Investigating the Performance of Hardware Transactions on a Multi-Socket Machine

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

Alex Kogan
Oracle Labs
alex.kogan@oracle.com

Yossi Lev
Oracle Labs
yossi.lev@oracle.com

Victor Luchangco
Oracle Labs
victor.luchangco@oracle.com

ABSTRACT

The introduction of hardware transactional memory (HTM) into commercial processors opens 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, 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 two-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. Using extensive evaluation of multiple microbenchmarks and real applications, we demonstrate that our technique achieves the full performance of the system for workloads that scale across sockets, and avoids the performance degradation that cripples TLE for workloads that do not.

Keywords

hardware transactional memory, nonuniform memory access, lock elision, concurrent data structures, locks

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) [20] to support efficient synchronization, and previous work has shown that it can be used effectively [2, 12, 16–18, 26, 34]. However, until recently, HTM has been available only on relatively small single-socket systems. For this paper, we investigated the behavior of HTM on a large multi-socket machine and observed that its behavior differs from that of smaller systems in ways that present challenges for scaling the performance on the larger machine.

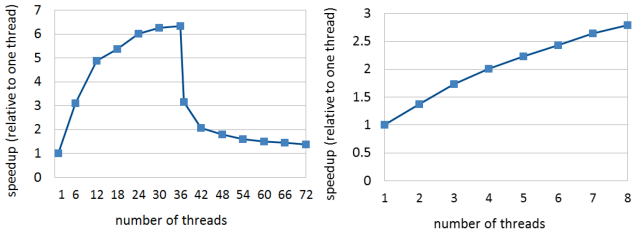


Figure 1: AVL tree microbenchmark on a large HTM system (left), and a small HTM system (right). The workload is 100% updates with key range $[0, 2048)$.

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 repeatedly insert and delete nodes in an AVL tree [1]. Operations on the tree are protected by a single lock, to which we apply *transactional lock elision* (TLE) [13], a popular technique for exploiting HTM in which lock-based critical sections are executed concurrently using hardware transactions, implemented on top of the Intel Haswell TSX/RTM interface. The machine has two sockets, each with 18 hyperthreaded 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, though the scaling is more moderate after 12 threads. However, as soon as any thread executes on the second socket, 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. These results are in stark contrast to those shown on the right, from a similar experiment on a smaller single-socket machine (4 hyperthreaded cores), whose performance continues to improve until the machine is saturated.

Not all benchmarks exhibit this pathology: with only lookup operations (i.e., a read-only workload), for example,

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

SPAA '16, July 11-13, 2016, Pacific Grove, CA, USA

DOI: http://dx.doi.org/10.1145/2935764.2935796

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. We implemented this idea entirely in a library providing a lock API, thus requiring no changes to the code that uses traditional lock-based synchronization. Using multiple microbenchmarks and a number of real applications, including STAMP [27], ccTSA [4] and paraheap-k [21], 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 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 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 us 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) [13] implemented on top of the Intel Haswell TSX/RTM interface 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. (To avoid executing concurrently with a critical section executed under lock, a transaction executing a critical section must check that the associated lock is free and abort otherwise.) If the transaction aborts and the hint bit is not set (typically, if the abort is due to overflow), it is common to not retry at all [3, 16].

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 [12, 18, 26, 28, 34]. 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 [12, 16] and introducing auxiliary locks [2, 17]. 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, conflicting 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 [6–8, 22, 23, 32, 33]. 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. [15], 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 [6, 24] or data migration and replication [33].

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 [9, 29, 30], 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 [5,35]. In the context of TLE, Diegues et al. [17] suggest using *core locks*, which synchronize between hardware transactions running on the 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 [15] are perhaps most relevant to the concur-

rency 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. [25] 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. [8] 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. In the context of software transactional memory, Hassan et al. [19] investigate the idea of dedicating a number of server threads to perform commit phase of software transactions executed by client threads.

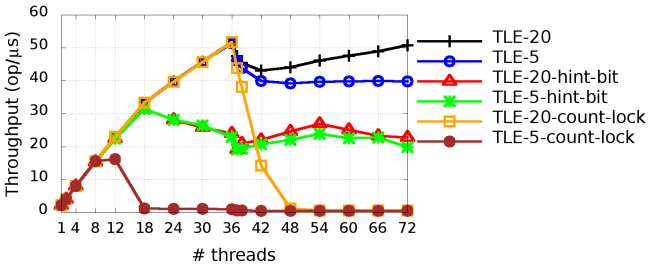
3. HTM ON A LARGE MACHINE

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 Section 3.1, we make observations that are not due to NUMA effects, but rather to the larger number of cores in our system. Section 3.2 details findings that are more NUMA-specific.

