Hardware Transactional Memory in C++

Mario Dehesa Azuara **Nick Stanley**

May 2016
Reissued: December 2018
CMU-CS-18-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We implemented various data structures and algorithms using Intel TSX (Transactional Synchronization Extensions) in order to determine how well TSX performs, how to optimize programs with TSX, and when TSX is an appropriate tool. We could not find general guidelines for when TSX is the appropriate solution to a problem, so we investigated when TSX is the appropriate tool and when it is less feasible. We discuss the challenges associated with implementing concurrent data structures with transactions, our solutions to some of these problems, and empirical evaluations of different heuristics and the performance of various data structures with TSX.

Note about republication: This report was originally produced as part of Carnegie Mellon University's course 15-618 and was published on a website hosted by the university in May of 2016.

Keywords: computer science, tech reports, TSX, hardware transactional memory

1 Background

Hardware Transactional Memory (HTM) - hardware support for transactions - is a relatively new approach to concurrency. While the first implementation plans have been around for almost a decade (starting with Sun's plans for the Rock Processor), the release of Intel's Transactional Synchronization Extensions (TSX) has created new opportunities for the programmer. In this project, we analyzed the performance of TSX compared to locking and lockless methods. In particular, we designed transactional hashtables and binary trees and compared them to their fine-grained locking variants. We also created microbenchmarks (e.g., swapping the contents of two memory locations) and attempted to speed up paraGraph, the third assignment for this class, by replacing the lockless operations with transactions. One should expect that TSX outperforms traditional methods (i.e., locks and atomic instructions) of concurrent programming when conflicts are sparse (since lockless methods flood the interconnect, and locks require many reads from memory). However, replacing these traditional methods is not as simple as find-replacing the critical sections and wrapping them in an atomic block. Various decisions have to be made about how to implement transactions. For instance, Intel makes it very clear that transactions are never guarenteed to make progress. If two transactions write to the same place in memory, they may both abort and retry forever. With TSX, the assembly instruction xbegin can return various results that represent the hardware's suggestions for how to proceed and reasons for failure: success, a suggestion to retry, a potential cause for the abort, etc. Listening to these primitives, we can decide whether or not to retry. If we don't retry, we will have to fall back on more traditional methods like locked data structures or atomic operations. For example, a global lock is a possible fallback for a randomized treap that operates primarily on transactions. To effectively use TSX it's imperative to understand it's implementation and limitations. TSX is implemented using the cache coherence protocol, which x86 machines already implement. When a transaction begins, the processor starts tracking read and write sets of cache lines which have been brought into the L1 cache. If at any point during a logical core's execution of a transaction another core modifies a cache line in the read or write set then the transaction is aborted. These implementation details have several important side effects and implications. The first is that the burden of aborting a transaction is on the core which detects the conflict. Moreover by using the L1 cache to track the read and write sets the amount of memory we can touch is limited by the characteristics of the cache, meaning it's size and eviction policy. As one would suspect, any eviction of a modified cache will cause a transaction to fail, but, perhaps somewhat surprisingly, evicting an unmodified cache line may not. The processor uses a secondary probabilistic data structure to track "read" cache lines which were evicted (we believe it's using a Bloom filter, although Intel does not specify) and checks against this data structure when a core announces it would like to modify a cache line. Apart from the causes described above there are a number of situations which will cause a transaction to fail, such as the following:

- A ring transition
- Invocations of 'strcmp'
- Invocations of 'strcpy'

- Invocations of 'new'
- Invocations of 'delete'
- Interrupts
- A CPUID instruction

