Scaling Address-Space Operations on Linux with TSX

by

Christopher Ryan Johnson

B.A., Knox College (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2014

Certified by . . . .
M. Frans Kaashoek
Professor of Computer Science
Thesis Supervisor

Certified by . . . .
Nickolai Zeldovich
Associate Professor of Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Scaling Address-Space Operations on Linux with TSX

by

Christopher Ryan Johnson

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2014, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Concurrent programming is important due to increasing core counts, but scalable concurrency control is difficult and error-prone to implement. Hardware Transactional Memory (HTM) addresses this problem by providing hardware support for concurrently executing arbitrary read-modify-write memory transactions [9]. Intel released Transactional Synchronization eXtensions (TSX), a HTM implementation, in select processors to support scalable concurrency control [11].

This thesis contributes a case study in applying TSX to the Linux virtual memory system, which currently serializes address-space operations with a lock. TSX should provide scalability by supporting concurrent address-space operations. Achieving scalability with TSX, however, turned out to be difficult due to transactional aborts. This thesis details how to identify and resolve abort problems, and it describes the necessary modifications to make address-space operations scale in Linux.

This thesis also describes a new TLB shootdown algorithm, `TxShootDown`, which removes TLB shootdown from a transactional critical section while avoiding races due to concurrent address-space operations.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor of Computer Science

Acknowledgments

I would like to thank Frans Kaashoek and Nickolai Zeldovich for their excellent advice and guidance. I have thoroughly enjoyed growing as a researcher and as an engineer as a member of their group. Thank you to Fredrik Kjølstad for our long discussions about research and graduate school. Finally, I would like to thank my parents, Stephen and Linda; my sister, Melissa; and my wonderful fiancée, Elizabeth, for their constant support and encouragement through this thesis and everything else.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Concurrent programming is important due to increasing core counts in commodity computer hardware, but scalable concurrency control is difficult to achieve.

Coarse-grained locks provide isolation for multiple threads operating on shared resources, but they serialize operations, which limits scalability. Fine-grained locking and lock-free techniques provide isolation while improving scalability, but they are difficult to implement correctly [8].

Hardware Transactional Memory (HTM) was introduced to simplify concurrent programming by supporting customized, atomic read-modify-write *transactions* on arbitrary memory locations [9]. Hardware transactions define critical sections at coarse-granularity, like coarse-grained locks, but multiple transactions can execute the same critical section concurrently to provide scalability as long as they use different addresses. Transactions that use overlapping address in a conflicting way (read-/write or write/write) are aborted and serialized to preserve isolation; aborts can limit scalability if transactions frequently conflict.

Despite the potential benefits, a mainstream implementation of HTM did not exist until recently when Intel included Transactional Synchronization eXtensions (TSX) in select Haswell processors. TSX has the potential to benefit systems that depend on coarse-grained locking by replacing those locking critical sections with hardware transactions.

The Linux virtual memory system is one area where we expected TSX to be beneficial. Consider the case where multiple threads within a single process perform operations on different areas of a shared address space. The operations are commutative, thus unlikely to conflict, but a coarse-grained reader-writer lock serializes their execution. TSX can exploit the fact that accesses are disjoint to execute concurrent address-space operations in parallel.

This thesis contributes a case study of applying TSX to scale address-space operations in Linux. We first modified the Linux kernel to execute address-space operations transactionally with TSX before falling back to the reader-writer lock if the transaction aborts. The operations should have executed concurrently because the operations changed different parts of a shared address space, but virtually all transactions aborted and fell back to locking. The TSX hardware aborted and serialized *disjoint* address-space operations because the implementation of those operations violated isolation: either by conflicting with one another, overflowing transactional buffers in hardware, or executing instructions that are invalid within a transaction.

Identifying and resolving transactional aborts is a major challenge in using TSX. Performance monitoring tools and hardware do not provide information about the cause of *asynchronous* conflict aborts. The hardware records an asynchronous conflict abort at the location where it detects a conflict, but this is not necessarily the location that caused the abort. Intel does not reveal details about their abort algorithm, so conflict aborts are typically found through trial and error.

This thesis details the necessary changes to scale address-space operations in Linux using TSX.

One challenge with TSX is dealing with operations that must execute instructions that are invalid within a transaction (e.g. writing to a control register, flushing the TLB, performing I/O). These instructions always cause the transaction to abort. One solution to this problem is to delay these operations until after the transaction ends, but this can introduce race conditions when the operation must be executed in the critical section. This thesis proposes a new TLB shootdown algorithm, called `Tx-`

ShootDown, that provides consistency despite delaying TLB operations to outside of a transaction.

The changes described in this thesis are the result of iteratively identifying and resolving abort issues in the Linux virtual memory system. The major changes involved pre-allocating memory and removing non-scalable caches and counters from transactions.

The contributions of this thesis are: (1) a case study of scaling address-space operations in Linux using TSX and (2) a new TLB shootdown algorithm which provides consistency despite non-transactionally executing TLB flush operations.

The rest of this thesis is organized as follows: Chapter 2 describes related work. Chapter 3 describes the hardware interface provided by TSX and how to use it. Chapter 4 presents our methodology for identifying aborts. Chapter 5, Chapter 6, and Chapter 7 describe TxShootDown and the design changes required to achieve scalability in Linux. Finally, Chapter 8 summarizes our experience using TSX and concludes.

Chapter 2

Related Work

This thesis is the first detailed exploration of transactional abort problems in applying TSX to scale a real operating system service.

Recent work has explored scaling existing software using TSX and stressed the importance of addressing abort problems [14]. Odaira et al. investigated using TSX to eliminate the global interpreter lock in Ruby [16], which involved resolving several abort issues similar to ones we address in this thesis. This thesis builds on previous work by providing concrete examples of how to identify and resolve aborts in a real system.

Andreas Kleen at Intel has been working on a complete conversion of locking in the Linux kernel to use TSX to elide locks [12]. In the latest version of that kernel, TSX is disabled for several operations (such as `munmap`) due to abort issues that we address in this thesis. This thesis includes experiments run on the latest version of that kernel to provide a direct comparison between our implementation and the current state of the art.

Previous work observed that transactional aborts are a major source of overhead using TSX in several applications; including data structure implementation [15, 21], memory reclamation [1], and in-memory databases [20]. Previous work on other transactional memory systems also considered aborts a major concern [6, 2]. This thesis utilizes a debugging methodology based on `perf` to identify and resolve aborts in Linux's virtual memory system.

The virtual memory system in Linux tracks allocations using red-black trees, which prior work implemented using software transactional memory [22, 7]. Other prior work recommends using a scheme that delays re-balancing operations in red-black trees to reduce conflicts [2], but we find that these designs are not necessary to achieve good performance under TSX. In fact, we find that using TSX with red-black trees achieves high performance and scalability when the tree is large and writes are distant.

The TxLinux operating system explored using transactional memory with an architectural model called MetaTM [19, 18]. TxLinux uses "cooperative transactional spinlocks," which manage contention between transactional and non-transactional lock acquisitions. MetaTM also provides instructions to suspend transactions to perform I/O operations. TSX does not support either of these features, so different solutions are required for Intel TSX. This thesis proposes `TxShootDown`, which solves a problem performing TLB shootdowns under transactions without support for suspending transactions.

Chapter 3

Intel TSX

Intel's TSX is the first mainstream implementation of hardware transactional memory. This section describes the features provided by TSX, a new interface to use TSX in the Linux kernel, and the parameters and performance of a TSX-enabled Xeon processor.