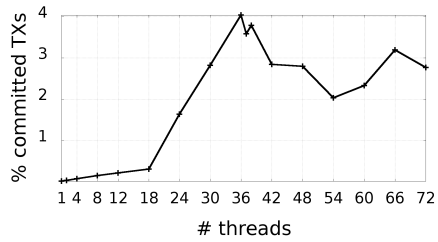
In most of these experiments, threads repeatedly invoke insert, delete and lookup operations on implementations of an abstract set (e.g., an AVL tree) using a key selected uniformly at random from a specified key range. The set is prefilled with approximately half the keys. We specify the percentage 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 otherwise specified, we pin the threads to cores so that the first 18 threads are executed by different cores on the first socket, and the next 18 are executed with hyperthreading on the same cores. (Thus, all threads execute on a single socket when there are 36 or fewer threads.) The next 36 threads are pinned similarly to cores on the second socket. (We discuss alternative pinning policies in Section 5.)

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(a) 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 concurrent operations are unlikely to conflict.) *TLE-5-hint-bit* 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 by another thread (in which case the transaction is not retried until the lock is released in order to avoid the *lemming effect* [14]). However, if a transaction aborts and the hint bit is not set by hardware, we fall back to the lock immediately (recall that this bit is set if,



(a) TLE using different retry policies



(b) Percent of transactions in *TLE-20* that commit after at least one failure with hint bit not set

Figure 2: An AVL tree on a large HTM system. The workload is 100% updates with key range $[0, 131072)$.

according to the hardware, the transaction is likely to succeed if retried). *TLE-20-hint-bit* 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 ignore the hint bit (i.e., they do not fall back to the lock immediately when a transaction aborts and the hint bit is not set). Finally, *TLE-5-count-lock* and *TLE-20-count-lock* also ignore the hint bit, but they do count attempts that fail because the lock is held by another thread.

Perhaps surprisingly, contrary to the common belief that a transaction that fails with the hint bit not set (which in most cases happens due to overflow) will continue to fail if it is retried, we observed that subsequent attempts may succeed. Thus, falling back immediately when a transaction aborts with the hint bit not set caused threads to fall back to the lock unnecessarily. Eliminating this "optimization" significantly improves performance. (Compare, for example, the curves for *TLE-20* and *TLE-20-hint-bit*.) This improvement shows up after 18 threads, suggesting that hyperthreading may cause transient overflow failures. Indeed, the percentage of transactions that commit after a previous attempt failed with the hint bit not set, shown in Figure 2(b), rises sharply when there are more than 18 threads. (The graph for transactions committing after a previous attempt failed due to overflow is similar.) Although this percentage never exceeds 4%, this result emphasizes the greater cost of falling back to the lock on a system running more than a dozen threads: Because a thread that takes the lock blocks every other thread, in a large system, it pays to tolerate more failed transactions to avoid taking the lock.

Allowing 20 transactional attempts rather than only 5 (compare the curves for *TLE-20* and *TLE-5*) also improves performance, though to a lesser extent. To determine whether raising the limit on transactional attempts beyond 20 might improve performance even more, we experimented with vary-

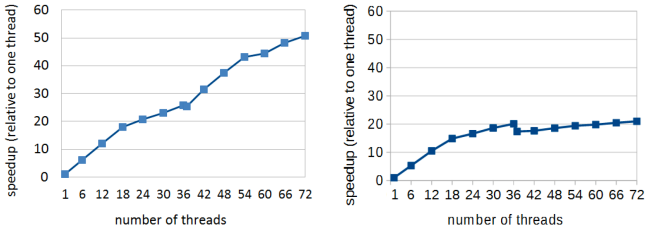


Figure 3: An AVL tree on a large HTM system. The workload is 100% lookup on the left, and 2% updates (i.e., 98% lookups) on the right, with key range $[0, 2048)$.

ing the limit over a wide range, up to hundreds of millions of retries, and 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*.

The curves for *TLE-5-count-lock* and *TLE-20-count-lock* show the importance of the anti-lemming-effect optimization. *TLE-5-count-lock* scales well up to 8 threads, the typical capacity of small systems, but collapses after just 12 threads. Allowing more retries in *TLE-20-count-lock* delays this collapse, but nonetheless, more data conflicts arise as the number of threads grows, leading more threads to acquire the lock. This in turn causes the lemming effect, which hurts performance.