The last one is important as many debuggers, linkers, or self modifying programs are likely to use this instruction. Other than these, there are number of miscellaneous reasons why a transaction may fail. For example, executing a routine from a dynamically linked library for the first time or having a thread which using the cache intensely hyper threaded on the same core. Since there are so many reasons external to the actual code being executed that can cause abortions, it is mandatory for a fallback path to always be available. To understand TSX and develop a practical understanding of it's characteristics we set out to implement a series of concurrent data structures. We chose a common set of associative containers which we thought would be good case studies for hardware transactional memory. Our goal with these implementations was to develop a set of best practices, get and empirical sense of the limitations of the feature, and investigate of the cost associated with using hardware transactional memory. While writing each of the data structures transactionally was certainly easier than their fine grained locking counter parts, we found that transactions by no means eliminate the issues surrounding the implementation of concurrent systems. One of the first challenges we encountered was writing a correct fall back path. We wanted to keep our fallback paths simple, but found that even naive solutions presented technical challenges. Transactional memory is advertised to be efficient when data collisions are unlikely, combined with our the intention of keeping our fallback paths simple we were motivated to implement "lock the world" fallback paths. A naive first attempt at this might be:

```
Result status = _xbegin();
if (status == SUCCESS) {
/* Enter Critical Section */
_xend();
} else {
// Fall back path

lock();

/* Enter Critical Section */

unlock();
}
```

The problem here is that if a thread is in the fallback path, reads from memory, then another thread starts a transaction, modifies the value, and commits the transaction before the thread with the lock runs, the data will be corrupted. So a correct implementations will instead follow the following pattern:

```
 // Spin on _stop_the_world == true
while (_stop_the_world);

Result status = _xbegin();
if (status == SUCCESS) {
if (_stop_the_world) {
_xabort();
}

/* Enter Critical Section */

_xend();
} else {
/* Fall back path */
lock();
_stop_the_world = true;

/* Enter Critical Section */

_stop_the_world = false;
unlock();
}
```

The key here is that any thread in the transaction has to read the `_stop_the_world` flag, so when someone executes the fallback path and modifies it all threads in the transaction will abort. Any new threads coming into the transaction will see the flag an hence cannot execute the critical section when a thread is in the fallback path. A dangerous issue is that with this pattern we can snowball into a situation where all threads are forced into the fallback path, which then causes all future threads to use the fallback path and so on and so forth. To prevent this, we add a while loop before the transaction begins that hopefully will prevent threads from aborting and taking the fallback path too many times unnecessarily.