3.1 Features

TSX follows a similar design to the original HTM proposal [9]. The processor tracks the set of read and written memory locations during transactional execution. A conflict exists if the write set of a transaction overlaps with the read or write set of another transaction. Read and write sets are tracked at cache-line granularity in the data cache in the current TSX implementation. Conflicts are detected by the cache-coherence protocol, and the core that detects the conflict must abort [11]. TSX provides strong isolation, which means that non-transactional operations that conflict with a transaction will cause that transaction to abort. The transactions provided by TSX are best-effort, which means that no transaction is guaranteed to complete (this may be due to micro-architectural issues). There are two feature sets provided by TSX: Restricted Transactional Memory and Hardware Lock Elision.

Restricted Transactional Memory The TSX interface to transactional memory is called Restricted Transactional Memory (RTM), and it consists of 4 new instructions: XBEGIN, XABORT, XTEST, and XEND. XBEGIN begins transactional execution by storing the current CPU register state and a relative address in shadow registers. The relative address given to XBEGIN is the address of the fallback handler. If the transaction aborts, the hardware will restore the original register state, jump to the fallback handler, and store an abort code in `EAX`. The XABORT instruction triggers a transaction abort. XABORT takes an immediate operand that the hardware includes in the abort code to support custom adaptive locking strategies. XTEST sets the zero flag to indicate if the processor is currently in a transaction. The XEND instruction commits the transaction by atomically removing all cache lines from the read and write sets and exiting transactional execution. RTM transactions require a *non-transactional fallback path*, because transactions are best-effort and may never succeed despite repeated retries. The non-transactional fallback path we implement acquires the lock, but related work proposes using Software Transactional Memory to implement the fallback path [5, 13]. We do not consider Software Transactional Memory in this thesis.

RTM Abort Codes The EAX register contains an abort code after an RTM transaction aborts. This code provides information about the abort so that the programmer can decide when to retry or when to use the non-transactional path. The abort codes for RTM are summarized in Table 3.1. The most important codes are RETRY, CONFLICT, and CAPACITY. RETRY and CONFLICT are frequently set together, which means that the transaction aborted due to a conflict with another memory operation, and the operation may commit if retried. It is recommended to restart a transaction when the RETRY flag is set. The CAPACITY flag indicates that the transaction did not fit in the cache because one of its cache lines was evicted. This can happen due to capacity or associativity issues. Transactions may also abort due to processor interrupts or exceptions, in which case no flags are set.

Hardware Lock Elision TSX has another transactional interface called Hardware Lock Elision (HLE), which is an implementation of speculative lock elision [17]. XAC-

EAX Bits	Name	Meaning
0	ABORT	Explicit abort with XABORT
1	RETRY	TX may succeed on a retry
2	CONFLICT	TX data conflict
3	CAPACITY	TX cache eviction
4	DEBUG	Debug breakpoint
5	NESTED	TX aborted while nested
23:6	Reserved	Reserved
31:24	XABORT CODE	Immediate from XABORT

Table 3.1: TSX abort codes in EAX

QUIRE and XRELEASE are instruction prefixes that modify instructions performing locking. An XACQUIRE prefixed lock acquisition on a TSX-enabled processor will begin transactional execution like an XBEGIN under RTM, and an XRELEASE prefixed unlock will behave like XEND. If the transaction aborts, execution falls back to lock acquisition, but the XACQUIRE prefix is ignored. Processors without TSX always ignore the XACQUIRE and XRELEASE prefixes, so locks are used normally. HLE provides a backwards-compatible interface to use transactions instead of locks on supporting hardware. This thesis proposes an RTM interface rather than HLE to experiment with various retry and backoff strategies.

3.2 Programmer Interface

In this section we propose a custom C interface to RTM for reader-writer semaphores (`rw_semaphore`) in the Linux kernel. `mmap_sem` is the reader-writer semaphore that serializes address-space modifications in Linux, so applying TSX to `mmap_sem` allows multiple threads to concurrently attempt address-space operations.

The custom C interface to RTM is based loosely on a design from Intel[1], but it has support for backoff and different retry strategies. It is based on low-level C implementations of the RTM instructions [12]. The function `xbegin()` returns the value of EAX on the abort path and a special value `_XBEGIN_STARTED` if the

[1] http://software.intel.com/en-us/blogs/2012/11/06/exploring-intel-transactional-synchronization-extensions-with-intel-software

transaction is started. The `xabort(char)` macro takes a `char` to paste into the XABORT instruction's immediate value. `xtest()` returns 1 if in a transaction, 0 otherwise, and `xend()` simply issues XEND.

The following C code shows the function that acquires a Linux kernel reader-writer semaphore in write mode. The code is nearly identical for locking in read mode.

```c
void tx_wlock(struct rw_semaphore *lock)
{
  int ret;
  int failures = 0;
retry:
  /* Use linear backoff on failure up to ALLOWED_FAILURES = 5 */
  if(failures && failures <= ALLOWED_FAILURES)
      backoff(failures);
  else if(failures > ALLOWED_FAILURES)
  {
      /* Actually lock semaphore if failure limit reached */
      down_write(lock);
      return;
  }

  /* wait_free_lock spins until the lock is free or 10K cycles have
     elapsed. It returns 0 if the lock is still locked. */
  if(!wait_free_lock(lock))
  {
      /* Actually lock semaphore if the lock is still not free*/
      down_write(lock);
      return;
  }

  ret = xbegin();
  if(ret == _XBEGIN_STARTED)
  {
      /* Ensure that lock is unlocked. This brings the lock into
         the transaction's read set. */
      if(tx_rwsem_is_locked(lock))
      {
          xabort(0);
      }
      return;
  }
  /* Retry as long as the transaction did not overflow */
  else if(!(ret & TXA_CAPACITY))
  {
      failures += 1;
      goto retry;
  }
  /* Actually lock semaphore if transaction overflowed */
  else
      down_write(lock);
}
```

The `tx_wlock` function provides isolation for threads that acquire the lock in write mode, and it prevents threads from acquiring the lock in write mode until all read-mode threads release the semaphore (similar to a regular reader-writer lock). `tx_wlock` differs from a regular reader-writer lock in that it first attempts to provide isolation using a TSX hardware transaction. Multiple kernel threads can then transactionally execute the critical section in parallel, which provides scalability as long as concurrent transactions do not conflict and can be stored in the transaction cache. Transactions that abort due to capacity or frequent conflicts acquire the semaphore in write mode, which excludes other threads and limits scalability.

`tx_wlock` provides a non-transactional fallback path for transactions that fail more than `ALLOWED_FAILURES` times or which abort due to a `CAPACITY` abort. `tx_wlock` retries transactions regardless of whether or not the RETRY flag is set in the abort code. This is because interrupts and other transient events cause aborts that do not set the RETRY flag; our experiments have shown that retrying anyway is beneficial. `tx_wlock` does not retry if the CAPACITY flag is set, since we assume that this is not a transient abort (the same transaction is likely to fail again since the set of memory locations it attempts to access will not fit in the cache). The exception to this is hyper-threading, where a process running on the other hyper-thread causes an L1 eviction that aborts the transaction. When the hyper-thread is not heavily using the cache (e.g. when a hyper-threaded transaction ends), the original transaction may commit. The experiments in this thesis run without hyper-threading to avoid this problem.

The non-transactional fallback path acquires the semaphore in write mode. `tx_wlock` provides isolation between the fallback path and transactional path by bringing the semaphore into the read set of the transaction. The transaction may proceed only if the lock is unlocked, otherwise the transaction aborts since it could observe partial results from the thread that locked the semaphore. Non-transactional threads acquiring the lock will cause transactions to abort because they write to the semaphore.