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 lightly contended, and most of them succeed in HTM on the 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 3).

Although NUMA effects are well known to negatively impact performance, this impact is amplified by the use of HTM. Figure 4 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. With 36 threads, the algorithm with no synchronization is 12.1x faster than with a single thread, but only 8.9x faster with 72 threads, a 26% decrease in performance. However, the TLE algorithm experiences a much larger (75%) drop in performance when going from 36 to 72

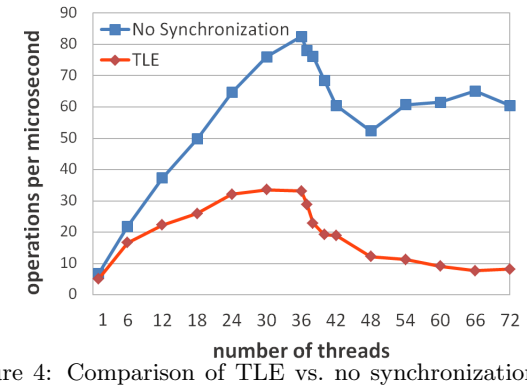


Figure 4: Comparison of TLE vs. no synchronization in a workload doing search-and-replace operations on an AVL tree with key range $[0, 4096)$.

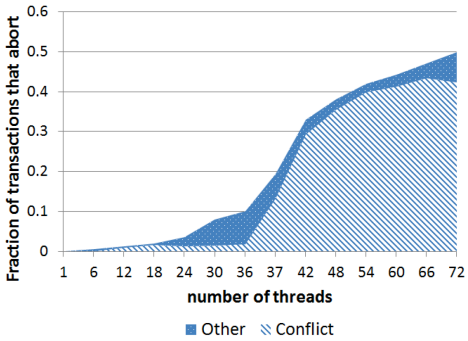


Figure 5: Abort rate for the TLE curve in Figure 4. (Note: the x-axis is not to scale near x=37).

threads (6.4x faster than a single thread with 36 threads vs. 1.6x faster with 72 threads).

To understand why TLE suffers so greatly from NUMA effects, consider Figure 5, 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.

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 more quickly 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 check our hypothesis, we ran a 36-thread *single-socket* experiment that added some artificial delay (spinning) just

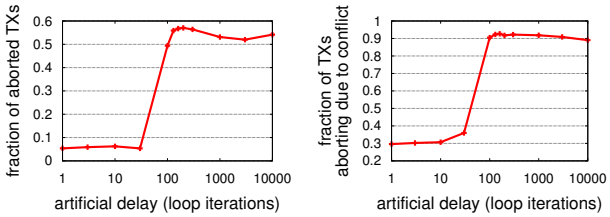


Figure 6: Abort rate (left) and fraction of transactions aborting due to conflict (right) for a 36-thread single-socket experiment in which delay is inserted before committing each transaction. The workload is 100% updates in an AVL tree with key range $[0, 131072)$.

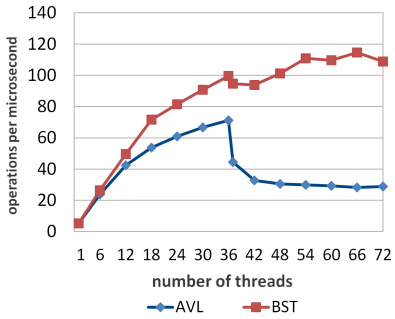


Figure 7: Comparison between an AVL tree and a leaf-oriented BST, with 20% updates and key range $[0, 2048)$.

before committing each transaction.[1] The results in Figure 6 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 increases, most transactions that abort do so because of conflicts, mimicking the phenomenon in Figure 5. 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 7, 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. Because this array is too large to fit in cache, 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, almost all the $\frac{2^{30}}{128} = 2^{23}$ reads should incur a LLC miss, and we confirmed (using the Linux tool `perf`) that the number of LLC misses was approximately 2^{23}. However, there were fewer than 100 transactional aborts, which proves that LLC misses do not necessarily cause transactions to abort. We did a similar experiment to rule out the possibility that *cross-socket* LLC misses cause transactions to abort.

[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 43μs (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.

