Investigating the performance of hardware transactions on a multi-socket machine

Trevor Brown *
University of Toronto
tabrown@cs.toronto.edu

Alex Kogan, Yossi Lev, Victor Luchangco
Oracle Labs
{alex.kogan, yossi.lev, victor.luchangco}@oracle.com

Abstract

The introduction of hardware transactional memory (HTM) into commercial processors opened a door for designing and implementing scalable synchronization mechanisms. One example for such a mechanism is transactional lock elision (TLE), where lock-based critical sections are executed concurrently using hardware transactions. So far, however, the effectiveness of TLE and other HTM-based mechanisms has been assessed mostly on small, single-socket machines.

This paper investigates the behavior of hardware transactions on a large multi-socket machine. Using TLE as an example, we show that a system can scale as long as all threads run on the same socket, but a single thread running on a different socket can wreck performance. We identify the reason for this phenomenon, and present a simple adaptive technique that overcomes this problem by throttling threads as necessary to optimize system performance. Throttling decisions are made on a per-lock basis, and we demonstrate that our technique performs well even in workloads where the best decision is different for each lock.

1. Introduction

To perform well on modern multiprocessor systems, applications must exploit the increasing core count on these systems by executing operations concurrently on different cores without introducing too much overhead in synchronizing these operations. Recent systems have introduced *hardware transactional memory* (HTM) [16] to support efficient synchronization, and previous work has shown that it can be used effectively [2, 10, 13–15, 21, 28]. However, until recently, HTM has been available only on relatively small single-socket systems. In this paper, we investigated the behavior of HTM on a large multi-socket machine and observed that it differs from the behavior of the smaller systems in ways that present challenges for scaling the performance on the larger machine.

For example, consider the graph on the left in Figure 1, which shows the speedup over single-thread execution of a microbenchmark in which we continually insert and delete items in an AVL tree [1]. Operations on the tree are protected by a single lock, to which we apply *transactional lock elision* (TLE) [8], a popular technique for exploiting HTM in which lock-based critical sections are executed concurrently using hardware transactions, and which is implemented on top of the Intel Haswell TSX/RTM interface. The machine has two sockets, each with 18 hyper-threaded cores, for a total of 72 threads. We bind threads so that the first 36 all run on one socket, and the last 36 threads run on the other socket. (This microbenchmark is described in more detail in Section 3.) As we can see, performance improves until we reach 36 threads, even though the scaling is much more moderate after 12 threads. As soon as any thread executes on the second socket, however, the performance drops dramatically. Performance continues to decline until the machine is saturated, at which point its performance is barely better than with a single thread. This behavior is in stark contrast to the one measured on a smaller single-socket (4 hyper-threaded cores) machine, shown on the right in Figure 1, in which the performance continues to improve until the machine is saturated.

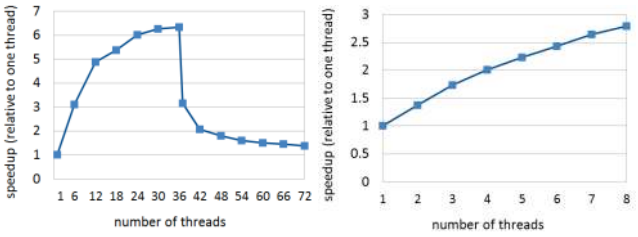

Figure 1. AVL tree microbenchmark on a large HTM system (left), and a small HTM system (right). The workload is 100% updates with key range $[0, 2048)$.

Not all benchmarks exhibit this pathology: with only lookup operations (i.e., a read-only workload), for example, performance scales all the way to 72 threads (i.e., the full capacity of the machine). We experimented with a variety of microbenchmarks to identify the causes of this performance pathology, as well as other differences between the behaviors of HTM on the large and small machines. In this paper we describe several of these experiments and their results, and the conclusions we draw from them.

Informed by these results, we explored several ways, described in the latter half of this paper, to use HTM effectively on the multi-socket machine. The technique we found most effective is to adaptively throttle the number of threads to optimize performance. At a high level, the idea is to profile TLE performance and decide, separately for each lock, whether critical sections protected by that lock should be executed by threads on multiple sockets, or only on a single socket. Through a set of experiments with several microbenchmarks we show that this simple approach avoids sharp performance degradation for workloads that do not scale beyond one socket while exploiting additional sockets for workloads that do scale. We describe this technique in greater detail and present experimental results that demonstrate its effectiveness.

2. Background

The machine we are studying is an Oracle Server X5-2, a two-socket system with two Intel Xeon E5-2699 v3 processors. Each processor has 18 cores, each with 2 hardware threads (i.e., 72 threads in all), running Ubuntu 15.04 at 2.30GHz. Because of its two-socket design, this machine has *non-uniform memory access* (NUMA): Cores on the same processor share an L3 cache, so

* This work was done while Trevor Brown was an intern at Oracle Labs.

communication between threads running on the same processor is much faster than inter-socket communication. For comparison, the other (smaller) machine we used in Figure 1 is a single-socket 4-core hyperthreaded Haswell Core i7-4770 (also with 2 threads per core, so 8 threads in all) running Oracle Linux 7 at 3.40GHz.

Both systems provide "best effort" hardware transactional memory (HTM): transactions are not guaranteed to *commit*, but any transaction that commits appears to other threads to have been executed atomically. A transaction that does not commit is *aborted*: its writes to memory are discarded and are never made visible to other threads. A transaction that aborts sets a condition code indicating the cause, and then jumps to a fallback code path specified at the beginning of the transaction. This condition code allows to distinguish, for instance, between transactions that abort because some internal buffer overflowed and those that abort due to conflict with another thread. It also includes a hint bit that indicates whether, according to the hardware, the transaction is likely to succeed if retried.

We use transactional lock elision (TLE) [8] as a vehicle to study the behavior of HTM. TLE is a practical synchronization technique for exploiting HTM that can be applied without requiring changes to code that uses traditional lock-based synchronization: it co-opts the *LockAcquire* and *LockRelease* operations that bracket a critical section, attempting to elide the lock acquisition (and subsequent release) by executing within a transaction. If the transaction commits, it ensures that the critical section executes without interference from other threads. If the transaction aborts, the critical section may be retried in another transaction or it may be executed in the traditional fashion by acquiring the lock. (A transaction executing a critical section must check that the associated lock is free to avoid executing concurrently with a critical section executed under lock.) If the transaction aborts due to overflow (for which typically the hint bit is not set), it is common to not retry at all.

Numerous studies have observed that TLE provides nearly ideal speedups when applied to workloads in which threads have few data conflicts and transactions do not overflow [10, 14, 21, 22, 28]. However, when these conditions do not hold, the performance of TLE deteriorates quickly. Several recent papers have suggested ways to improve the performance of TLE in these scenarios, employing various approaches such as adaptively tuning retry policies [10, 13] and introducing auxiliary locks [2, 15]. All these papers, however, evaluated TLE and suggested improvements using relatively small, single-socket machines.