Threads call `wait_free_lock` to wait until the semaphore is unlocked before starting a transaction. This de-couples waiting for the semaphore and transaction aborts. Otherwise when one thread acquires the semaphore it causes later threads to all abort `ALLOWED_FAILURES` times, which results in a fallback to locking under heavy load. `wait_free_lock` is also important for debugging because it prevents frequent `XABORT` aborts from appearing in the debugging results.

`tx_wlock` calls `backoff`, which performs linear backoff after aborts. Intel's optimization manual states it is necessary to use some backoff strategy to prevent frequent aborts [10], but this thesis does not perform a comparison of backoff strategies.

3.3 Intel TSX Implementation

The experiments in this thesis run on a system with a TSX-enabled 4-core Intel Xeon E3-1270 v3 CPU running at 3.50GHz. Each core has private 32K 8-way set-associative L1 data and instruction caches, a 256K unified L2 cache, and the cores share an 8MB unified L3 cache. The system has 32GB of installed RAM.

Cache Size Wang et al. measured the maximum size of the read and write sets at 4MB and 31KB respectively [20]. Their experiment retried many times and reported the transaction success rate to deal with implementation-specific issues. A non-zero success rate for a certain size transaction means that the transaction cache is at least that size.

Certain features of TSX make it difficult to measure the maximum size of transactions: First, the TSX implementation seems to dynamically change its abort criteria based on a learning algorithm. Odaira et al. provide an experiment that shows TSX eagerly aborts transactions that suffered from repeated capacity aborts [16]. Second, the read set spills out of the L1 cache into a secondary structure [11], but details about this structure are unknown.

The L1 cache is shared between both hyper-threads on a core, which reduces the available transaction cache space when both hyper-threads are using the cache. This can cause frequent capacity aborts if both cores are executing transactions simultaneously. The experiments in this thesis avoid these issues by not using hyper-threading.

Performance We measured the performance of RTM and HLE using a small benchmark that measures the number of cycles for TSX operations using the serializing RDTSCP instruction. We wrapped each operation with RDTSCP to measure start and end times, and the results are average cycle counts over 2.5 million operations. Results are presented in Figure 3-1. All results exclude the measured 30 cycle RDTSCP overhead.

The overhead of simply beginning and ending a transaction (xbegin/xend) is 40 cycles, while the overhead of compare exchange (cmpxchg) and atomic increment operations are 17 and 16 cycles respectively. TSX should not be used for individual atomic increments and swaps due to this overhead, but multiple reads and writes may be performed within a transaction to amortize the upfront cost and provide atomicity for the entire operation (begin/readX and begin/writeX, where X is the number of cache lines, and xend is used to commit the transaction). HLE (xacquire/xrelease) has a slight overhead compared to RTM (52 cycles vs. 40 cycles), which is likely due to tracking elided locks [11]. An aborted transaction (xbegin/xabort, with the start time immediately before XBEGIN and the end time at the top of the abort handler)

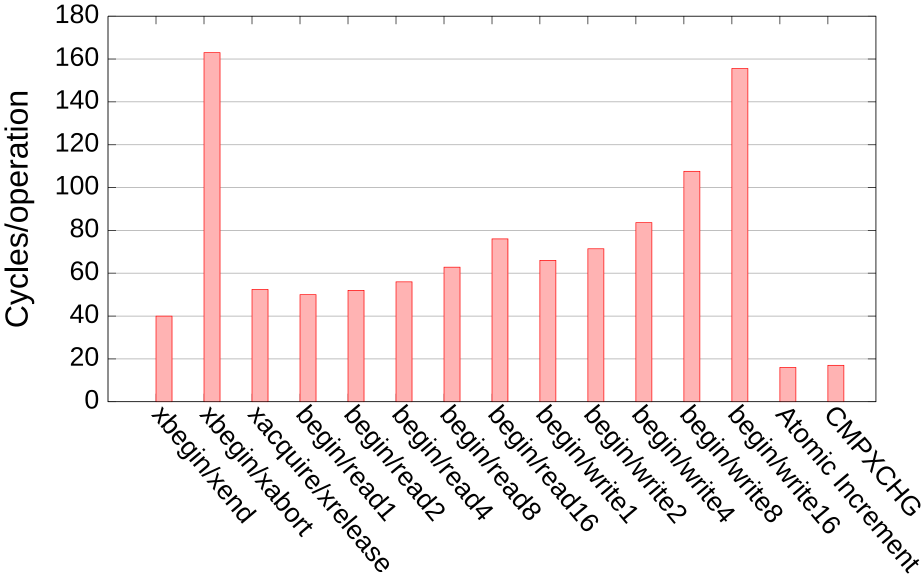

Figure 3-1: Microbenchmark of TSX instructions

has an overhead of 163 cycles, which is on top of the wasted work performed in the aborted transaction. This overhead is likely due to restoring saved register state and jumping to the abort handler.

Chapter 4

Case Study

4.1 Overview

This thesis presents a case study in using Intel TSX in the Linux virtual memory system. The Linux kernel we use is a patched 3.10 kernel with TSX profiling support in the `perf` utility [12].[1] This section describes the motivation for investigating the virtual memory system, the benchmark that we use to evaluate scalability, the debugging methodology we follow to identify and address transactional aborts, and a summary of the modifications we made to Linux.

4.1.1 Investigating Linux Virtual Memory

The Linux virtual memory system relies on a per-address-space reader-writer semaphore called `mmap_sem`. The `mmap` and `munmap` address-space operations acquire `mmap_sem` in write mode, and the page fault handler acquires `mmap_sem` in read mode so that page faults can be handled concurrently. `mmap_sem` is a coarse-grained lock which serializes address-space operations, so we wrap `mmap_sem` acquisition with the TSX interface described in Chapter 3. TSX should then execute operations on disjoint virtual memory locations in parallel.

[1] Branch `hsw/pmu7`

4.1.2 Benchmark

The case study uses the “local” benchmark from RadixVM [4]: Each thread in the experiment `mmap`s a page of virtual memory, touches that page (causing a page fault), and then `munmap`s that page in each iteration. Each thread operates on a different page. These operations touch disjoint pages and should not need to share data, which means that the operations should work well with TSX.

Conflicts in internal data structures and page tables can cause transactional aborts even though the operations do not need to share data. The benchmark pads out internal data structures by including 16MB gaps between thread pages filled with 4MB of non-contiguous single-page mappings. This modified benchmark should be easy to scale: the operations touch disjoint pages and necessary sharing is minimal. Operations are serialized for this benchmark only because of the `mmap_sem` semaphore.

All experiments in this thesis measure average iterations per second over a 5 second run of the benchmark with 2 seconds of warmup.

This thesis does not include a comparison with an application workload because its operations may result in actual data sharing. Actual sharing increases the abort rate and hides abort problems that limit scalability due to implementation details.

4.1.3 Early Results

We started by wrapping the acquisition of `mmap_sem` using the interface described in Chapter 3. Results for this experiment appear in Figure 4-1.

The non-transactional kernel (Original) fails to scale to even 2 cores because all operations acquire `mmap_sem` in write mode. Performance actually collapses from around 850,000 to around 250,000 iterations per second, which is consistent with prior results reported in RadixVM [4]. Figure 4-1 also shows the Lock Elision implementation mentioned in Chapter 2, which wraps lock acquisition and release with TSX throughout the kernel [12].[2] The Lock Elision kernel's performance is similar

[2] We used the latest commit to branch `hle313/combined` on 2014-03-06.