4. DEALING WITH NUMA

4.1 Considered approaches

In this section, we briefly describe a few unsuccessful approaches we tried to deal with NUMA effects, 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 AVL tree 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 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 [8]. We were able to extract some benefit from delegation by packing multiple operations into a single critical section (reducing the relative overhead due to the increased size of each critical section). In 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 can cripple 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 when only threads on one socket are allowed to execute (i.e., starving threads on the other socket) or when all threads regardless of socket are allowed to execute. We exploit this observation by periodically profiling TLE performance to determine, for each lock L, whether it is better to allow any thread to execute critical sections protected by L or to restrict concurrency to threads on one socket at a time. The results of this profiling is used to guide when a thread is allowed to execute critical sections outside

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

  long acquisitions[][];
    // acquisitions[i][m] = #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
  long  lastProfStart = 0;
    // when this lock was profiled for the last time
}
```

Figure 8: *Lock* data structure.

of the profiling phase. In particular, if we determine that it is better to restrict concurrency, then after profiling, we cycle through the sockets, allowing only threads from one socket at a time to execute critical sections protected by L, with the amount of time allocated to each socket determined by their relative performance as measured during the profiling. Critically, throttling decisions are made on a per-lock basis, and as we demonstrate in Section 5, our solution may choose to use all available sockets for some locks, and alternate between sockets for others. In the remainder of this section, we provide a high-level description of our implementation, which we call NATLE (for NUMA-aware TLE).

4.2 NATLE implementation

For simplicity, we assume a two-socket system, but it is straightforward to extend this implementation to systems with more sockets. NATLE allows performance tuning through a number of configurable parameters (such as the number of TLE attempts, the length and the frequency of the profiling period, etc.), which we currently set to fixed values that work reasonably well for our benchmarks. Of course, some applications may benefit by using different settings. Dynamically adapting these settings is left for future work.

For the NATLE 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 are delayed 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.

The running time is conceptually divided into cycles; each cycle is composed of a profiling phase and a post-profiling phase. The profiling phase is equally divided between different modes. The post-profiling phase is equally divided into quanta, and each quantum is split (based on relative performance during profiling) between two modes corresponding to two sockets (or treated as one unit if the favorable mode of operation found during profiling is using both sockets). In our implementation, the length of the profiling time in each cycle was 30ms; thus, each mode was profiled for 10ms. The length of each quantum was set to the length of the profiling time (30ms), and we used 9 quanta per cycle. Thus, the length of each cycle was 300ms, with 10% of this time allocated to profiling.

Figure 8 provides details on the *Lock* data structure maintained by NATLE. The *Lock* data structure contains the metadata of the original (pthread) lock implementation (*lockData*), the start time of the last profiling phase (*lastProfStart*), the fastest mode (i.e., the mode in which the most critical sections were executed) as determined by the most recent profiling session (*fastestMode*), the relative length of the quantum allocated to the fastest mode (*fastestModeSlice*) and an *acquisitions* collection, which is used for profiling. The last two bits in the *lastProfStart* field identify the stage of the profiling phase: 0 indicates that profiling is on but the profiling data (stored in *acquisitions*) has not yet been initialized; 1 indicates that profiling is on and the data has been initialized; 2 indicates that profiling is done but the profiling data has not yet been aggregated; 3 is when the profiling done and the profiling data has been aggregated. In the pseudocode, `S(x)` denotes the last two bits of `x`, and the tuple `<x,y>` denotes the value obtained by setting the last two bits of `x` to `y`.

```
int LockAcquire(Lock *lock) {
  int repetitions=0;
  while (repetitions++ < REPETITIONS_THRESHOLD) {
    // check if we are allowed to acquire the lock
    int mode = getMode(lock);
    if (mode == NUM_MODES-1 || mode == getSocket()) {
      ++lock->acquisitions[getTid()][mode];
      // call the underlying TLE implementation
      return LockAcquireTLE(lock);
    }
    wait for a while
  }
  // call the underlying TLE implementation
  return LockAcquireTLE(lock);
}

int LockRelease(Lock *lock) {
  // just call the underlying TLE implementation
  return LockReleaseTLE(lock);
}
```

Figure 9: LockAcquire and LockRelease.

The *acquisitions* collection stores, for each thread and mode, the number of times the critical section was executed by that thread in that mode since the last profiling phase began. In each cycle, when the profiling phase is finished, the data in this collection is aggregated and used to decide which mode is fastest and for how long it should be used, 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.

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. The details of these operations as implemented in NATLE are given in Figure 9. While the *LockRelease* operation in NATLE is simply a wrapper for the underlying TLE implementation, a thread executing *LockAcquire* must now first check the mode of the lock to determine whether it is allowed to execute the critical section. This is done by calling the auxiliary *getMode* function (see Figure 10).

To determine the mode of the lock, the thread in *getMode* reads the current system time[2] and determines at what part of the current cycle it is. If it is in the profiling phase, it ensures that profiling data has been initialized by calling *start-*

[2]To reduce overhead, the system time may be cached in a thread-local variable and read less frequently.