Most of our experiments use TLE applied to an AVL tree implementation that uses a single global lock to synchronize access to the tree. An AVL tree [1] is a balanced binary search tree that ensures that the heights of the left and right subtrees of any node differ by at most 1. Whenever an operation disrupts this balance (by adding or deleting a node), it "rotates" the node and/or its ancestors to restore the balance. Most insert and delete operations do not conflict with other operations because they update only a few nodes at the bottom of the tree, but a few operations may rebalance even the root node, and so conflict with every other operation.

The effects of NUMA on the performance of multithreaded systems has been an area of active research in the past few decades [4–6, 17, 18, 25, 26]. This research is motivated by the observation that remote memory accesses and remote cache misses (i.e., cache misses served from another cache located on a different socket) are expensive, and therefore should be reduced. Dice et al. [12], for instance, achieve this goal through the design of a series of NUMA-aware *cohort locks*, which allow threads running on the same socket to pass the lock among themselves (and thus exploit cache locality within the socket) before releasing the lock so that it can be acquired by a thread on a different socket. Other researchers attempt to colocate threads and the data they access through thread migration [4, 19] or data migration and replication [26].

Prior work in various contexts has considered restricting concurrency to improve system performance. Multiple papers, for instance, observe that using fewer threads than the available cores can lead to better performance [7, 23, 24], and suggest ways to tune the number of running threads to achieve that. In the context of software transactional memory, several papers consider contention management mechanisms that throttle some of the conflicting software transactions to increase the chance that remaining transactions succeed [3, 27]. In the context of TLE, Diegues et al. [15] suggest using *core locks*, which synchronize between hardware transactions running on same core when those transactions abort due to overflow. In the same context, Afek et al. [2] consider adding an auxiliary lock to TLE, which is acquired by conflicting transactions. The above mentioned cohort locks [12] are perhaps most relevant to the concurrency restriction ideas that we consider in this paper. They throttle threads running on sockets other than the one holding the lock, thus sacrificing short-term fairness for higher throughput.

Delegation is another approach aimed at reducing remote cache misses. The idea of delegation is to mediate access to a data structure or critical section by one or more server threads, which execute requests from client threads. Client and server threads communicate via message passing implemented on top of shared memory. As an example, Lozi et al. [20] propose structuring a client-server system in which server threads running on dedicated cores execute critical sections on behalf of client threads. Calciu et al. [6] investigate different approaches for implementing message passing and show that while delegation can be effective, the communication overhead of message passing often outweighs its benefits.

3. HTM on a Large Machine

In this section, we present and analyze the results of several experiments we ran to better understand the behavior of HTM on the large (72-thread) NUMA system. In most of these experiments, threads repeatedly invoke insert, delete and lookup operations on implementations of an abstract set (e.g., an AVL tree) using a key selected uniformly at random from a specified key range. The set is prefilled with approximately half the keys. We specify the percentage of lookup operations, with the remaining operations being evenly split between insert and delete operations (collectively update operations) to keep the set approximately half filled. Unless stated otherwise, in our experiments on this system threads are pinned so that threads 1-36 run on the first socket, and threads 37-72 run on the second socket. (We describe alternative pinning policies in Section 6). As we show next in Section 3.1, some of our results do not necessarily relate to NUMA, but rather to the fact that our system has a large number of cores. Section 3.2 details findings that are more NUMA-specific.

3.1 Going beyond 8-12 cores

We tested whether the conventional wisdom about retry polices in TLE on small systems is appropriate for the large system. Figure 2 shows the results of a microbenchmark on the large HTM system wherein threads perform insertions and deletions (i.e., no lookup operations) in an AVL tree with key range $[0, 131072)$ (i.e., containing approximately 65536 keys) using different retry policies. (This tree fits in the L3 cache but is large enough that there is a very little chance for two concurrent operations to conflict.) *TLE-5 fallback on overflow* implements a policy commonly used in small HTM systems: we attempt to execute a critical section using a transaction up to 5 times before falling back to the lock. We do not count attempts that fail because the lock is held (in which case the transaction is not retried until the lock is released in order to avoid the *lemming effect* [9]). However, if a transaction aborts due to over-

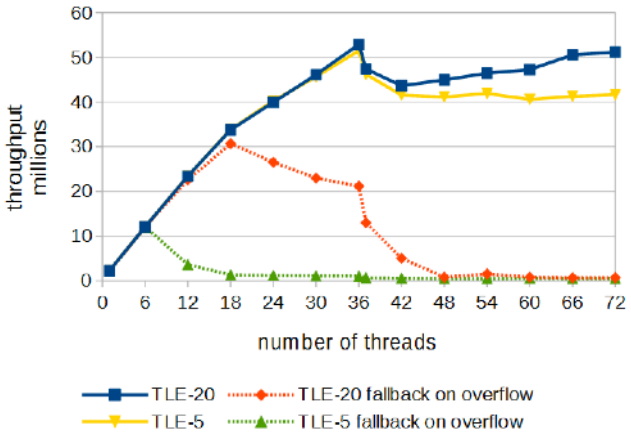

Figure 2. TLE using different retry policies on a large HTM system with 100% updates on an AVL tree with key range $[0, 131072)$.

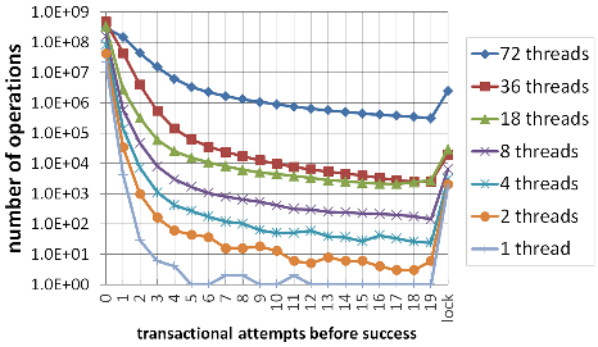

Figure 3. Histogram showing how many transactional attempts are made before a transaction successfully commits in a workload with 100% updates on an AVL tree with key range $[0, 131072)$. After 20 attempts, the lock is taken. Note the log-scale.

flow, we fall back to the lock immediately. *TLE-20 fallback on overflow* implements the same policy except that 20 attempts are allowed before falling back to the lock. *TLE-5* and *TLE-20* are similar except that they do not fall back to the lock immediately when a transaction aborts due to overflow.

The difference between *TLE-5 fallback on overflow* and *TLE-20 fallback on overflow* demonstrates that it may be useful to attempt to execute a critical section transactionally more than 5 times on a large HTM system. This reflects the greater cost of falling back to the lock on a system running 72 threads than on one running only 8, because a thread that takes the lock blocks every other thread. Thus, in the large system, we can tolerate more failed transactions to avoid taking the lock.

Perhaps surprisingly, contrary to the intuition that a transaction that fails due to overflow will continue to fail if it is retried, we observed that subsequent attempts often succeed. Thus, the "optimization" of falling back immediately when a transaction aborts due to overflow caused threads to fall back to the lock unnecessarily, and eliminating it significantly improves performance. (For instance, at 36 threads, *TLE-20* achieves more than twice the performance of *TLE-20 fallback on overflow*.) Allowing 20 transactional attempts rather than only 5 also improves performance, though to a lesser extent.