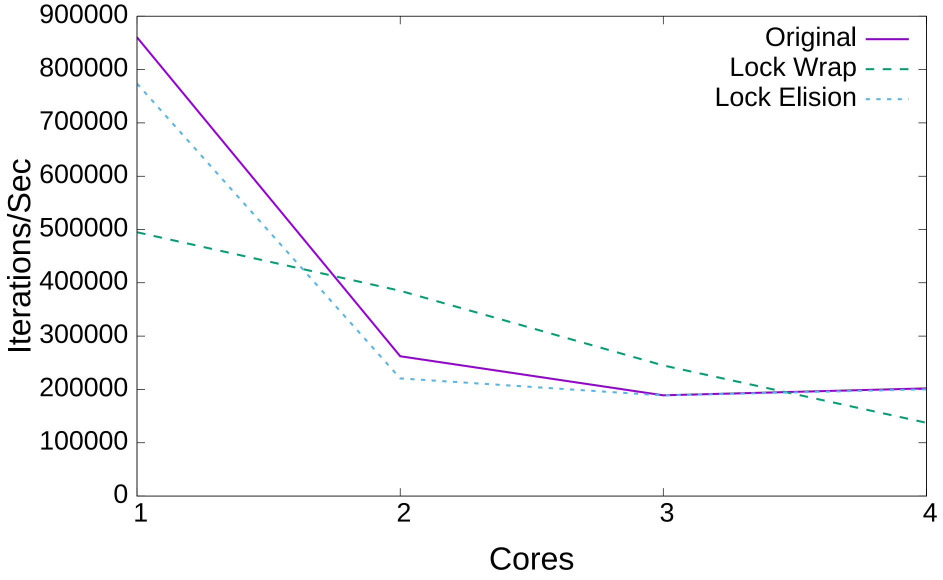

Figure 4-1: Comparison of simple approaches to using TSX in Linux

to the unmodified kernel. This is because the Lock Elision kernel disables TSX for virtual memory operations to avoid aborts. We address these aborts in this thesis.

The performance of wrapping `mmap_sem` with TSX (Lock Wrap), however, is worse than an unmodified kernel. We expected Lock Wrap to scale because the benchmark operates on different parts of the virtual memory space. The operations do not depend on each other or need to share data, so TSX should have executed address-space operations concurrently.

The `perf` performance analyzer showed that a large proportion of cycles per iteration are spent in aborted transactions and waiting for `mmap_sem` to be free, which is the cause of the performance problems in the Lock Wrap kernel. Table 4.1 shows the percentage of transactions that abort for the Lock Wrap approach.

Cores	Abort Rate
1	85 %
2	90 %
3	99 %
4	99 %

Table 4.1: Abort rates for Lock Wrap kernel

Very few transactions actually run to completion on one core, and virtually no transactions complete at higher core counts. Frequent transactional aborts cause operations to serialize on the lock, limiting scalability.

4.2 Identifying Aborts using `perf`

We resolved the scalability problems with our initial results by identifying and resolving issues that caused transactional aborts. `perf` provides "record" functionality, which records a specific event (aborts, in this case).[3] `perf`'s "report" mode displays a list of functions and abort conditions (capacity, conflict, etc.) sorted by the proportion of all aborts represented by that `(function, abort condition)` pair. `perf` also records the instruction pointer (IP) at the event. `perf report` can interactively display disassembly and source annotations along with per-instruction abort information. An example of the output from `perf report` appears in Figure 4-2.

```
Run: perf report --sort=symbol,transaction
Samples: 59K of event 'cpu/tx-aborts/pp',
Event count (approx.): 2391430
37.16%  [k] find_vma          TX ASYNC RETRY CON
11.67%  [k] mmap_region       TX ASYNC RETRY CON
 9.51%  [k] do_munmap         TX ASYNC RETRY CON
 8.05%  [k] vma_compute_s...  TX ASYNC RETRY CON
 4.25%  [k] kmem_cache_al...  TX ASYNC RETRY CON
 3.75%  [k] _raw_spin_lock_irqsave       TX SYNC
```

Figure 4-2: Example `perf` output at 4 cores

4.2.1 Synchronous Aborts

Operations that cannot be done inside transactions, such as TLB shootdowns, are easy to identify in the assembly dump output. They appear as a single instruction with a large amount of aborts and a *synchronous* abort condition. The *synchronous*

[3] The specific command we run is `perf record -a -g -transaction -e cpu/tx-aborts/pp ./mapbench CORES local`, where `CORES` is the number of threads created. This records all abort events on all cores (`-a`) with stack traces (`-g`) and abort flags (`-transaction`).

flag means that the recorded instruction is the actual cause of the abort, so the reason for the abort is clear.

Running the benchmark with a single thread makes identifying synchronous aborts easier because it eliminates asynchronous aborts. For example, the output appearing in Figure 4-3 was generated with a single core and the output in Figure 4-2 was generated with 4 cores. Running on a single-core avoids conflict aborts, so only a function containing a `cli` instruction (which is not allowed in a transaction) appears in the list. Figure 4-3 includes the interactive disassembly output showing the abort. We fixed this problem by simply not suspending or restoring interrupts in transactions.

```
Run: perf report --sort=symbol,transaction
Samples: 27K of event 'cpu/tx-aborts/pp',
Event count (approx.): 19953364
100.00%  [k] _raw_spin_lock_irqsave  TX SYNC

Disassembly:
static inline void native_irq_disable(void)
              {
                        asm volatile("cli"
                                   :::"memory");
100.00       cli
```

Figure 4-3: Example `perf` output at 1 core

4.2.2 Asynchronous Abort Identification

Conflict abort events are *asynchronous*. The IP in the `perf` output corresponds to the instruction being executed when a processor happened to detect the conflict, but this is not necessarily the instruction causing the conflict. This occurs when a core that previously accessed a cache line detects the conflicting access first. All that we know in this case is that a conflicting memory access occurred before the recorded instruction. We call the two conflicting memory operations that caused an abort the *offending instructions*. The benchmark runs for many iterations, and the areas near offending instructions often show up in the top abort results.

The current TSX implementation aborts the first transaction detecting a conflict at an arbitrary point, which is not necessarily an offending instruction [10]. The manual does not justify this design, but it is likely due to the performance overhead of synchronously checking for remote transactions that conflict with each access. Synchronous reporting for conflict aborts would simplify debugging, but the performance overhead makes it expensive to implement in practice. Intel does not provide any guarantees about which transaction will detect the conflict first and abort; this information is omitted from the specification [11].

The offending instructions causing false conflicts (improper padding or counters on the same cache line) are particularly difficult to track down due to asynchronous abort information. The offending instruction on one core may not detect the conflict; another core, which had already proceeded past its offending instruction, detects the conflict and triggers an abort. This process repeats with different abort points until there are many recorded conflict aborts spread throughout the program rather than concentrated near the offending instruction. Figure 4-4 shows what the `perf` output looks like in this case. At first it looks like there is a data conflict in the tree traversal operation in the disassembly, but the real problem is a conflict on a cache line accessed earlier in the transaction. The most common abort events appear in locations that have little to do with the actual problem, which we consider *noise* in the debugging output.

Disabling the retry policy helps to reduce noise by reducing the number of repeated aborts. Threads that keep retrying can repeatedly abort each other, resulting in the output in Figure 4-4. Figure 4-5 shows the result of setting the number of allowed retries to 0. Aborts are spread more evenly between the functions, but the second-highest aborting instruction in `mmap_region` (the highest aborting function with 5 retries) is an offending memory read of `mm->total_vm` (the offending write is found later in the `perf` output in `vm_stat_account`).