```
int getMode(Lock *l) {
  long start = startTime; // shared variable, set
       once when it is read for the first time
  long now = getCurrentTime();
  long timeIntoCycle = (now - start) % CYCLE_LEN;
  if (timeIntoCycle < PROFILING_LEN) {
    startProfiling(l, now - timeIntoCycle);
    return timeIntoCycle / (PROFILING_LEN/NUM_MODES);
  } else {            // profiling is finished
    finalizeProfiling(l);

    if (l->fastestModeSlice == 1 ||
      ((timeIntoCycle - PROFILING_LEN) % QUANTUM_LEN
           < l->fastestModeSlice * QUANTUM_LEN))
      return l->fastestMode;

    return l->alternateMode;
  }
}

void startProfiling(Lock *l, long profStart) {
  long t = l->lastProfStart;
  while (t < <profStart,1>) {
    if (t < <profStart,0> &&
        CAS(&l->lastProfStart, t, <profStart,0>)) {
      // reset learning counters
      set all entries of l->acquisitions to 0
      // signal that profiling data is initialized
      CAS(&l->lastProfStart, <profStart,0>,
          <profStart,1>);
      return;
    }
    t = l->lastProfStart;
  }
}
```

Figure 10: Auxiliary functions for getting the current mode and starting a new profiling phase.

Profiling and returns the mode corresponding to the current time. If it is in the post-profiling phase, it ensures that the profiling data has been finalized by calling *finalizeProfiling*.

The profiling functions are shown in Figures 10 and 11. In *startProfiling*, the thread compares the current time with the value of *lastProfStart* stored in the lock. If the initialization is required, it atomically sets (using CAS) the *lastProfStart* to the current time and resets the entries of the *acquisitions* collection to 0. In *finalizeProfiling*, if the data has not been finalized (determined by checking the last two bits of the *lastProfStart* field stored in the lock), the thread toggles these bits (using CAS), signaling other threads that it is in the process of summarizing profiling data, which it does by calling *computeBestLockModes*. There, if it finds out that most acquisitions took place in the mode allowing both sockets to run, it sets *fastestMode* to 2 and *fastestModeSlice* to 1; otherwise, it sets *fastestMode* to the mode with the most acquisitions and sets *fastestModeSlice* to the ratio of the number of acquisition in the *fastestMode* to the total number of acquisition in that mode and in the mode $1 - fastestMode$ (i.e., the mode corresponding to the other socket).

To avoid cases where insufficient profiling data (e.g., during warmup time) may lead to a suboptimal decision of using only one of the sockets, we use a threshold (set to 256); if the total number of lock acquisitions in all modes falls below that threshold, *fastestMode* is set to 2 (and *fastestModeSlice* is set to 1). (For simplicity, this optimization is omitted from the pseudo-code.)

After calling *finalizeProfiling*, the thread decides (in *getMode*) based on the current time and the values of *fastestMode* and *fastestModeSlice* on the current mode for the lock.

```
void finalizeProfiling(Lock *l) {
  long t = l->lastProfStart;
  // check if the profiling data has been finalized
       already
  if (S(t) == 3) return;

  if (S(t) == 1 &&
      CAS(&l->lastProfStart, <t,1>, <t,2>)) {
    computeBestLockModes(l);
    CAS(&l->lastProfStart, <t,2>, <t,3>;
  } else
    while (S(t) == 2 &&
      <t,0> == <l->lastProfStart,0>) {
      t = l->lastProfStart;
    }
}

void computeBestLockModes(Lock *l) {
  // compute fastest mode for the given 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] += l->acquisitions[j][m];
    }
  }
  l->fastestMode = index of largest element of acqs
  l->alternateMode = index of second largest
                          element of acqs

