), then this research might yield some basic ideas of the contention one would expect in certain scenarios. In short, the research gives us some idea of the issues STM in a kernel might face, and argues against the notion that systems-level TM is a "lost cause".

7 Further Investigation

From our view, there are two obvious directions that follow from this research. The first direction is to investigate FUSE-TM further, in order to test the hypotheses in the Conclusions, and potentially to try to improve on these results. We could time the critical sections themselves, giving us a better idea of their cost with and without contention. We might measure, in addition to how much contention occurs, how often transactions run successfully in parallel, helping us answer some of the hypotheses posed about the Tree tests. Inserting some probes into the scheduler

might give us an idea of how often context switching occurs during a transaction and help answer our hypothesis about the differences between TL2-x86 on 2- and 4-cores. In order to gather information on exactly what parts of FUSE are generating the most contention, we might label the data to indicate how often and when each transaction occurs. In a related approach, we might write some tests to target specific sections of FUSE directly. Finally, we might attempt to transactionalize some of the sections that we did not in this paper. This would involve either replacing the special operations with transactional equivalents, or allowing for such operations to be separated from the rest, by e.g. having them be executed at the end of a transaction.

The second direction is outward, involving the use of TM in other parts of the Linux kernel or other systems software. Though trying to put TM in the heart of the kernel is probably beyond reach at this point, there are other viable targets. However, we might try to make the kernel more TM-aware by, for example, telling the scheduler not to switch out a task in the middle of a transaction, as suggested in [6]. Another logical place to go (and in fact one of the original ideas for this paper) would be to transactionalize a filesystem, be it a simple one such as UFS or a more modern one like BTRFS. We could also try to transactionalize a device driver, such as an Ethernet driver, as in [19]. And of course, if consumer-grade HTM ever becomes available in the future, replacing our STM implementation with HTM would give insight as to how many of our problems are caused by the inherent inefficiency of software algorithms.

8 Acknowledgements

We would like to thank advisors Tom Doeppner and Maurice Herlihy for their encouragement and support. Also, we owe a heavy debt to the authors of TL2 and TL2-x86, without which this research would not have been possible.

References

[1] N. Carvalho et al. Versioned transactional shared memory for the FénixEDU web application. In *Proceedings of the 2nd workshop on Dependable distributed data management*, pages 15-18, March 2008.

[2] A. Chang and M.F. Mergen. 801 Storage: Architecture and Programming. In *ACM Transactions on Computer Systems* 6(1), February 1988.

[3] W. Chuang et al. Unbounded page-based transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 347-358, 2006.

[4] J.W. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-Safe Dynamic Binary Translation Using Transactional Memory. In *Proceedings of the 18th international symposium on High-performance computer architecture*, pages 279-289, February 2008.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th international symposium on Distrubted computing*, 2006.

[6] R. Ennals. Software Transactional Memory Should Not Be Obstruction-Free. Technical Report Nr. IRC-TR-06-052, Intel Research Cambridge Tech Report, January 2006.

[7] Filesystem in Userspace. January 2012. Retrieved 16 April 2012 from `http://fuse.sourceforge.net/`.

[8] S. Hammond et al. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st annual internation symposium on Computer architecture*, pages 102-113, June 2004.

[9] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the 23rd international conference on Distributed computing systems*, pages 522-529, June 2003.

[10] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the 22nd ACM symposium on Principles of distributed computing*, pages 92-101, July 2003.

[11] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289-300, May 1993.

[12] IBM Corporation. IBM Systems and Technology Group. IBM System Blue Gene/Q. 2011. Retrieved 10 April 2012 from `http://www-03.ibm.com/systems/deepcomputing/solutions/bluegene/`.

[13] Intel Corporation. Intel C++ STM Compiler, Prototype Edition. April 2009. Retrieved 24 April 2012 from `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`.

[14] S. Lie. Hardware Support for Unbounded Transactional Memory. MS thesis, Massachussets Inst. of Technology, May 2004.

[15] P. McGachey et al. Concurrent GC leveraging transactional memory. In *Proceedings of the 13th AM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 217-226, February 2008.

[16] C.C. Minh. Stanford Transactional Applications for Multi-Processing. Standford U., 2008. Retrieved 13 April 2012 from `http://stamp.stanford.edu/`.

[17] R. Rajwar and J.R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, October 2002.

[18] C.J. Rossbach et al. TxLinux and MetaTM: transactional memory and the operating system. In *Communications of the ACM*, 51(9):83-91, September 2008.

[19] M.J. Schultz. Using Software Transactional Memory in Interrupt-Driven Systems. MS thesis, Marquette U., May 2009.

[20] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM symposium on principles of distributed computing*, pages 204-213, August 1995.

[21] S. Wang et al. DTM: Decoupled Hardware Transactional Memory to Support Unbounded Transaction and Operating System. In *Proceedings of the 39th international conference on Parallel processing*, pages 228-236, September 2009.

[22] R.M. Yoo et al. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures*, pages 265-274, 2008.

[23] ZFS-FUSE: ZFS for Linux. March 2011. Retrieved 16 April 2012 from `http://zfs-fuse.net/`.