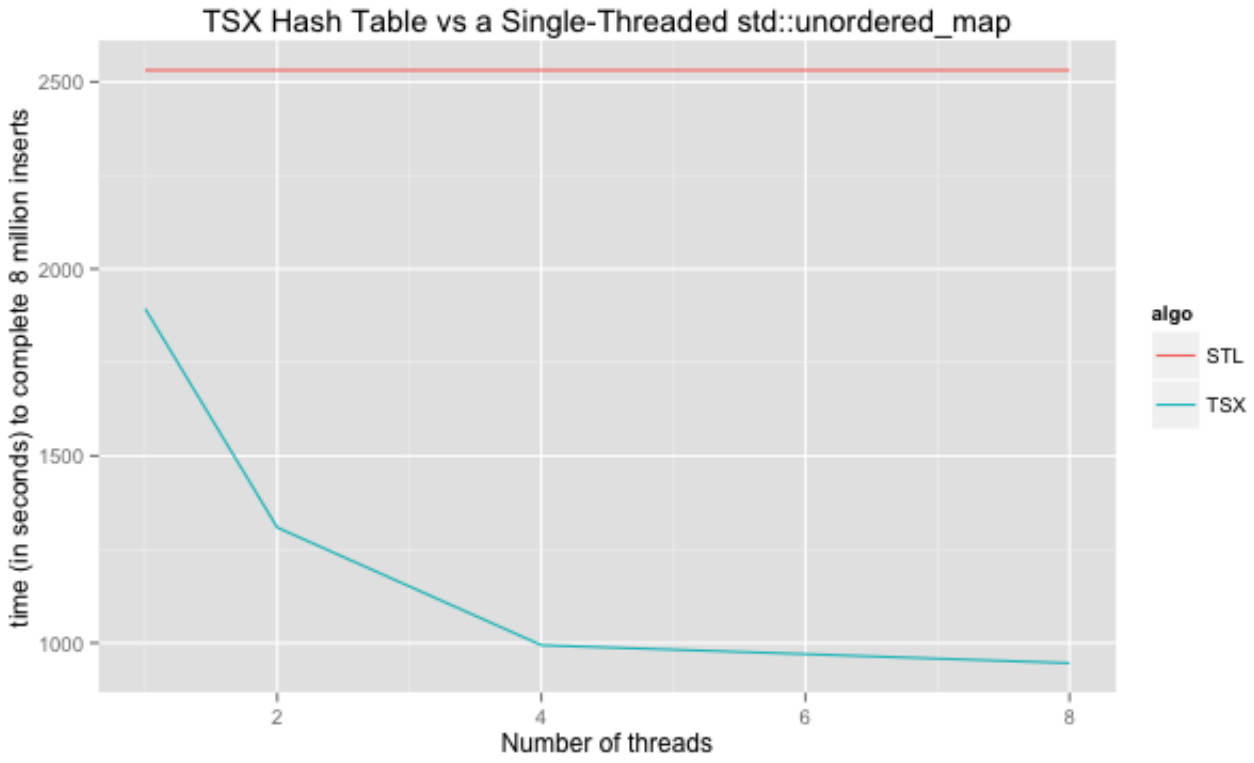
2 Approach

We began our project by writing transactional AVL Trees, treaps, and a Hashtable. We compare the first two against a fine grained locking BST and the latter against STL's unordered map. We chose these trees because they seemed easy targets for a transactional implementation: their operations are fairly simple and moreover they touch a small number of elements in each operation. We found that the randomized treaps outperformed both our TSX AVL trees and our fine grained BSTs. While the AVL trees maintain better balance (at depth 30 for 40 million insertions) comapred to treaps (depth 100 for 40 million insertsion), the AVL tree's rotaions cause a huge number of data conflicts which leads to more transactional failures than commits. The hashtable was an interesting