  if (l->fastestMode == NUM_MODES-1) {
    // both sockets run in fastest mode, so no need
         to alternate modes
    l->fastestModeSlice = 1;
  } else {
  // Only one socket runs in fastest mode. Divide
       quantum between fastest and alternate modes.
  l->fastestModeSlice = acqs[l->fastestMode] /
    (acqs[l->fastestMode] +
      acqs[1 - l->fastestMode]);
  }
}
```

Figure 11: Auxiliary functions for finalizing a profiling phase and computing modes for a post-profiling phase.

If the mode returned by *getMode* is the one corresponding to the socket on which this thread is running (which can be found through a library call and is stored in a thread-local variable) or corresponding to the mode that allows both sockets to run, the thread proceeds with the underlying TLE operation. Otherwise, if the current mode corresponds to the socket different from the one on which the thread is running, the thread spins for a while (or yields) and then repeats from the beginning of the *LockAcquire* function. To avoid pathological cases (e.g., when a thread continuously misses the times when the corresponding lock is in the mode that would allow it to run), the number of repetitions is limited by a large constant. To accommodate possible thread migrations, each thread infrequently (e.g., every 1K attempts to execute *LockAcquire*) rechecks the socket on which it is running. Critically, even if the thread decides to run in the "wrong" mode, correctness is preserved, and only performance may be affected (for limited time, until the thread finds out the real mode it should run with). Also, note that NATLE does not assume any specific underlying TLE implementation. While in our evaluation in Section 5 we use a common TLE implementation (denoted as *TLE-20* in Section 3.1), other variants are possible, e.g., a TLE implementation that performs contention management using an auxiliary lock [2].

5. EXPERIMENTAL RESULTS

5.1 Data structure microbenchmarks

In this section, we present a selection of results from a large suite of microbenchmarks. We implemented the set data type (i.e., supporting insert, delete, and search operations) in three different ways, as an AVL tree, as an unbalanced binary search tree (BST), and as a skip-list. Each implementation has a single lock that protects every operation, and we used two variants of TLE to implement this lock. The first variant (which we simply call TLE) is the common implementation described in Section 3.1, in which a thread makes 20 attempts using HTM before falling back to a (test-and-test-and-set) lock. The second variant is the NATLE algorithm described in Section 4 implemented on top of this common TLE implementation.

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 invoke operations with keys drawn uniformly at random from a fixed key range with a specified proportion of update vs. search operations. Before a trial begins, the data structure is prefilled so that it contains half of its key range, and among the update operations, inserts and deletes are equally likely, so the data structure remains approximately half full. As mentioned in Section 3, we pin the first 36 threads to one socket and the next 36 to the other socket. In all experiments, we use an HTM-friendly memory allocator [10]. 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.

Figure 12 shows the results of several experiments on AVL trees. First consider the graphs on the left, in which threads do not do any "external work" between operations. For the read-only workload (top left graph), both TLE variants scale (the reductions in slope from 18 to 36 threads and from 54 to 72 threads occur because of hyperthreading). However, NATLE achieves about 27% less throughput than TLE due to periodic profiling overhead and time sampling, which are redundant in this case because the best performance is achieved by simply using both sockets. Tuning the frequency and the length of profiling, as well as the frequency of time sampling is possible, but is left for future work.