Preliminary experiments show some promise in artificially extending transactional execution using a no-op loop to increase the chance of finding conflicting aborts. For example, Figure 4-7 shows the result of inserting the no-op loop in Figure 4-6 before

Chapter 5	TLB shootdown: delay and target shootdown
Chapter 5.3	`TxShootDown`: solve race condition
Chapter 6	Data conflicts: counters, caches, locks, padding
Chapter 7	Memory allocation: per-core pools, delay free

Table 4.2: Summary of Linux modifications

the tree traversal in Figure 4-4. This output uncovers the offending memory write to `mm->total_vm` in `vm_stat_account`. At the time of writing, however, this technique is not well understood. We leave exploration of this technique to future work.

4.2.3 Result Summary

Using `perf`, we iteratively identified and removed abort problems from Linux's virtual memory system. A summary of our changes appears in Table 4.2. The results for all changes combined appear in Figure 4-8. The benchmark scales up to 4 cores, which is the maximum number of cores that avoids L1 cache sharing.

The modifications result in a 14% performance decrease from the Original kernel at 1 core. This is the cumulative effect of changes that included removing several performance optimizations and caches.

`TxShootDown` is an algorithm that uses a three-phase strategy to correct a race condition in TLB shootdown operations. Figure 4-9 demonstrates that using `TxShootDown` to solve the TLB shootdown problem does not negatively impact scalability.

In the next three chapters, we detail the changes that were required to scale address-space operations in the Linux kernel.

```
Run: perf report --sort=symbol,transaction
Samples: 109K of event 'cpu/tx-aborts/pp',
Event count (approx.): 32187883
13.86%  [k] mmap_region       TX ASYNC RETRY CON
10.60%  [k] zap_pte_range     TX ASYNC RETRY CON
 6.96%  [k] unlink_anon_vmas  TX ASYNC RETRY CON
 6.17%  [k] alloc_vma         TX ASYNC RETRY CON
 6.03%  [k] unmap_single_vma  TX ASYNC RETRY CON

Disassembly of mmap_region:
  5.42  a8:   cmp    %r15,-0x20(%r8)
  1.05      âĘ§ jb     170
        return -ENOMEM;
        __rb_link = &__rb_parent->rb_left;
  0.02        lea    0x10(%r8),%rbx
  2.44  b6:   mov    %r8,%r13
    } else {
        rb_prev = __rb_parent;
        __rb_link = &__rb_parent->rb_right;
  3.31        mov    (%rbx),%r8
        struct rb_node **__rb_link, *__rb_parent,
            *rb_prev;
        __rb_link = &mm->mm_rb.rb_node;
        rb_prev = __rb_parent = NULL;
        while (*__rb_link) {
  5.36  bc:   test   %r8,%r8
  0.22      âĘ§ je     d0
            struct vm_area_struct *vma_tmp;
            __rb_parent = *__rb_link;
            vma_tmp = rb_entry(__rb_parent,
                struct vm_area_struct, vm_rb);
            if (vma_tmp->vm_end > addr) {
 20.64       cmp    -0x18(%r8),%r12
 22.02      âĘŠ jb     a8
```

Figure 4-4: Example `perf` output showing spread out aborts

```
Run: perf report --sort=symbol,transaction
Samples: 85K of event 'cpu/tx-aborts/pp',
Event count (approx.): 3771331
 6.91%  [k] unlink_anon_vmas                   TX ASYNC RETRY CON
 5.92%  [k] zap_pte_range                      TX ASYNC RETRY CON
 5.48%  [k] tx_wunlock                         TX ASYNC RETRY CON
 5.41%  [k] mmap_region                        TX ASYNC RETRY CON
 5.30%  [k] do_munmap                          TX ASYNC RETRY CON

Disassembly of mmap_region:
                cur = mm->total_vm;
                    lim = rlimit(RLIMIT_AS) >> PAGE_SHIFT;
                    if (cur + npages > lim)
              mov    %rdx,%rsi
 11.16        add    0xa8(%r10),%rsi
```

Figure 4-5: Example `perf` output after disabling transaction retries

```c
#define NOP_TIME 7000
noinline void nop_loop(void)
{
    register int i;
    for(i=0; i<NOP_TIME; i++)
        asm volatile ("nop":::"memory");
}
```

Figure 4-6: Example no-op loop

```
Run: perf report --sort=symbol,transaction
Samples: 93K of event 'cpu/tx-aborts/pp',
Event count (approx.): 6646981
60.94%  [k] nop_loop          TX ASYNC RETRY CON
12.75%  [k] tx_wunlock        TX ASYNC RETRY CON
 2.94%  [k] tx_wlock          TX SYNC :1
 2.76%  [k] mmap_region       TX ASYNC RETRY CON
 2.44%  [k] tx_wlock          TX ASYNC RETRY CON
 1.50%  [k] do_mmap_pgoff     TX ASYNC RETRY CON
 0.93%  [k] vm_stat_account   TX ASYNC RETRY CON

Disassembly of vm_stat_account:
           mm->total_vm += pages;
 98.47 add    %rcx,0xa8(%rdi)
           if (file) {
       test   %rdx,%rdx
```