Might raising the limit on transactional attempts beyond 20 improve performance even more? Figure 3 shows how many attempts were needed to complete each operation using *TLE-20*. The last

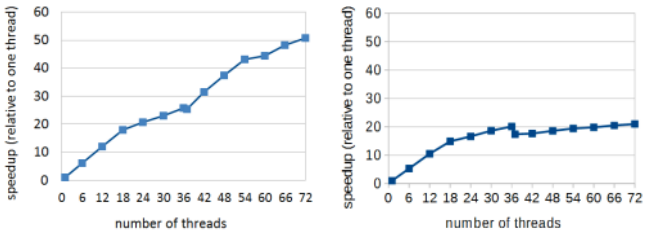

Figure 4. An AVL tree on a large HTM system. The workload is 100% lookup on the left, and 2% updates (i.e., 98% lookups) on the right, with key range $[0, 2048)$.

point (labeled lock) shows the number of operations that fall back to taking the lock after 20 failed attempts, a small fraction of the total number of operations (note the log scale the figure). Nonetheless, we experimented with varying the limit over a wide range, up to hundreds of millions of retries, and, unsurprisingly, we were not able to consistently improve performance beyond that achieved with 20 attempts. In the rest of this paper, unless otherwise specified, we report on results for *TLE-20*.

3.2 How NUMA affects HTM

For the workload depicted in Figure 2, performance does not significantly decrease when threads run on two sockets. This is because operations on a large AVL tree are only lightly contended, and in fact, as demonstrated by Figure 3, most of them succeed on HTM in the very first attempt. In a smaller tree, however, concurrent operations are more likely to conflict, and thus the performance suffers greatly once threads execute on the second socket. We already saw this in Figure 1, which depicted *TLE-20* in an AVL tree with key range $[0, 2048)$: Adding a single thread on the second socket cut performance in half, and performance at 72 threads was reduced almost to that of a single thread. This drop in performance is caused by NUMA effects (i.e., the increased latency of inter-socket communication).

It is possible to scale to the full capacity of the large HTM system. As an example, in a read-only workload, TLE scales all the way to 72 threads. However, performing just 2% updates flattens the curve after 36 threads, completely negating the benefit of the second socket (see Figure 4).

Although NUMA effects are well known to negatively impact performance, this impact is amplified by the use of HTM. Figure 5 shows the results of a simple experiment in which threads repeatedly search for a randomly chosen key and then write the key found in the last node visited by this search into the key field of that node. (This might not be the key chosen because that key may not be in the set). This *search-and-replace* operation can be implemented without any synchronization because it does not actually change the key (i.e., it writes the same value that is already in the field). As we can see from this figure, NUMA effects impact TLE much more than the algorithm with no synchronization. The algorithm with no synchronization drops from 12.1x speedup (over a single thread) at 36 threads to 8.9x speedup at 72 threads (a 26% decrease in performance). However, the TLE algorithm experiences a much larger drop from 6.4x speedup at 36 threads to 1.6x speedup at 72 threads (a 75% decrease in performance).

To understand why TLE suffers so greatly from NUMA effects, consider Figure 6, which shows how many transaction attempts aborted, and the reasons (i.e., the condition code) they did so. The fraction of transactions that abort dramatically increases as threads are added on the second socket, from 10% at 36 threads to 33% at only 42 threads. Beyond 36 threads, the vast majority of these aborts are reported by hardware as data conflicts.

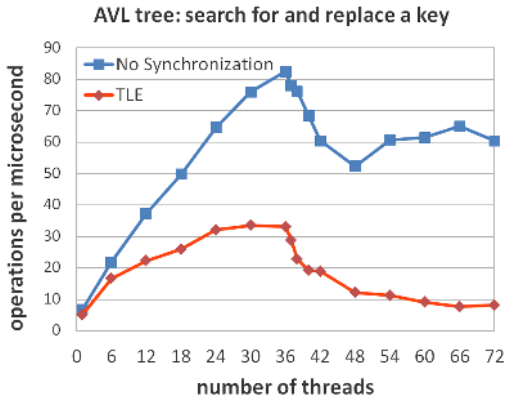

Figure 5. Comparison of TLE vs. no synchronization in a workload doing search-and-replace operations on an AVL tree with key range $[0, 4096)$.

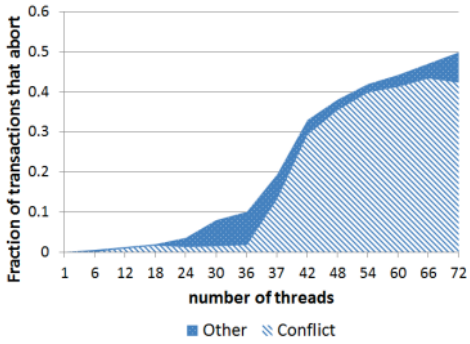

Figure 6. Abort rate for the TLE curve in Figure 5. (Note: the x-axis is not to scale near x=37). The workload is 100% updates with key range $[0, 2048)$.

We hypothesize that these aborts occur because cross-socket cache invalidations lengthen the time needed to complete a transaction, which, in turn, lengthens the "window of contention" during which it may conflict with other transactions, increasing the likelihood of such conflicts. In contrast, when a cacheline is invalidated by a thread on the same socket, it can be restored much faster because the threads share an L3 cache.

Our hypothesis explains why performance is poor with even a single thread on the second socket (operations performed on the second socket cause expensive cache misses on the first socket), why read-only workloads scale on both sockets (threads do not cause cross-socket cache invalidations), and why the impact of NUMA effects on *TLE-20* is more severe in Figure 1 than in Figure 2 (in the small tree, there is a higher chance of conflicts, and operations complete more frequently, so more cache invalidations occur).

To confirm our hypothesis, we ran a 36-thread *single-socket* experiment that added some artificial delay (spinning) just before committing each transaction.[1] The results in Figure 7 show that, with a certain amount of delay, the abort rate jumps significantly, mimicking the results on two sockets, and that once the abort rate

[1] The delay was implemented by varying a number of loop iterations, each consisting of a small, constant number of instructions. The average length of successful transactions increased from about 61ns (without the delay) to about 43 μs (with the maximal delay of 10K iterations). Even with the maximal delay, transactions are short enough so that the chance for aborts imposed by context switch interrupts is negligible.

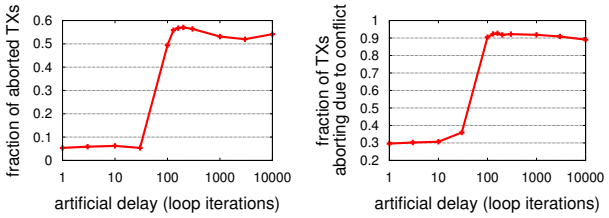

Figure 7. Abort rate (left) and fraction of transactions aborting due to conflict (right) for a 36-thread single-socket experiment in which delay is inserted before committing each transaction. The workload is 100% updates in an AVL tree with key range $[0, 131072)$.