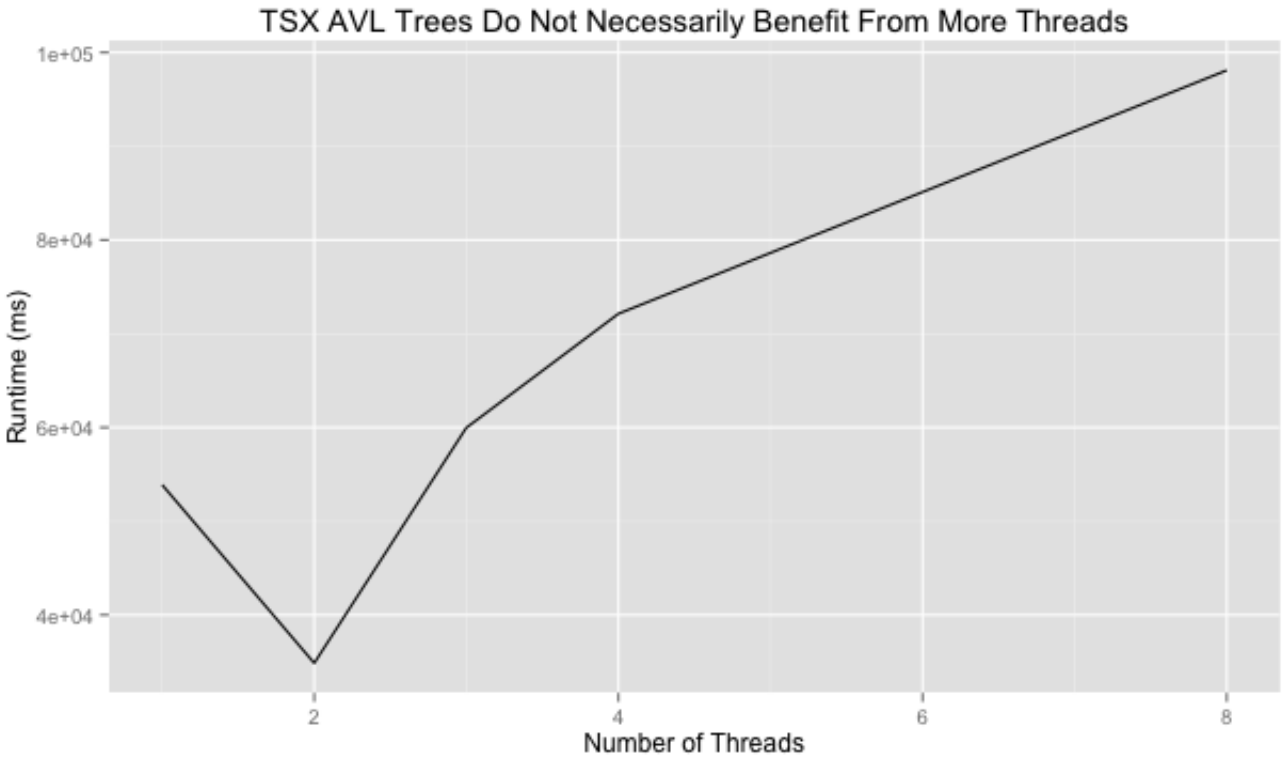
case study because of regrowth. We chose to implement regrowth as a stop-the-world event to keep things simple and see how this approach scaled. We found that with a few optimizations our transaction hash table scaled well. While each of these data structures certainly required a certain amount of care, since we are still writing concurent code, we found that TSX greatly simplified the problem. A good example is in the case of treaps. We wanted to maintain the invariant that our treaps had no duplicate keys, so to achieve this property upon insertion we first lookup the key we are inserting and then insert the new node. In a fine grained locking approach this would not work, because two threads could do the look up "at the same time", both observing the key to be absent in the tree and then insert the same key twice. With a our TSX approach this problem does not occur, because performing a lookup will read every node on the path required to search for the key in question and any insertion of that key would have to modify a node on that path. So the transaction will fail who ever attempts to commit second. After implementing each of these data structures, we attempted to improve the performance of our paraGraph implementation. BFS and Graph Decomposition seemed like particularly good targets, but replacing the critical sections of the corresponding update functions in each of these applications showed to actually slow down the applications. The critical sections in each of these applications were very short with only a very small number of atomic operations, so we found that the overhead of performing a transaction outweighed the cost of these small operations. To test these claims, we performed microbenchmarks to examine the overhead of TSX itself. Our three micro benchmarks were 1) spinning on a counter, 2) fine-grained locking to swap two elements in an array, and 3) batch-incrementing an array. For spinning on a counter and incrementing, we performed much worse than simply using atomic operations - this is unsurprising because the counter is highly contended. For swapping under a lock, we randomly chose two elements in an array, locked them (in ascending order), and swapped their values. Here, TSX beat the locked swapping significantly, because it had few conflicts, and fewer writes to memory. Finally, to test the theory that xbegin and xend had a high initial cost, we attempted to increment the values in an array in "batches". That is, for every xbegin, we increment on K different memory locations, where K is a variable set by the programmer. We then compared this to performing K atomic increments on separate memory locations. Here, we found that there is a crossing-over point between lockless and transactional implementations: see the results section.