Figure 4-7: Example `perf` output using a no-op loop

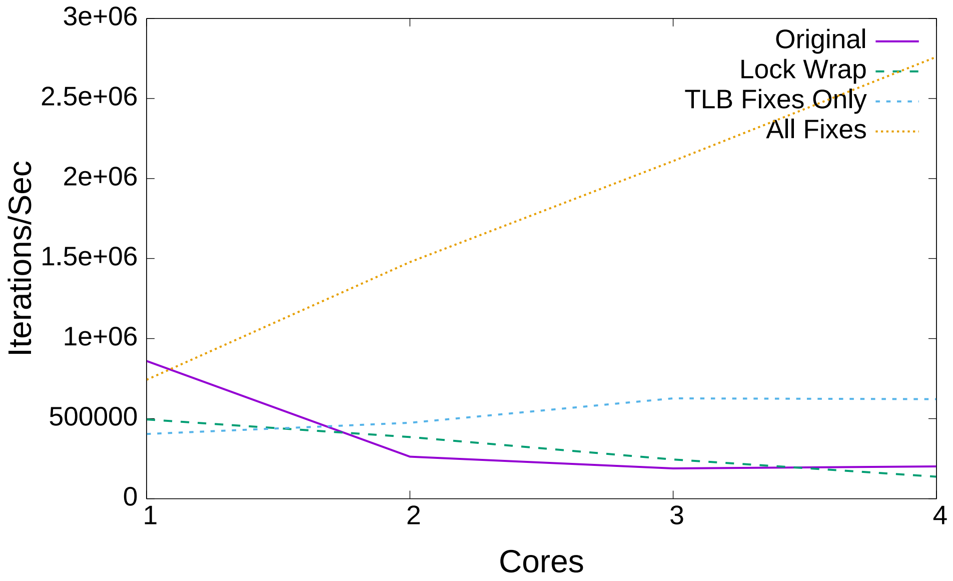

Figure 4-8: Performance of all changes

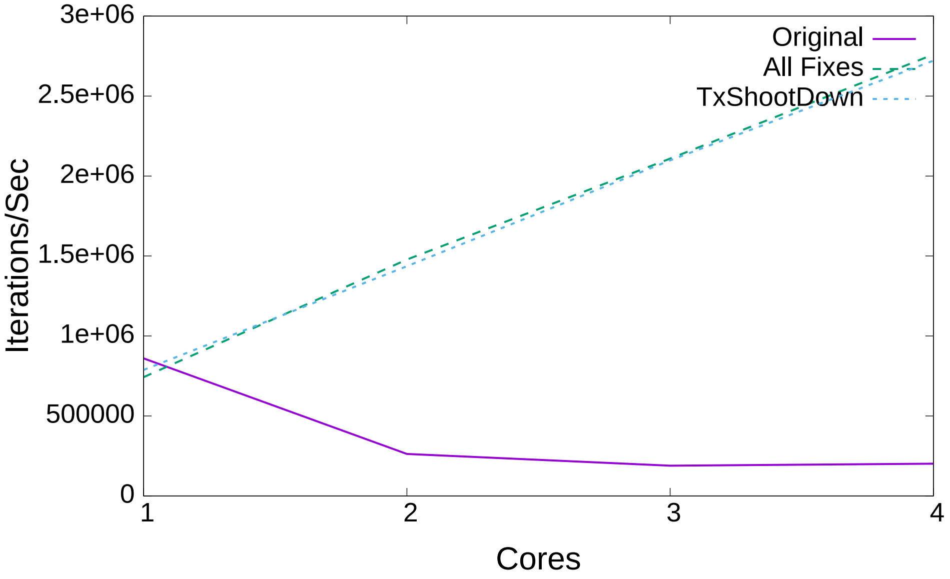

Figure 4-9: Results for `TxShootDown` compared to our modified kernel

Chapter 5

Delay TLB Shootdown

The translation look-aside buffer (TLB) caches the results of page directory walks in hardware on the X86. When a thread unmaps a region of virtual memory, the kernel must ensure that all cores flush any TLB entries related to that region so that future accesses in that region do not see the old page.

5.1 TLB Shootdown Aborts

Linux implements TLB shootdown using the following process: The core which unmaps a Virtual Memory Area (VMA) will invalidate its own TLB and queue a TLB invalidate function to be run on each other core. It then triggers an inter-processor interrupt (IPI) on those cores while still holding `mmap_sem`. The interrupt handler on those cores runs the queued TLB invalidate, and the original core continues (releasing `mmap_sem`) once all other cores finish the TLB invalidate.

TSX transactions abort due to TLB invalidations and IPIs, which we discovered using the output in Figure 5-1. The `invlpg` instruction flushes the local TLB for a specific page, but that instruction is invalid within a TSX transaction.

```
Run: perf report --sort=symbol,transaction
Samples: 27K of event ’cpu/tx-aborts/pp’,
Event count (approx.): 12789619
99.99%  [k] flush_tlb_mm_range   TX SYNC

Disassembly:
static inline void
__native_flush_tlb_single(unsigned long addr)
    {
        asm volatile("invlpg (%0)"::"r" (addr):
          "memory");
100.00  180:   invlpg (%rax)
```

Figure 5-1: Output of `perf` showing TLB invalidation abort

5.2 Race Condition

One possible solution is to delay TLB shootdown operations until after the transaction is committed, but this solution is not correct in general due to a race condition (as shown in Figure 5-2). If an in-progress `munmap` completed unmapping an area and is beginning its TLB shootdown outside the transaction, a concurrent `mmap` in the same area may complete before the TLB shootdown is completed. The core completing the `mmap` can notify a third core, which is caching the old mapping, to read the new mapping. The notified core should see the new mapping, but because the TLB shootdown is incomplete it will see the cached old mapping instead. This violates the semantics of `mmap` and `munmap`.

5.3 `TxShootDown`

This section describes `TxShootDown`, a new TLB shootdown process. `TxShootDown` solves the TLB shootdown problem using a three-phase strategy for `mmap` and `munmap` to properly order TLB shootdowns, but it still allows concurrency for operations on non-overlapping regions.

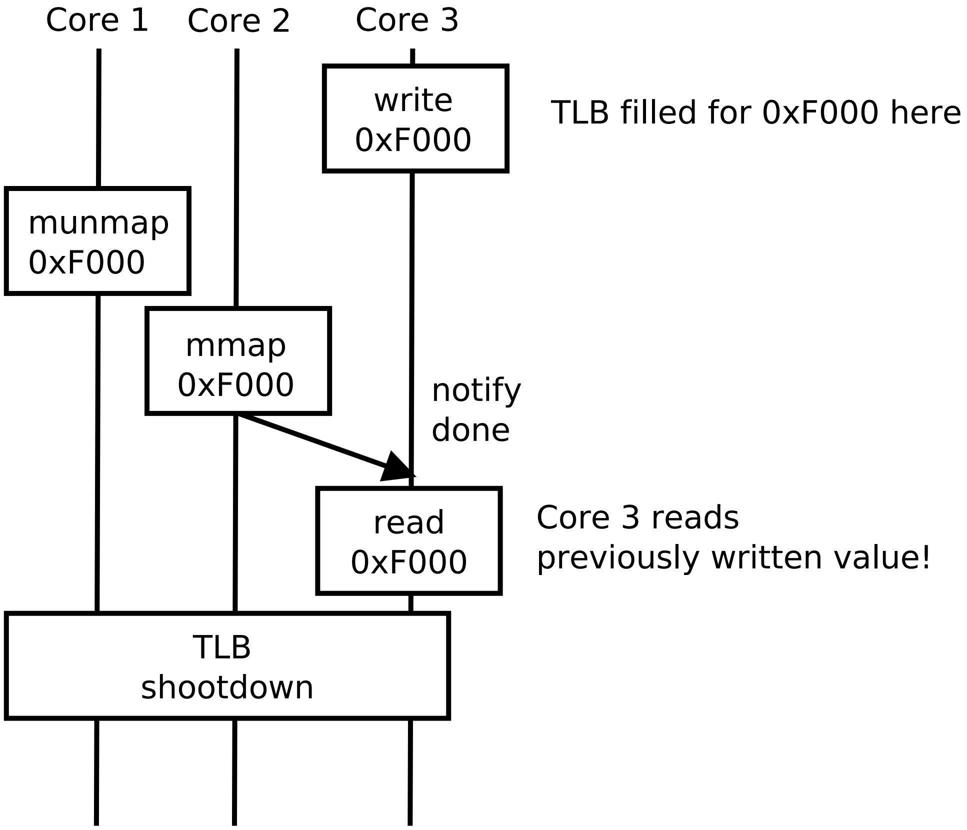

Figure 5-2: Race condition that occurs if TLB shootdown is not completed while holding `mmap_sem`

5.3.1 Algorithm

At a high level, TxShootDown ensures that concurrent operations in a memory region overlapping a non-transactional TLB shootdown are delayed until after the TLB shootdown is complete. TxShootDown is a three-phase strategy in which an operation completing the first phase has exclusive access to an area of the virtual memory space until after the third phase. TLB shootdown is performed in the second phase, which does not need to be executed in a transaction.

To enforce exclusive-access, TxShootDown augments each VMA node with a lock variable and enforces the following rule: Any operation (mmap, munmap, or page fault) that depends on a locked node must abort and retry.

The algorithm for munmap (which requires a TLB shootdown every time it successfully unmaps a region) in TxShootDown appears in Figure 5-3. Phase 1 locks all nodes in the region, and Phase 3 removes the locked nodes. The TLB shootdown occurs in Phase 2 while the region is locked. This ensures that any concurrent operation in that region is ordered after the TLB shootdown is completed. The locked nodes exactly represent the intersection of previously existing mappings and the region being removed, allowing operations in different regions to proceed while the TLB shootdown occurs. Node removal and freeing require two temporary lists, since removal of a value in a red-black tree may unlink a successor node rather than the node storing that value itself. List t2 contains the list of nodes whose values must be removed, and t3 contains the list of nodes which were unlinked and need to be freed.