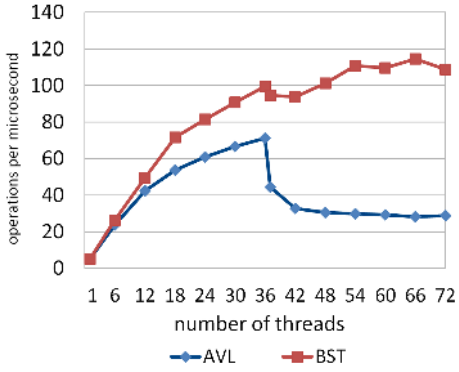

Figure 8. Comparison between an AVL tree and a leaf-oriented BST, with 20% updates and key range $[0, 2048)$.

increases, most transactions that abort do so because of conflicts, mimicking the phenomenon in Figure 6. These results provide strong evidence for our hypothesis.

Our hypothesis predicts that NUMA effects will be less significant for unbalanced leaf-oriented trees, where each update modifies a leaf, or the parent of a leaf. In such a tree, threads can only cause cross-socket cache invalidations near leaves, so the top of the tree is likely to remain cached. We tested this prediction by comparing the AVL tree to an unbalanced leaf-oriented binary search tree. As we can see in Figure 8, which shows the result of a workload with 20% updates and a key range of $[0, 2048)$, the leaf-oriented BST scales much better than the AVL tree.

An alternative explanation for this behavior is that a transaction may abort whenever it experiences a last-level cache (LLC) miss (as it does, for example, whenever it experiences a page fault). To rule out this possibility, we designed a simple experiment in which a single thread allocates a one gigabyte array of bytes, and then iterates over the cells of this array, starting a transaction, reading a word, and committing. This array is too large to fit in cache, so many of the thread's reads cause LLC misses. There is one complication: Whenever a thread reads a location in memory, modern Intel processors fetch the entire cache line containing the memory location, and also prefetch the next cache line. To avoid prefetching effects, the thread skips two cache lines (128 bytes) in between each pair of reads. In this experiment, nearly every read should incur a LLC miss, so we can approximate the number of LLC misses that should occur: $\frac{2^{30}}{128} = 2^{23}$. Using the Linux tool `perf`, we confirmed that the number of LLC misses was approximately 2^{23}. However, the number of transactional aborts was extremely small (less than 100), which proves that LLC misses do not necessarily cause transactions to abort. A similar experiment was done to rule out the possibility that *cross-socket* LLC misses cause transactions to abort.

4. Dealing with NUMA

In this section we briefly describe a few unsuccessful approaches to deal with NUMA effects that we tried, and then concentrate on one simple technique that did work.

One approach to deal with NUMA effects is to delegate each operation to the socket on which it should ideally execute (i.e., the socket on which most of its accesses would be local). If a thread gets an operation that should be executed on a different socket, it packages it up and sends it to the other socket. We implemented a number of delegation algorithms, in which we manually decided which socket to send each operation to by checking whether its key fell into the lower or upper half of the key range. The results showed that while delegation doubled performance *per unit of time spent executing delegated operations*, the overhead of involved coordination between threads was too high. This experience echoes the results of previous work, which has shown that the benefit of delegation is often outweighed by the overhead of implementing it [6]. We note that increasing the size of critical sections (or more precisely, the amount of shared data accesses), e.g., by packing and delegating multiple AVL tree operations, helped to extract some benefit out of delegation. In our future work, we plan to explore better delegation techniques and ways to apply them generically and with reduced overheads.

Another approach to deal with NUMA effects is to restrict the concurrency of threads executing on different sockets. For instance, one might allow only a few threads to run on the second socket. However, as Figure 1 shows, even a single thread running on the second socket cripples performance. Alternatively, one might force threads on the second socket to backoff before retrying an aborted transaction. We tried this approach early in our exploration, and we found that performance improved only when the backoff was so long that the second socket was almost completely starved. Although starving the second socket avoids performance degradation beyond 36 threads, it yields poor performance for workloads that scale on two sockets.

The most effective solution we found builds on the observation that, for many workloads, the best performance is achieved either by letting threads run on a single socket (and starving threads on the other socket) or by letting threads run on both sockets. In the remainder of this section, we present an algorithm called TLEStarve, which periodically profiles TLE performance, measuring the number of critical section executions when threads run only on one of the sockets and on both sockets, and decides which configuration is favorable. In particular, if TLEStarve finds out that for some lock, letting threads run only on one socket yields higher performance, it will starve threads running on the other socket (and trying to elide the same lock). Clearly, this approach can be unfair to some of the threads in its goal to achieve higher throughput. In Section 5 we describe how TLEStarve can be improved to impose long-term fairness without loosing performance.

For TLEStarve implementation, we augment each lock with a *mode* that specifies which threads may execute critical sections protected by that lock. In our two-socket system, there are three possible modes: In mode 0 or mode 1, only threads on socket 0 or 1 respectively may execute the critical section (threads on the other socket simply block until the mode changes); in mode 2, threads on either socket may do so. In general, the number of modes is equal to the number of sockets plus one.

To select a mode for each lock, we introduce a new operation, *ProfileAndThrottle*. This operation starts a profiling session that iterates over the lock modes, setting all locks to the current mode, and measuring the total number of lock acquisitions (over all locks) for a short, fixed period of time. Then, it sets each lock to the mode for which it performed best.

As usual with TLE, we co-opt the *LockAcquire* and *LockRelease* operations that bracket the critical section to attempt to elide the lock acquisition and release using transactions. However, a thread executing *LockAcquire* must now first check the mode of the lock to determine whether it is allowed to execute the critical section. Pseudocode for *LockAcquire*, *ProfileAndThrottle*, and their subroutines, appears in Figure 9. (The *LockRelease* operation is the same as usual with TLE.)

The *Lock* type contains the metadata of the original lock implementation (*lockData*), the fastest mode as determined by the most recent profiling session (*fastestMode*), and an *acquisitions* collection, which is used for profiling. This collection stores, for each thread and mode, the number of times the lock was acquired by that thread, in that mode, since the last profiling session began.[2] The data in this collection is aggregated and used to decide which mode is fastest, for each lock.

For simplicity, we assume that an upper bound is known on the number of threads, and we implement *acquisitions* using an array with one slot for each thread and mode. One could eliminate this assumption by using, e.g., a linked list in which each thread maintains its statistics in a separate node.

There are two shared variables: *locksToProfile* is an array of pointers to *Lock*s; *profileStartTime* is a lock whose value is nonzero when a profiling session is in progress. When it is locked, it contains -1, and when unlocked, it contains either 0, or the start time of the last profiling session.

ProfileAndThrottle takes an array, *locks*, of pointers to the locks that should be profiled, as its argument. It begins by attempting to lock *profileStartTime* using compare-and-swap (CAS). If this CAS fails, then a profiling session is already in progress. Otherwise, the *acquisitions* collections of all locks in *locks* are reset to contain all zeros, *locksToProfile* is set to *locks* (so information about which locks are being profiled can be accessed elsewhere in the code), and *profileStartTime* is set to the current time. This has the effect of starting a profiling session.