3 Results

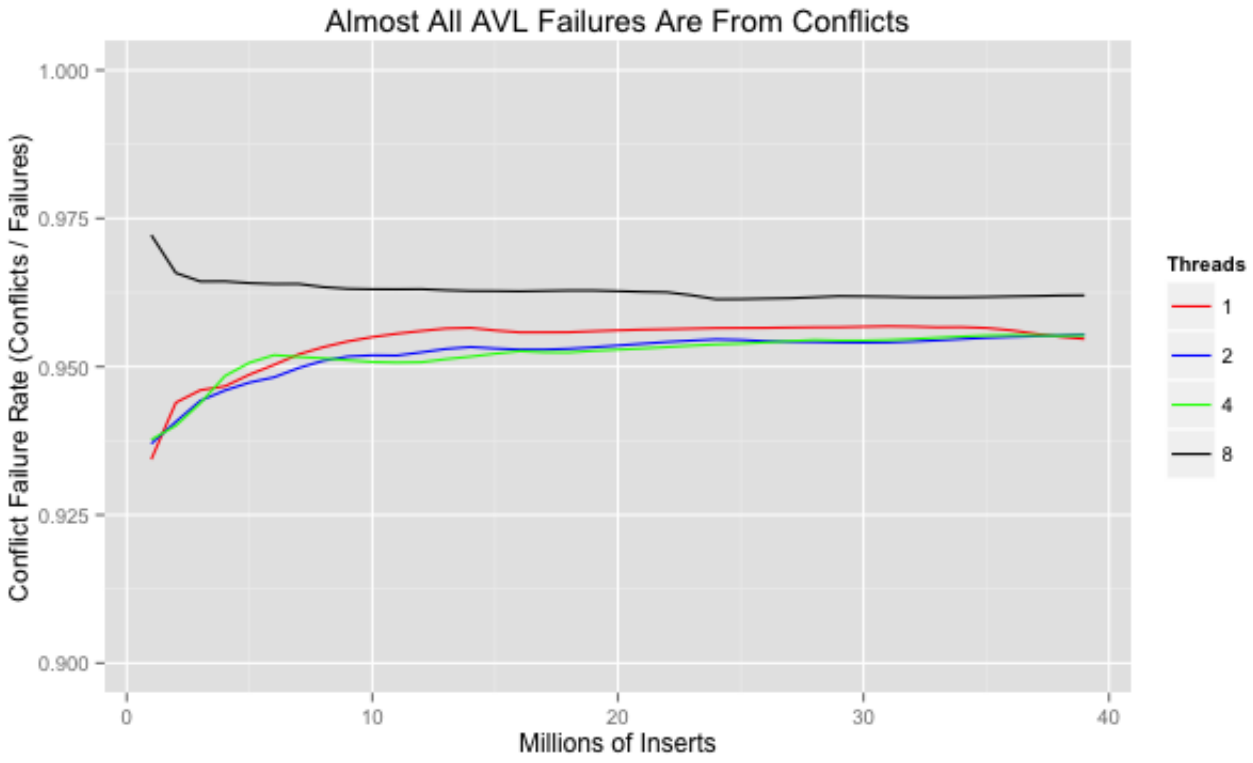
We conclude with an account of our empirical results.



For a hashtable, we created a separate chaining hashmap with a global fallback lock. Luckily, fallbacks happened rarely, and so this did not affect our overall speed. As seen in the figure above, our TSX implementation of a hashtable scales well to multiple threads. Note the performance leveling off on 8 threads, where hyperthreading is activated - here, we jump from conflict failure rates of 0.15% to 0.6%. While we did outperform the standard library we map we recognize that our hashtable exposes a much more limited functionality and thus the comparision is unfair, we simply display it for a frame of reference.