The algorithm for mmap appears in Figure 5-4; it is similar to munmap with some modifications. Phase 1 in mmap combines the resizing from Phase 1 and the removal from Phase 3 in munmap. mmap also inserts and locks a node that represents the new region. Operations on the inserted region are prevented from completing until the inserted node is unlocked in Phase 3, which occurs after the TLB shootdown for unmapped overlapping regions in Phase 2. The TLB shootdown operation in mmap runs only if a region was removed in the first phase, preventing spurious shootdowns.

`munmap(start, len)`:
Temporary lists `t1, t2, t3`

Phase 1:

1. Begin TX
2. Collect nodes overlapping `[start, start+len)` in `t1`
3. Restart if any node in `t1` is locked
4. Modify, split, and lock nodes from `t1` such that the final set of locked nodes exactly represents the intersection of all ranges in `t1` with `[start, start+len)`; store locked nodes in `t2`.
5. Remove page table entries overlapping region
6. End TX

Phase 2:

1. If `t1` is not empty, perform TLB shootdown on union of cores tracked by nodes in `t1`

Phase 3:

1. Begin TX
2. Remove all regions in `t2`, store unlinked nodes in `t3`.
3. End TX
4. Free all nodes in `t3`

Figure 5-3: `munmap` algorithm

`mmap(start, len)`:
Temporary lists `t1, t2`

Phase 1:

1. Begin TX
2. Collect nodes overlapping `[start, start+len)` in `t1`
3. Restart if any node in `t1` is locked
4. Remove or modify nodes in `t1` so that none overlap `[start, start+len)`; store unlinked nodes in `t2`
5. Insert new node `[start, len)`, called `ins`, and lock it
6. Remove page table entries overlapping region
7. End TX

Phase 2 (outside TX):

1. If `t1` is not empty, perform TLB shootdown on union of cores tracked by nodes in `t1`

Phase 3 (in TX):

1. Begin TX
2. Unlock `ins`
3. End TX
4. Free nodes in `t2`

Figure 5-4: `mmap` algorithm

The TxShootDown page fault handler simply finds the node containing the location of the fault, retrying if the node is locked. When the page fault handler finds an unlocked node it populates the page table using an implementation copied from Linux. Waiting for unlocked nodes in page faults is necessary to prevent a core with a TLB entry for the page and a core which faulted during a concurrent mmap from seeing inconsistent values at the same virtual memory address.

TxShootDown's three-phase algorithm solves the race condition in Figure 5-2. The mmap on Core 2 cannot complete until the VMA node representing 0xF000 is unlocked, but that only happens after the TLB shootdown operation on Core 1 completes. As a result, Core 2 will notify Core 3 to read 0xF000 only after Core 3's TLB is cleared for that address.

5.3.2 Implementation

Integrating TxShootDown into the stock Linux virtual memory system would be a highly invasive change. We opted to implement TxShootDown as an alternative address-space within Linux to validate its design.

TxShootDown sits beside the original Linux virtual memory system. We modified the kernel to dispatch address-space operations based on address. TxShootDown handles operations within a reserved region, and Linux handles the rest. TxShootDown handles only anonymous memory mappings private to a single process.

TLB shootdowns target only those cores mapping the page using a scheme suggested in RadixVM [4]. Targeted TLB shootdowns are important to achieve high performance, as demonstrated by RadixVM, so we target TLB shootdowns in both the Linux virtual memory system and TxShootDown by tracking page faults for each VMA node. TLBs are hardware-filled on X86, so RadixVM uses per-core page tables to catch all faulting cores. We omit this implementation because it is unnecessary for the benchmark considered in this thesis.

Chapter 6

Removing Data Conflicts

A data conflict occurs when a thread writes to a cache line that is being tracked by a transaction on another thread. This violates isolation and causes a conflict abort. The benchmark considered in this thesis should not need to share data, but the debugging output contains frequent conflict aborts. We investigated the reason for these conflicts and discovered that they were caused by shared caches, global counters, locking in transactions, and data structure implementation issues. This section details the conflict problems and how we resolved them.

6.1 Shared Caches and Counters

The top bottleneck after solving TLB shootdown was a shared `mmap_cache`, which the kernel used to speed up repeated `find_vma` operations; writes to this variable frequently appeared in debugging output. `find_vma` is used to look up a VMA corresponding to a given virtual memory address, returning the relevant `vm_area_struct` and caching the result in `mmap_cache`. The `mmap_cache` causes frequent transactional aborts, since every operation on the address space reads and likely modifies the variable. We simply removed the cache to prevent a conflict. The result of this was a small reduction in single-core performance for our benchmark, but it reduced the abort rate with more than one core.

Shared counters were also a problem for scalability, since they are read and written by multiple cores. We simply removed shared counters that caused frequent aborts in the benchmark. The main offenders were: (1) The `map_count` counter, which keeps track of the number of VMAs in the address space, (2) The RSS counters, which keep track of how many pages are used in the address space, and (3) `/proc` file-system counters used for reporting virtual memory information. We removed `map_count` and the sanity checks that depended on it. The RSS counters were lazily synced from per-core counters already, so we decreased the frequency of synchronization from once every 64 updates to once every 4096 updates, and we moved synchronization outside of transactions. These problems were relatively easy to find with `perf`, because the memory write instructions appeared in the top abort locations.

The `/proc` file system reads from per-address-space counters that track virtual memory usage. `mmap` reads one of these counters early in a transaction, causing noisy asynchronous aborts when a remote thread writes to that counter. The output in Figure 4-5 shows the shared counter read, but the update function `vm_stat_account` (further down the `perf` output) contains the conflicting write. We simply disable writing to the counters (`total_vm`, `shared_vm`, `exec_vm`, and `stack_vm`) in `vm_stat_account` to prevent this abort.

Each core also keeps least-recently-used (LRU) lists of pages for that core. The function `lru_add_drain` is called to drain these pages to a shared LRU list, which causes transaction conflicts. We made this operation lazy, draining the LRU list only every 4096 times it originally was supposed to drain, and we moved drain operations outside of transactions where possible.

Finally, several data structures required alignment and padding to fit exactly within 64-byte cache lines. Padding reduces false sharing between different structures packed on the same cache line as recommended in the manual [10]. The structs that we padded are listed in Table 6.1.

Name	Description
pagevec	Represents a vector of pages
vm_area_struct	VMA node in VMA tree and lists
per_cpu_pageset	Set of pages for page allocator
anon_vma	Reverse map pages to VMAs
anon_vma_chain	Link anon_vmas and VMAs

Table 6.1: Linux structs that required padding

6.2 Locking

Linux acquires spinlocks while holding mmap_sem, so the transaction started by tx_wlock attempts to acquire them as well. This requires writing to the lock, so multiple transactions taking the same lock will attempt to put the lock into their write sets and abort due to write conflicts. Acquiring a lock in a transaction is unnecessary, since transactions already provide isolation.

Two locks that are acquired while holding mmap_sem are the global mmlist_lock, which protects lists of swapped address spaces during fork(), and the per-address-space page_table_lock, which protects the page tables of a process. Conflicts on these locks caused a large number of aborts.