LockAcquire starts by invoking a subroutine called *getMode* to compute the lock's current mode, and then determines which socket the current thread is running on by invoking a subroutine called *getSocket* (which may be implemented as a system call). It then checks whether the current mode permits the thread to acquire the lock. If so, it increments the appropriate element in the *acquisitions* collection and invokes the lock acquisition procedure provided by the underlying TLE implementation (*LockAcquireTLE*). Otherwise, it retries from the beginning.

getMode begins by getting the time when the last profiling session began (*startTime*), and recording the current time (*now*). It then computes how many lock modes have been profiled since *startTime*, and uses this information to determine whether the profiling session that began at *startTime* is ongoing. If so, the current mode is simply returned. Suppose, however, that the profiling session which began at *startTime* is finished. To make use of the results collected during profiling, some thread must compute the best mode for each lock by invoking a subroutine called *computeBestLockModes*. *getMode* reserves the right to invoke *computeBestLockModes* by locking *profileStartTime*, and then does so. *computeBestLockModes* needs only be invoked once each time a profiling session finishes. So, before *getMode* locks *profileStartTime* and invokes *computeBestLockModes*, it verifies that *startTime* is greater

[2] Technically, each member of the *acquisitions* collection is an *upper bound* on the number of lock acquisitions by a given thread in a given mode. The elements are reset to zero non-atomically in preparation for a new profiling session, just before the profiling session begins. Consequently, some elements may be incremented after being set to zero, but before the new profiling session starts. Any error in these elements affects throttling decisions, but does not affect correctness.

```
type Lock {
  lock_t lockData; // original lock metadata

  // fields added for ProfileAndThrottle
  long acquisitions[][];
    // acquisitions[i][m] = number of lock
    // acquisitions for thread i and mode m
  int fastestMode;
};

Lock ** locksToProfile;
  // set of locks currently being profiled

long profileStartTime = 0;
  // lock guarding metadata for all instances of Lock
  //  0: unlocked and not currently profiling
  // >0: unlocked and currently profiling
  // -1: locked (and not currently profiling)

int getMode(Lock * lock) {
  long startTime = profileStartTime;
  long now = getCurrentTime();
  int mode = (now - startTime) / PROFILE_MODE_TIME;
  if (mode < NUM_MODES) { // if still profiling
    return mode;
  } else {            // profiling is finished
    if (startTime > 0) {
      // if the best mode for each lock has not been
           determined, try to lock profileStartTime
      if (CAS(&profileStartTime, startTime, -1)) {
        computeBestLockModes();
        profileStartTime = 0; // unlock
    } }
    return lock->fastestMode;
} }

void computeBestLockModes() {
  for each lock in locksToProfile {
    // compute fastest mode for lock
    long acqs[];
    for (int m = 0; m < NUM_MODES; m++) {
      // sum acquisitions in mode m for all threads
      acqs[m] = 0;
      for (int j = 0; j < NUM_THREADS; j++) {
        acqs[m] += lock->acquisitions[j][m];
    } }
    lock->fastestMode = index of largest elem of acqs
} }

int LockAcquire(Lock * lock) {
  while (true) {
    // check if we are allowed to acquire the lock
    int mode = getMode(lock);
    if (mode == NUM_MODES-1 || mode == getSocket()) {
      ++lock->acquisitions[getTid()][mode];
      return LockAcquireTLE(lock);
} } }

void ProfileAndThrottle(Lock ** locks) {
  // try to lock profileStartTime
  if (CAS(&profileStartTime, 0, -1)) {
    for each lock in locks {
      set all entries of lock->acquisitions to 0
    }
    locksToProfile = locks;
    profileStartTime = getCurrentTime(); // unlock
} }
```

Figure 9. Pseudocode for *ProfileAndThrottle*, *LockAcquire* and their subroutines.

than zero. If not, then another thread already invoked *computeBestLockModes* and set *profileStartTime* to zero before it was read by this invocation of *getMode*. It can be shown that, if *getMode* fails to lock *profileStartTime*, then another thread has already locked *profileStartTime* to invoke *computeBestLockModes*. The implementation of *computeBestLockModes* itself is straightforward.

As an optimization, one can invoke *getMode* and *getSocket* only once per C invocations of *LockAcquire*, and the results of the last invocation of each can be saved in thread local variables. For instance, in our experiments we invoke these subroutines only once per 128 invocations of *LockAcquire*.

Naturally, the success of profiling depends on the workload being somewhat homogeneous. The *ProfileAndThrottle* operation allows a programmer who knows that the program is about to begin a homogeneous workload to manually trigger profiling, optimizing performance with minimal effort. However, there are other ways that *ProfileAndThrottle* could be used. For instance, if a programmer knows that a workload consists of one or more relatively long, homogeneous phases, but has no compile-time knowledge of when phases begin or end, s/he could simply invoke *ProfileAndThrottle* periodically. Alternatively, *ProfileAndThrottle* could be invoked internally by *LockAcquire*, without any modification to user code. The set of locks passed to *ProfileAndThrottle* could be maintained internally using static variables, by having each lock register itself the first time it is passed to an invocation of *LockAcquire*.

5. Fairness

As described thus far, TLEStarve may cause threads on one socket to starve: even if *ProfileAndThrottle* is called repeatedly, it may always select mode 0, for example, in which case threads on socket 1 will never get to execute the critical section.

Starving threads may be acceptable in certain scenarios, e.g., when each thread acquires a global lock, retrieves a job from a work queue, performs it, and releases the lock. There, blocking threads on the second socket does not impede the progress of threads on the first socket, and there is no work that can be done *only* by threads on the second socket. However, in many scenarios, it is desirable to provide some form of fairness, so that no threads starve.

We address this problem by introducing *time sharing* to allow threads on both sockets to run. Recall that, in TLEStarve, each lock stores the mode wherein it performs best. In this new algorithm, TLEShare, each lock additionally stores its *second best* mode. Conceptually, we divide the execution into fixed time *quanta* (e.g., 10 milliseconds), and each lock spends some fraction of the quantum executing in its fastest mode, and the remaining time executing in its second fastest mode[3].

The amount of time given to each of these modes represents a trade-off between achieving fairness and ideal single-socket performance. One seemingly reasonable policy is to give each mode a time slice that is proportional to the performance of the lock in that mode. However, a particularly interesting case arises in, for example, the update-heavy workload in Figure 1, when there are 36 threads on the first socket, and one thread on the second socket. In this case, the best mode is to throttle one socket, and the second best mode is to let both sockets run. Since the throughput when both sockets run is approximately half of the throughput when a single socket runs, approximately one third of the execution time is allocated to let both sockets run. However, since the only reason we time share in this case is to avoid starving the single thread on the second socket, one-third is too much time to allocate to run both sockets. We avoid this problem by expressing the time slices we give to the fastest and second fastest modes as a function of the performance of the lock in modes where only a *single socket* can run. Thus, we would allocate approximately 1/36th of the execution time to let both sockets run. The pseudo-code for the TLEShare implementation is provided in Appendix A.

The length of a time quantum is another factor in the trade-off between achieving fairness and ideal single-socket performance.

[3] More precisely, we do so only if the fastest mode requires only one socket. Otherwise, we use both sockets for the whole duration of the quantum.