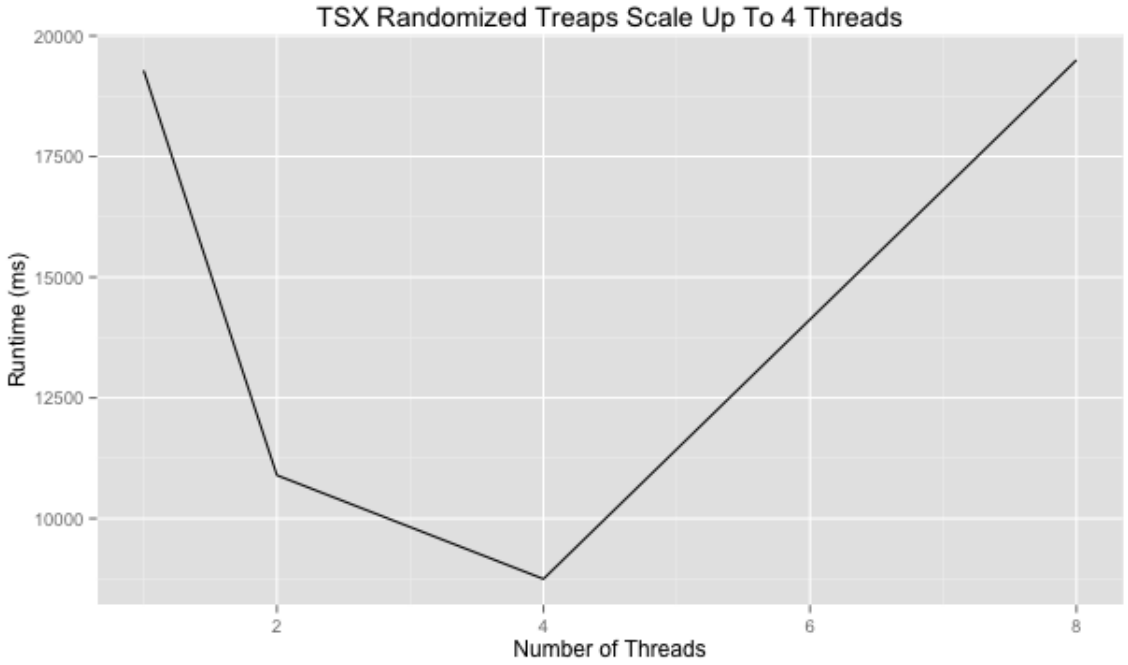


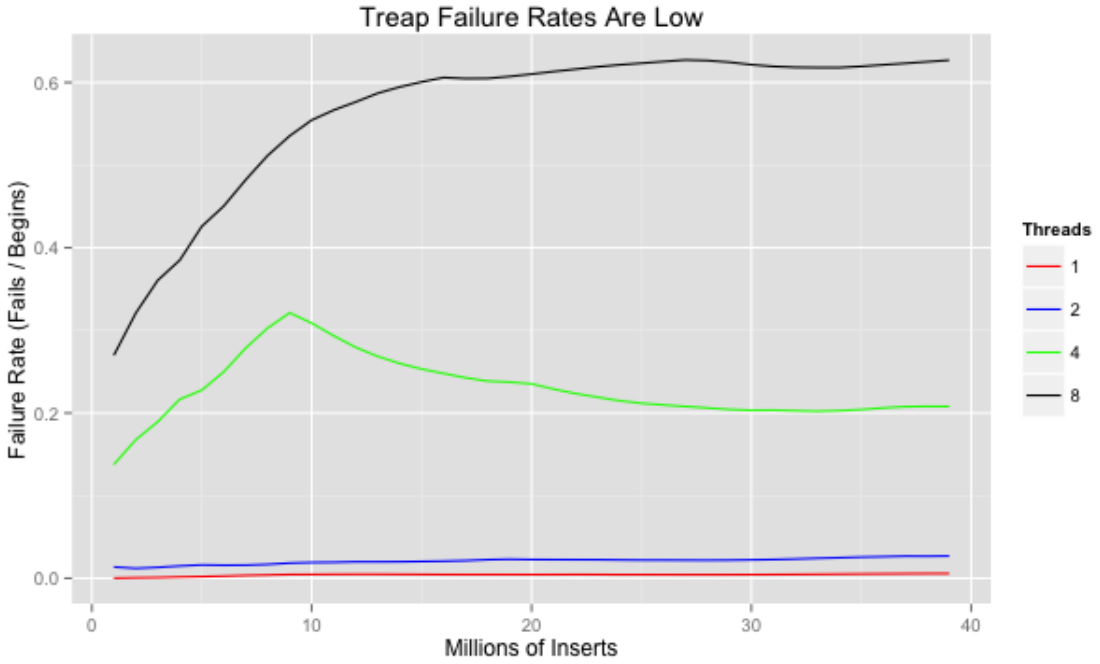
While it may seem intuitive that having twice as many threads means your program will run at least as fast over a large input, this is not always true when using transactions. We initially believed that this is because a higher number of transactions increase the likelihood that two transactions will conflict and cause one to abort. In particular, for AVL trees, every insert touches many nodes to keep the balanced invariant, so as the thread count gets high, so does the chance of conflict.