In more realistic settings, threads typically perform some work between consecutive operations on a shared data structure. We emulate this external work by having each thread call a short function a random number of times; this random number is chosen from some preset range (e.g., $[0, 256)$). The graphs on the right side of Figure 12 show the results of experiments with such external work on the AVL tree. In the read-only workload and a single-thread experiment, the external work reduces the throughput of TLE (and NATLE) by about 2/3. At the same time, the gap between NATLE and TLE in this workload shrinks to less than 13%.

When we introduce update operations, both TLE and NATLE do not scale as well, and this worsens as the proportion of updates increases. However, once threads start to execute on the second socket (i.e., when there are more than 36 threads), TLE's throughput drops dramatically, whereas NATLE manages to stay closer to the peak throughput achieved at 36 threads. This is true with or without external work. In summary, NATLE can exploit both sockets for workloads

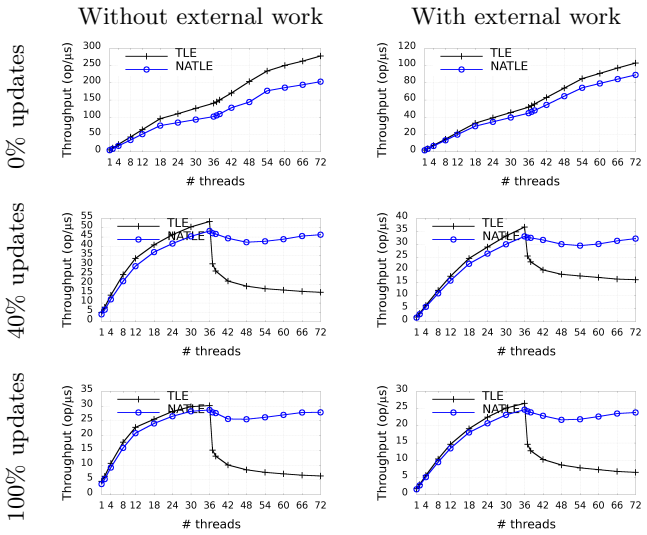


Figure 12: Experimental results for AVL trees. The key range is $[0, 2048)$.

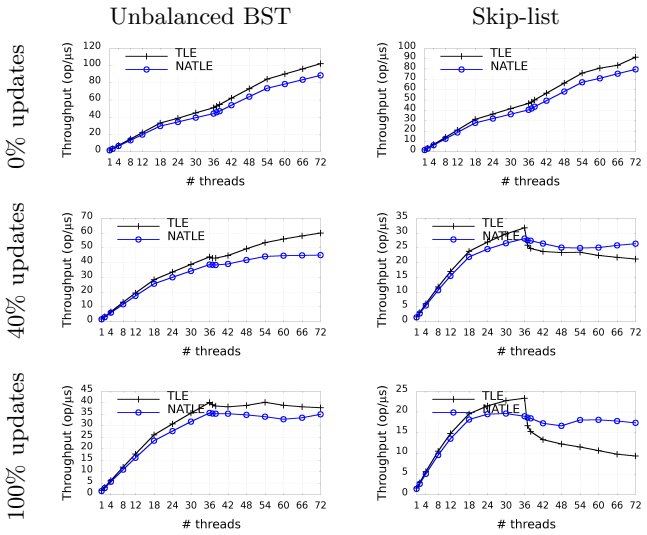
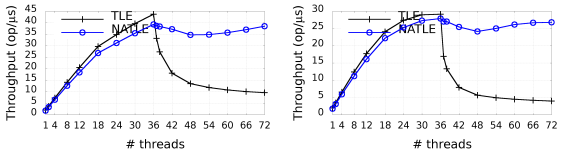


Figure 13: Experimental results for unbalanced BSTs (left) and skip-lists (right). The key range is $[0, 2048)$.

that scale on both sockets with TLE (e.g., the read-only workload), but unlike TLE, it exhibits little or no performance degradation when using cores on both sockets for workloads that do not scale.

The results for experiments with unbalanced BSTs and skip lists are provided in Figure 13. There, threads perform the same amount of external work as in the right part of Figure 12. Interestingly, TLE performance in unbalanced BSTs is not prone to the NUMA effects as operations do not rotate the tree and thus always modify only nodes at or near tree leaves. Indeed, the NATLE profiling statistics show that the number of critical section executions is slightly larger when both sockets are used, and thus it chooses to use both sockets. However, as Figure 14 demonstrates, reducing the key range increases the chance for a data conflict between tree operations, which in turn makes TLE susceptible to the NUMA effect. In this case, NATLE is able to avoid the performance degradation. Its profiling statistics show



(a) 40% updates (b) 100% updates

Figure 14: Experimental results for unbalanced BSTs with key range $[0, 128)$.

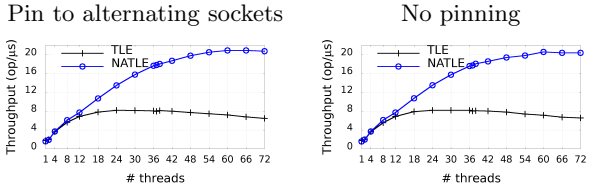


Figure 15: Experimental results for AVL trees with different thread pinning policies. The workload is 100% updates with key range $[0, 2048)$.

that using one socket at a time is by far more efficient that using both sockets. Along with that, the performance of skip lists (cf. Figure 13) is more like AVL trees, although the performance degradation of TLE is slightly less dramatic.

We also experimented with several different policies for pinning threads. In Figure 15, we show graphs using two alternative pinning policies for workloads with 100% updates, key range $[0, 2048)$, and the same amount of external work as in right part of Figure 12. For the graph on the left, threads are pinned to alternating sockets (i.e., even-numbered threads are pinned to cores on socket 0 and odd-numbered threads are pinned to cores on socket 1). For the graph on the right, threads are not pinned; rather, the operating system decides placement. The similarity between these two graphs suggests that the Linux scheduler attempts to evenly distribute the load across sockets. Since the workload does not scale on two sockets, the benefit of NATLE is more significant, and is evident earlier (starting at eight threads).

Although the benchmarks we have considered thus far use only one lock, NATLE can profile and throttle threads for applications that use multiple locks, and it does so independently for each lock. That is, the mode(s) selected for each lock is determined separately for each lock based on the statistics gathered for that lock during the profiling phase. For example, in Figure 16, we show the results of an experiment in which there are two AVL trees, both with key range $[0, 2048)$. Half of the threads invoke only update operations on one tree; the other half invoke only search operations on the other tree. (We only run this experiment with even numbers of threads.) As before, threads are pinned to spread across the cores of a single socket as much as possible (i.e., until there are more than 18 threads), and then use hyperthreading before spilling over onto the other socket. When there are more than 36 threads, we ensure that there are equal numbers of threads accessing each tree. In other words, the first 36 threads (divided equally between two groups) run on the first socket, and the remaining threads (again, divided equally between two groups) run on the second socket.