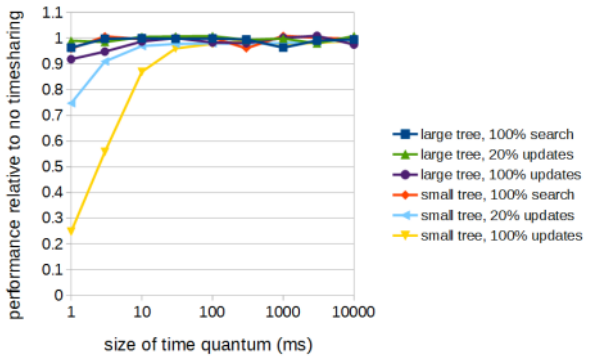

Figure 10. Overhead caused in six different workloads by the choice of time quantum length. All data points represent the performance of TLEShare, relative to TLEStarve, with 72 threads.

Shorter time quantums promote a higher degree of fairness, but introduce greater overhead, as the sockets pollute each others' caches more frequently. Figure 10 shows the results of a simple experiment that we ran to illustrate the overhead introduced by different quantum lengths in six different workloads. The results show that there is little overhead when time quanta are at least 30 milliseconds. Additionally, smaller quanta appear to be feasible, except under high contention.

6. Experimental Results

In this section, we present a selection of results from a large suite of microbenchmarks. We designed three different implementations of the set ADT: an AVL tree, an unbalanced binary search tree (BST), and a skip-list. We then paired each data structure with TLE, TLEStarve, TLEShare, a recent TLE implementation called TLESCM that performs contention management using an auxiliary lock for conflicting transactions [2], and a new variant of TLESCM called TLEShareSCM that uses *ProfileAndThrottle*. The latter is interesting, because it demonstrates that our technique can be applied to different TLE algorithms. TLEStarve, TLEShare and TLEShareSCM profile each lock mode for 30 milliseconds before moving on to the next mode. TLEShare and TLEShareSCM use 100 millisecond quanta for time sharing. That is, each lock changes modes twice every 100 milliseconds.

Each data point in our graphs is an average of five timed trials, each lasting approximately 10 seconds. In each trial, a fixed number of threads repeatedly perform random operations according to some workload, on uniformly random keys drawn from a fixed key range. Before a trial begins, the data structure is prefilled so that it contains half of its key range. For TLEStarve, TLEShare and TLEShareSCM, *ProfileAndThrottle* is invoked by the main thread approximately 50 milliseconds after a trial starts (to give all threads time to begin working and warm up their caches before profiling begins). Profiling is done only once per trial. We note that our experiments (not included) show that even if profiling is done more frequently, e.g., every second, it has a negligible cost on performance. This makes our techniques suitable also for workloads that are not completely homogenous. In all experiments, we use an HTM-friendly memory allocator [11]. To reduce noise from the power management system, the machine was set up in performance mode (i.e., the power governor was disabled, while all cores were brought to the highest frequency) with turbo mode disabled.

We first discuss the results for AVL trees, which appear in Figure 11. For the read-only workload, all algorithms scale, for both key ranges [0, 2048) and [0, 131072). The reductions in slope at 18-36 threads and 54-72 threads occur because of hyperthreading.

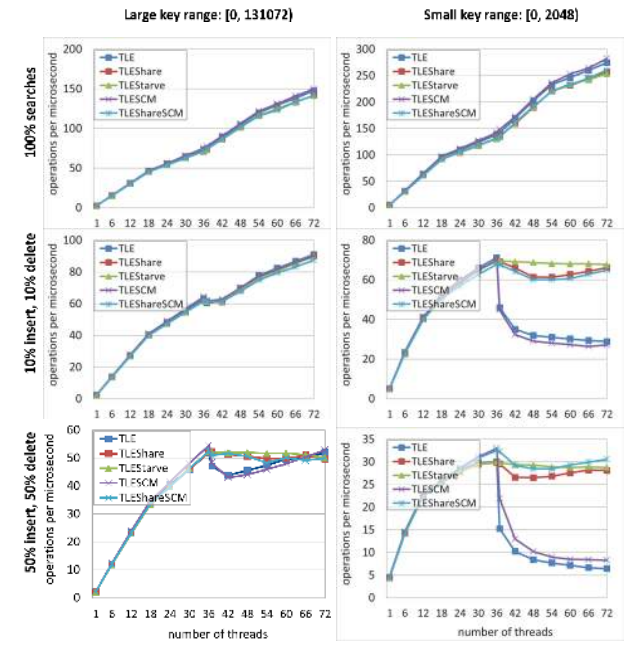

Figure 11. Experimental results for AVL trees.

(For thread counts 1 through 18, each core on the first socket runs a single thread. However, for thread counts 19-36, each core runs two threads. Similar remarks hold for the second socket.)

The scaling worsens as the number of updates increases. In the small tree, TLE already fails to scale on two sockets with only 20% updates, as there is a higher chance of conflicts, and operations complete more frequently, so more cache invalidations occur. Scaling is generally better in the large tree. In all cases, however, our new algorithms take full advantage of two sockets when workloads scale, and have little or no performance degradation above 36 threads for workloads that do not scale. TLESCM and TLEShareSCM sometimes perform slightly better than TLE and TLEShare, respectively, and sometimes perform slightly worse. However, the performance differences are fairly small.

Due to lack of space, the results for experiments with unbalanced BSTs and skip lists are provided in Appendix B. At a high level, when contention is high, TLE performs in skip lists similarly to AVL trees, degrading substantially once the number of threads exceeds the capacity of one socket; TLEShare is able to avoid this degradation. Along with that, TLE performance in unbalanced BSTs is less prone to the NUMA effects as operations do not rotate the tree and thus always modify only nodes at or near tree leaves. Yet, in the most contended workload (small tree with 100% update operations), TLE performs twice worse at 72 threads compared to 36 threads, while the performance of TLEShare at 36 and 72 threads is similar.

All of the graphs discussed up to this point in the paper have shown results for experiments where the first 36 threads are pinned to the first socket, and the next 36 are pinned to the second socket. Other thread pinning policies are possible, and we ran all experiments with several different policies. Due to lack of space, we only include graphs for two alternative policies with 100% updates workload and key range $[0, 2048)$. These graphs appear in Figure 12. In the left graph, threads are pinned to alternating sockets, i.e., threads 0, 1, 2 and 3 are pinned to sockets 0, 1, 0 and 1, respectively. In the right graph, threads are not pinned, and the operating system decides placement. The similarity between these two graphs suggests that the Linux scheduler attempts to evenly dis-

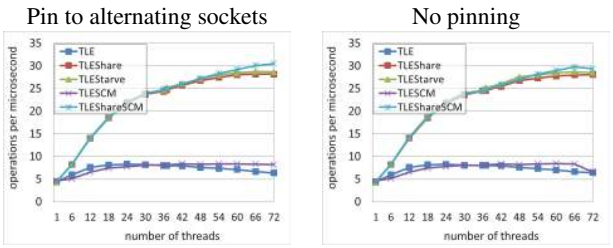

Figure 12. Experimental results for AVL trees with different thread pinning policies. The workload is 100% updates with key range $[0, 2048)$.

tribute the load across sockets. Since 100% updates workload with key range $[0, 2048)$ does not scale on two sockets, the benefit of TLEShare and TLEStarve is much more significant, and becomes evident much earlier (starting at just two threads).