We created new functions, spin_lock_tsx and spin_unlock_tsx, which elide lock acquisition and release under transactions. They behave normally when used outside of a transaction. Inside a transaction, spin_lock_tsx checks that the lock is free before continuing, and the transaction aborts if the lock is not free. spin_lock_tsx brings the lock into the transaction's read set to provide isolation between the transaction and non-transactional threads taking the lock. spin_unlock_tsx does nothing when executed in a transaction. These functions prevent write conflicts between transactions that conflict only on the spinlock while providing isolation between transactions and non-transactional threads holding the lock.

In general, converting fine-grained locking code to use TSX requires removing fine-grained locks and using a single coarse-grained lock. This not only makes it easier to reason about correctness, but it can also result in better performance by preventing unnecessary cache invalidations.

6.3 Data Structure Choice

The central data structure for the virtual memory system, an augmented red-black tree storing VMAs, does not work well with transactional memory. Each node in the VMA tree is augmented with its "subtree gap," the largest free memory gap in bytes to its left. `mmap` uses this field to find a suitable location for `mmaps` of a specific size. Modifications to the tree propagate updates to this field up the tree. This causes many conflict aborts since these updates frequently conflict near the root of the tree.

We find, however, that red-black trees in general can achieve high performance and concurrency. This is surprising because regular red-black trees must maintain balance properties that also require traversing the tree, but they do not necessarily need to traverse to the root.

We removed the subtree gap field entirely and changed the tree to a regular red-black tree. Non-fixed `mmap` operations that need to find an empty region to map will need to search more nodes, but this is preferable to aborts.

Chapter 7

Avoiding Memory Allocation

We observed frequent synchronous capacity aborts in memory allocation and initialization functions, and we observed conflict and capacity aborts in functions that free memory. Memory allocation, initialization, and freeing are expensive operations with a large cache footprint, which causes transactional aborts.

Linux's virtual memory system allocates zeroed 4KB pages of memory for page tables and anonymous memory mappings. Initializing one of these pages within a transaction fills up an eighth of the L1 cache on the test hardware, which causes frequent transaction aborts. We modified the kernel to create per-core pools of zeroed pages for use in virtual memory operations. Each per-core pool is refilled to capacity when empty, and we use pools with 2^{16} pages to prevent frequent refilling. Performing allocation and initialization outside of the transaction avoids aborts due to capacity issues.

`vm_area_struct` allocation has a similar issue. Linux allocates a `vm_area_struct` when it inserts a new region into the VMA tree in `mmap`, but this allocation causes capacity aborts. We modified the kernel to use per-core caches of pre-allocated `vm_area_struct`s. The capacity of each per-core cache is 2048, chosen to balance memory footprint and filling frequency. When Linux frees a `vm_area_struct` it goes back into the current core's cache to further reduce the frequency of refills.

Transactions that free memory cause conflict and capacity aborts due to sharing and the large cache footprint of de-allocating many objects. We modified the kernel to record memory frees inside the transaction in per-core buffers. The kernel actually performs buffered frees after the transaction ends.

Chapter 8

Conclusion

TSX is a powerful new tool for implementing concurrent systems, but programmers must design their systems carefully. Writing systems software using transactions requires special attention to avoid potential abort problems. This thesis provided a case study of applying TSX to the Linux virtual memory system, which turned out to be non-trivial because it is difficult to identify and resolve transactional aborts.

Operations that cause aborts, such as TLB shootdowns, must be delayed to outside of the transaction, but Chapter 5 shows that this may require secondary concurrency control to avoid race conditions. We implemented `TxShootDown`, which solves a race that resulted from moving TLB invalidations outside of a critical section.

Expensive operations within transactions, such as memory allocation, must be moved outside of the transaction. Per-core caches and pools of pre-allocated objects are a solution to avoid contention and initialization within transactions.

Unnecessary sharing through shared caches, global counters, fine-grained locking, and data structure modification caused conflicts that limited scalability in the benchmark. We removed sources of unnecessary sharing that prevented the benchmark from scaling.

Programmers must iteratively identify and resolve transactional aborts to scale large systems. `perf` provides access to hardware event counters, but asynchronous aborts introduce noise into the debugging output, which makes it difficult to identify the reason for an abort. We disabled retrying transactions to reduce noise in the debugging output, but this is only a partial solution. An interesting area for future work is designing a reliable mechanism to identify the instruction that is the source of an asynchronous abort.

Red-black trees, somewhat surprisingly, were easy to scale with TSX. Previous work proposed complicated schemes to scale red-black trees [8, 3, 2], but we use standard red-black trees under TSX without any changes beyond cache-aligning nodes in memory. Simply wrapping red-black tree operations in a transaction seems sufficient to provide good scalability in large trees. As transactional memory becomes more common, red-black trees may be useful as a simple concurrent data structure.

Bibliography

[1] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the 9th ACM European Conference on Computer Systems*. ACM, 2014.

[2] Lucia Ballard. Conflict avoidance: Data structures in transactional memory. *Brown University Undergraduate Thesis*, 2006.

[3] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45, pages 257–268. ACM, 2010.

[4] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 211–224. ACM, 2013.

[5] Peter Damron, Alexandra Fedorova, and Yossi Lev. Hybrid transactional memory. *ACM Sigplan Notices*, pages 336–346, 2006.

[6] D. Dice, Y. Lev, V.J. Marathe, M. Moir, D. Nussbaum, and M. Oleszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 325–334. ACM, 2010.

[7] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *ACM Sigplan Notices*, volume 44, pages 155–165. ACM, 2009.

[8] K. Fraser. *Practical lock-freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[9] M. Herlihy, J. Eliot, and B. Moss. Transactional Memory: Architectural Support For Lock-free Data Structures. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Number 248966-029. March 2014.

[11] Intel Corporation. *Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual.* Number 325462-050US. February 2014.

[12] Andi Kleen. https://git.kernel.org/cgit/linux/kernel/git/ak/linux-misc.git. Accessed: 4/25/2014.

[13] Andreas Kleen. TSX fallback paths, June 2013. https://software.intel.com/en-us/blogs/2013/06/23/tsx-fallback-paths.

[14] Andreas Kleen. Scaling existing lock-based applications with lock elision. *Queue*, 12(1):20, 2014.

[15] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the 9th ACM European Conference on Computer Systems*. ACM, 2014.

[16] Rei Odaira, Jose G Castanos, and Hisanobu Tomari. Eliminating global interpreter locks in ruby through hardware transactional memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 131–142. ACM, 2014.

[17] R. Rajwar and J.R. Goodman. Transactional lock-free execution of lock-based programs. In *In Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[18] H.E. Ramadan, C.J. Rossbach, D.E. Porter, O.S. Hofmann, a. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional Memory for an Operating System. *IEEE Micro*, 28(1):42–51, January 2008.

[19] C.J. Rossbach and O.S. Hofmann. TxLinux: Using and managing hardware transactional memory in an operating system. *SIGOPS Operating Systems Review*, 26(2):87–101, 2007.

[20] Zhaoguo Wang, Han Qian, Jinyang Li, and Haibo Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th ACM European Conference on Computer Systems*. ACM, 2014.

[21] Zhaoguo Wang, Hao Quan, Haibo Chen, and Jinyang Li. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *Proc. 4th Asia-Pacific Workshop on Systems (APSYS 2013)*, 2013.

[22] Richard M. Yoo, Bratin Saha, Yang Ni, Adam Welc, and Hsien hsin S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *In ACM Symp. on Parallelism in Algorithms and Arch. (SPAA)*, pages 265–274. ACM Press, 2008.