But, we can see that the failure rates for AVL trees is in fact very high - Even with a single thread, the failure rate (number of failed transactions / number of begun transactions) is very high, leveling off at about 60%. While this can be explained by data conflicts for multiple threads, we believe the failure rate for the single threaded program is explained by the fact that AVL trees touch many different cache lines, and so cause its own failure simply by touching too much memory (e.g., by evicting a cache line in its write set).

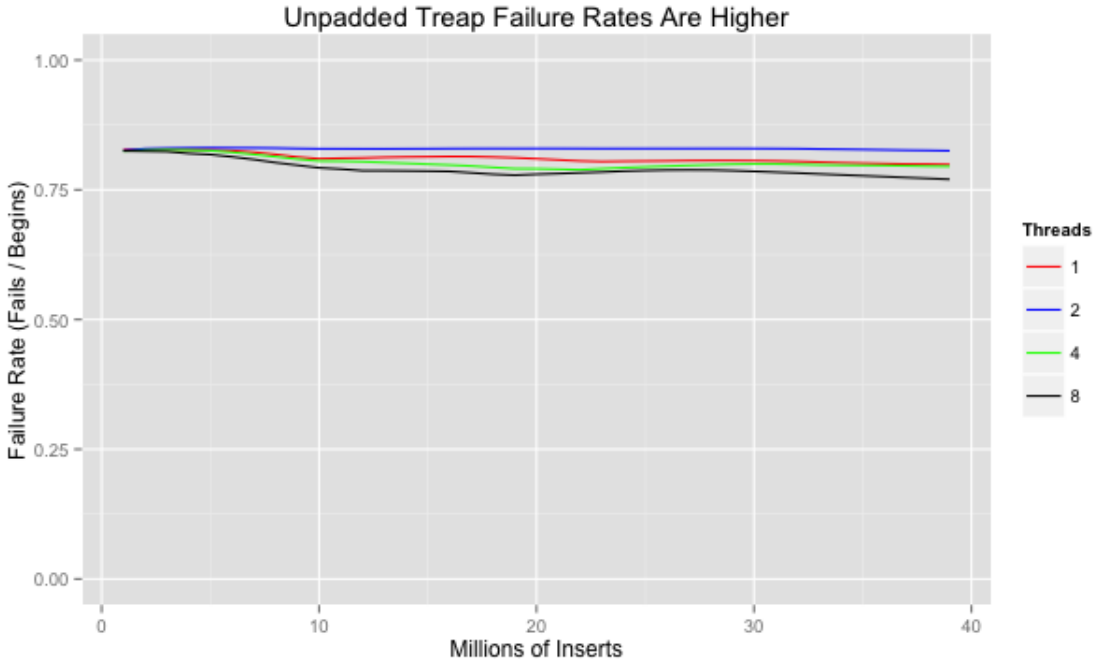
But as interesting counterevidence, the flags returned from the assembly xbegin instruction indicate that almost all AVL failures happen from ”conflicts”, even on a single thread. Intel’s TSX Programmer’s Manual mentions that the return values from xbegin are not meant to be exact guides. So we believe this may be a result of some ambiguity in the return values of xbegin.