We now describe a pair of experiments with more sophisticated workloads, involving multiple AVL trees.

Smart tree & dumb tree. In this experiment, there are two AVL trees (a so-called *smart tree* and *dumb tree*) and three independent groups of threads performing operations. The dumb tree is accessed by one group of threads, which contains half of the total number of threads. Each thread performs 100% updates workload with key range $[0, 2048)$. Half of the threads in this group (one quarter of the total number of threads) are pinned to the first socket, and the other half are pinned to the second socket. Thus, for example, in an execution with a total of 8 threads, the dumb tree is accessed by four threads, two of which are pinned on the first socket, and two of which are pinned on the second socket. As we saw in Figure 11, the dumb tree workload does not scale on two sockets.

The smart tree is accessed by two groups of threads, each containing one quarter of the total number of threads. The first group is pinned to the first socket, and operates on keys in the key range [0, 1024). The second group is pinned to the second socket, and operates on keys in the key range [1024, 2048). Since threads in the smart tree essentially operate in two separate subtrees, the workload scales on two sockets. Thus, the optimal decision is to throttle the dumb tree to run on a single socket, and allow the smart tree to run on both sockets.

Performance results for this experiment appear in the left half of Figure 13. Both TLEStarve and TLEShare make correct decisions, and perform well. The graphs in the bottom two rows of Figure 13, which break out performance by data structure, show that the poor scaling of TLE is the result of flat performance of the dumb tree (as demonstrated in Figure 12). At the same time, TLEShare is able to achieve scalability for both trees.

The previous experiment demonstrates that our technique can help when a programmer knows how to partition data so that different parts are accessed by different sockets. The following experiment shows that, even without this knowledge, our technique is sufficiently intelligent to use two sockets for one AVL tree whose workload just happens to scale, and only one socket for another AVL tree whose workload does not.

Small updates & big searches. In this experiment, there are two AVL trees and two independent thread groups. The first thread group contains half of the threads, and each thread in this group performs 100% updates workload with key range $[0, 2048)$ on the first tree. This workload does not scale on two sockets. The second thread group contains the other half of the threads, which each performs 100% searches with key range [0, 131072) on the second tree. Since threads in the second group do not cause any cross-socket cache invalidations, this workload scales on two sockets (as we saw in Figure 11). Thus, the optimal decision is to throttle

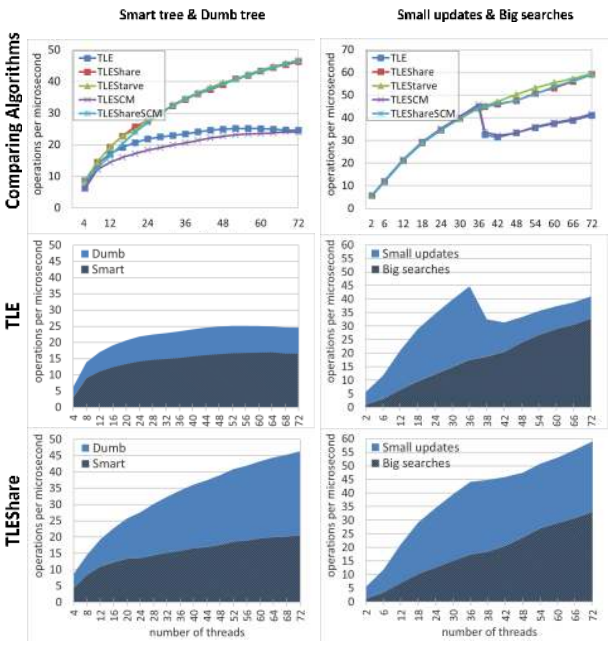

Figure 13. Experimental results for workloads involving multiple AVL trees. The top two graphs compare the performance of the TLE algorithms. The bottom four graphs each show a single TLE algorithm, and break out the performance of *each data structure*. The area graphs are *stacked*, meaning that the top of the upper area represents the aggregate performance for both trees.

threads running on one socket and accessing the smaller tree, and allow threads accessing the larger tree to run on both sockets.

Performance results for this experiment appear in the right half of Figure 13. For thread counts up to 36, all threads are pinned to the first socket. For larger thread counts, the first 36 threads are pinned to the first socket, and the remaining threads are pinned to the second socket. As in the previous experiment, both TLEStarve and TLEShare make the optimal decision, and both significantly outperform TLE above 36 threads. The graphs that break out performance by data structure show that the drop at 38 threads for TLE occurs specifically because of a drop in the performance of the smaller tree.

7. Conclusion

In this paper, we have presented results of experiments we have done in an investigation into the behavior of HTM on a large 72-thread dual-socket machine. We have shown that some recommendations and common usage patterns for HTM on smaller single-socket machines do not carry over to larger machines. In particular, the NUMA characteristics of the multi-socket machine can have dramatic effects on the performance of applications that use HTM: for some applications, using all 72 threads on the machine yields behavior that is only marginally better than that of single-threaded execution. On the other hand, other applications may scale to the full capacity of the machine.

Based on these observations, we proposed a technique for making effective use of HTM on large NUMA machines by adaptively throttling threads as necessary to optimize performance based on profiling information collected during execution. Our experiments show that our technique achieves the full performance of both sockets for workloads that scale, and avoids the performance degradation that cripples TLE for workloads that do not.

References

[1] G. Adelson-Velsky and E. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] Y. Afek, A. Levy, and A. Morrison. Software-improved hardware lock elision. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 212–221, 2014.

[3] E. Atoofian. Improving performance of software transactional memory through contention locality. *J. Supercomput.*, 64(2):527–547, 2013.

[4] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[5] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. *SIGOPS Oper. Syst. Rev.*, 23(5):19–31, Nov. 1989.

[6] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. J. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *Proceedings of the International Conference on Principles of Distributed Systems OPODIS*, pages 83–97, 2013.

[7] G. Chadha, S. Mahlke, and S. Narayanasamy. When less is more (LIMO): Controlled parallelism for improved efficiency. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 141–150, 2012.

[8] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the International Conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 157–168, 2009.

[9] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical report, Sun Labs, 2009.

[10] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir. Adaptive integration of hardware and software lock elision techniques. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 188–197, 2014.

[11] D. Dice, T. Harris, A. Kogan, and Y. Lev. The influence of malloc placement on TSX hardware transactional memory. *CoRR*, 2015.

[12] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: A general technique for designing NUMA locks. *TOPC*, 1(2):13, 2015.

[13] N. Diegues and P. Romano. Self-tuning Intel transactional synchronization extensions. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 209–219, 2014.

[14] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 3–14, 2014.

[15] N. Diegues, P. Romano, and S. Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *To appear in Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[16] M. Herlihy and E. Moss. Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*, 1993.

[17] R. P. LaRowe, Jr., C. S. Ellis, and M. A. Holliday. Evaluation of NUMA memory management through modeling and measurements. *IEEE Transactions on Parallel and Distributed Systems*, 3:686–701, 1991.