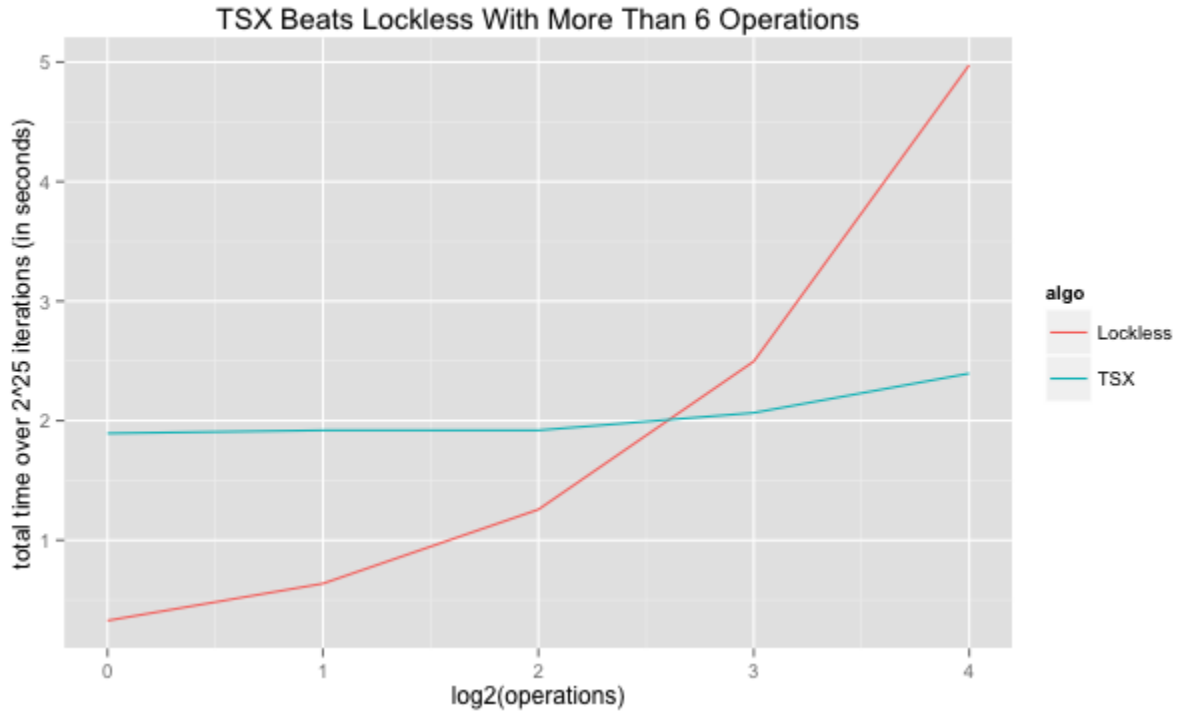


Even though we were unsuccessful for the AVL tree, for the randomized treap, we achieve a great amount of speedup with TSX. This is because the AVL tree must perform many writes in its rebalancing step, while the randomized treap writes to at most a small constant number of nodes (three nodes: to write the parent's pointer, the new node's pointers, and the displaced child's pointers). While it only provides probabilistic balancing and, in general, has a greater depth than

the AVL tree implementation, we believe that the importance of the lack of balancing (which would touch too many cache lines and force aborts) was outweighed by the lack of contention in binary trees and the fact that only one write is made. However, while the runtime decreases as one would expect for 1, 2, and 4 threads, our TSX randomized treap algorithm performs worse for 8 threads than even 1 thread. We believe that this is because a context switch due to hyperthreading will touch many cache lines, increasing the likelihood of an eviction happening and thus aborting a transaction.



However, to achieve faster speedup, we believe that we need to separate the data in the randomized treap as much as possible. This means that if we have a seed for the treap that generates random priorities, and a pointer to the root, we should keep them on separate cache lines so that generating the seed does not conflict with grabbing the root node. The above graph shows that, if we do not align these values to their own cache lines, then we have many more conflicts: every thread count has a failure rate of around 80%.



In addition to larger data structures, we also implemented several programs that tested the efficiency of TSX over the more traditional concurrent methods. The above figure shows the runtime when we batch incremented a number of elements in an array in a transaction, and when we performed the same increments with atomic instructions. We ran a fixed number of batches of various numbers of instructions (1 to 16). Our original hypothesis was that calling xbegin and xend had a fixed cost, and that after the critical section reached a certain number of operations, TSX strongly outperformed the atomic instructions. In particular, we believe that the ”magic number” is approximately 6 (by interpolating between 4 and 8 in the above graph). This was done on a single thread to control for collisions

4 Related Work

- Cuckoo Hashing with TSX
- Intel’s Optimization Manual
- Intel’s Recommendations On Fallback Paths

5 Note about republication

This report was originally produced as part of Carnegie Mellon University’s course 15-618 and was published on a website hosted by the university in May of 2016.