[18] B. Lepers, V. Quema, and A. Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the USENIX Annual Technical Conference*, 2015.

[19] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2007.

[20] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, 2012.

[21] A. Matveev and N. Shavit. Reduced hardware lock elision. In *Proceedings of 6th Workshop on the Theory of Transactional Memory (WTTM)*, 2014.

[22] T. Nakaike, R. Odaira, M. Gaudet, M. Michael, and H. Tomari. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[23] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 116–125, 2011.

[24] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using dope: The degree of parallelism executive. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37, 2011.

[25] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 188–197, 2013.

[26] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. *SIGOPS Oper. Syst. Rev.*, 30(5):279–289, 1996.

[27] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 169–178, 2008.

[28] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

```
int getMode(Lock * lock) {
  long startTime = profileStartTime;
  long now = getCurrentTime();
  int mode = (now - startTime) / PROFILE_MODE_TIME;
  if (mode < NUM_MODES) { // if still profiling
    return mode;
  } else {            // profiling is finished
    if (startTime > 0) {
      // if the best mode for each lock has not been
           determined, try to lock profileStartTime
      if (CAS(&profileStartTime, startTime, -1)) {
        computeBestLockModes();
        profileStartTime = 0; // unlock
    } }
    // compute time elapsed in the current quantum
    long quantumElapsed = now % QUANTUM;
    if (quantumElapsed > lock->fastestModeSlice) {
      return lock->alternateMode;
    }
    return lock->fastestMode;
} }

void computeBestLockModes() {
  for each lock in locksToProfile {
    // compute fastest mode for lock
    long acqs[];
    for (int m = 0; m < NUM_MODES; m++) {
      // sum acquisitions in mode m for all threads
      acqs[m] = 0;
      for (int j = 0; j < NUM_THREADS; j++) {
        acqs[m] += lock->acquisitions[j][m];
    } }
    lock->fastestMode = index of largest element of
         acquisitions
    lock->alternateMode = index of second largest
         element of acquisitions

    if (lock->fastestMode == NUM_MODES-1) {
      // the fastest mode lets both sockets run, so
           there's no point in alternating modes.
      lock->fastestModeSlice = QUANTUM;
    } else {
      // the fastest mode lets a single socket run.
           We divide the quantum between fastestMode
           and alternateMode according to the solo
           performance of each socket.
      lock->fastestModeSlice = QUANTUM *
          acqs[lock->fastestMode] /
            (acqs[lock->fastestMode] +
             acqs[1 - lock->fastestMode]);
} } }
```

Figure 15. Pseudocode for time sharing between lock modes.

A. Implementation details for TLEShare

```
type Lock {
  lock_t lockData; // original lock metadata

  // fields added for ProfileAndThrottle
  long acquisitions[][];
    // acquisitions[i][m] = number of lock
    // acquisitions for thread i and mode m
  int fastestMode;
  int alternateMode;
  long fastestModeSlice;
    // time slice out of each quantum for which
    // the lock mode should be fastestMode
};
```

Figure 14. *Lock* data type for time sharing between modes.

Pseudocode for TLEShare appears in Figure 14 and Figure 15. The implementations of *ProfileAndThrottle*, *LockAcquire* and *LockRelease* are omitted, since they are the same as in TLEStarve.

computeBestLockModes is similar to the implementation for TLEStarve, but it also saves the second best mode (in the variable

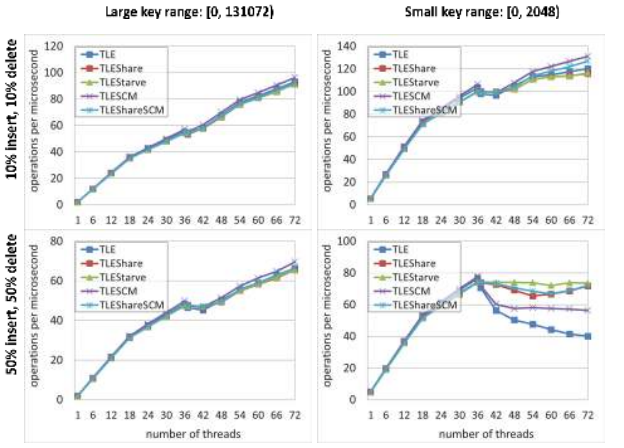

Figure 16. Experimental results for unbalanced BSTs.

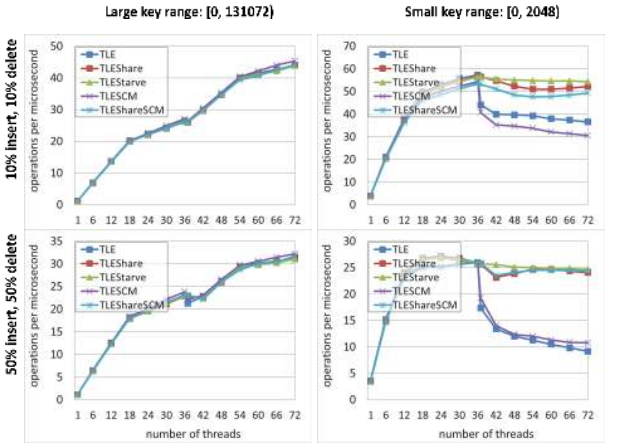

Figure 17. Experimental results for skip-lists.

alternateMode), and computes how much of each time quantum should be allocated to the fastest mode. If the fastest mode is to let both sockets run, then the lock simply lets both sockets run, and does not alternate modes. Otherwise, it decides how much time to give to the fastest mode using the number of lock acquisitions in modes zero and one that were recorded in the last profiling phase.

getMode is similar to the implementation for TLEStarve except that instead of simply returning the fastest mode, it uses the current time to determine whether the lock is currently in its fastest mode, or its second fastest mode.

B. Additional experimental results

In this section, we present additional experimental results with other implementations of the set ADT. Figure 16 shows the results for the unbalanced BST microbenchmark. The graphs for the 100% searches workload are omitted as they are similar to those for the AVL tree.

For the unbalanced BST, we see much less performance degradation for TLE when threads run on both sockets. In fact, TLE continues to scale on two sockets for 20% updates workload with the small key range, where it plummeted for the AVL tree. Even 100% updates workload with the large key range scales beyond one socket. As we mentioned in Section 3, this is in line with our hypothesis about cross socket cache invalidations. In 100% updates workload with key range $[0, 2048)$, TLESCM has a significant per-

formance advantage over TLE. Despite this, TLEShareSCM and TLEShare achieve approximately the same performance. One plausible explanation is that the concurrency restrictions imposed by TLESCM to improve performance become irrelevant when threads running on one of the sockets are throttled.

Next, we discuss the results for the skip-list, which appear in Figure 17. Similarly to the unbalanced BST, the graphs for the 100% searches workload are omitted. The skip-list performs more similarly to the AVL tree for the small key range, while it is more similar to the unbalanced BST with the large key range. The performance degradation of TLE when the second socket is in use is slightly less severe than for the AVL tree, but it is still debilitating. TLEShare is able to avoid this degradation.