Note that these two groups of threads are accessing different trees, so threads in one group never conflict with threads in the other group. To simplify the evaluation, we want both groups of threads to have approximately the same throughput (when running without contention). Since search operations are generally much faster than update operations, we achieve this by adding external work between invocations of the search operations.

We know from Figure 12 that the workload with 100% updates does not scale on two sockets, whereas the workload with 0% updates does. Thus, the threads doing update operations should be throttled so that at any time, only threads from one socket are accessing the update-only tree. In contrast, the threads doing search operations should not be throttled: threads executing on different sockets can, and should, access the read-only tree concurrently.

The graphs in Figure 16 show the combined throughput of both groups of threads, and also the contribution of each group to that combined throughput. NATLE significantly outperforms TLE when there are more than 36 threads because it throttles the update-only threads but not the search-only threads. The graphs that break out performance by data structure (two rightmost charts in Figure 16) show that the drop at 38 threads for TLE occurs mostly because of a drop in the performance of the tree with updates operations. At the same time, NATLE keeps scaling by avoiding this drop.

5.2 STAMP

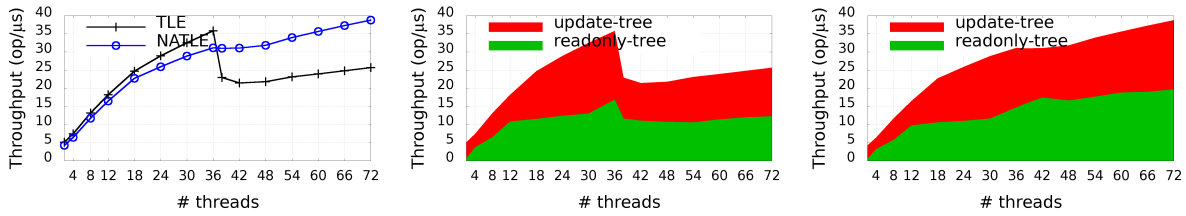
In this section, we present results with STAMP [27], a common benchmarking suite for transactional memory programs. We evaluate the version of the suite provided by Ruan et al. [31], which includes a number of code modifications that allow adaptation to the standard transactional memory interface. Note that NATLE does not require the latter, however, it can be used with transactional programs as long as the transactional runtime configured to simply use a pthread lock for each transaction. This is exactly what we have done by overwriting the transactional runtime implementation library (libitm) provided with GCC.

The results for various STAMP benchmarks and standard workloads are shown in Figure 17. (We omit the results for the bayes benchmark since it highly depends on the order of various parallel computations and thus exhibits high variance; this fact was noted in [34].) These benchmarks report the total runtime, hence the lower result is better.

STAMP benchmarks appear to be very sensitive to cross-socket contention: In 7 out of 9 charts the runtime of TLE skyrockets when the number of threads exceeds the capacity of one socket. In all these cases, however, NATLE is able to maintain roughly the same performance across all thread counts beyond 36.

5.3 ccTSA

The ccTSA software is an open source implementation of a de novo gene sequence assembler [4]. It gets an input file with DNA segments and tries to assemble the whole genome by looking for overlap between the segments. It uses a lock-protected hash-map to store subsequences of DNA segments during their processing. ccTSA achieves good scalability on real genome data by splitting the main hash-map data structure into thousands of smaller hash-maps, each protected by its own lock. For our evaluation, however, we are using the



(a) Total throughput (b) Break-down for TLE (c) Break-down for NATLE

Figure 16: Experimental results for a workload involving two AVL trees. The left graph compares the performance of the TLE algorithms. The other two area graphs each show a single 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.

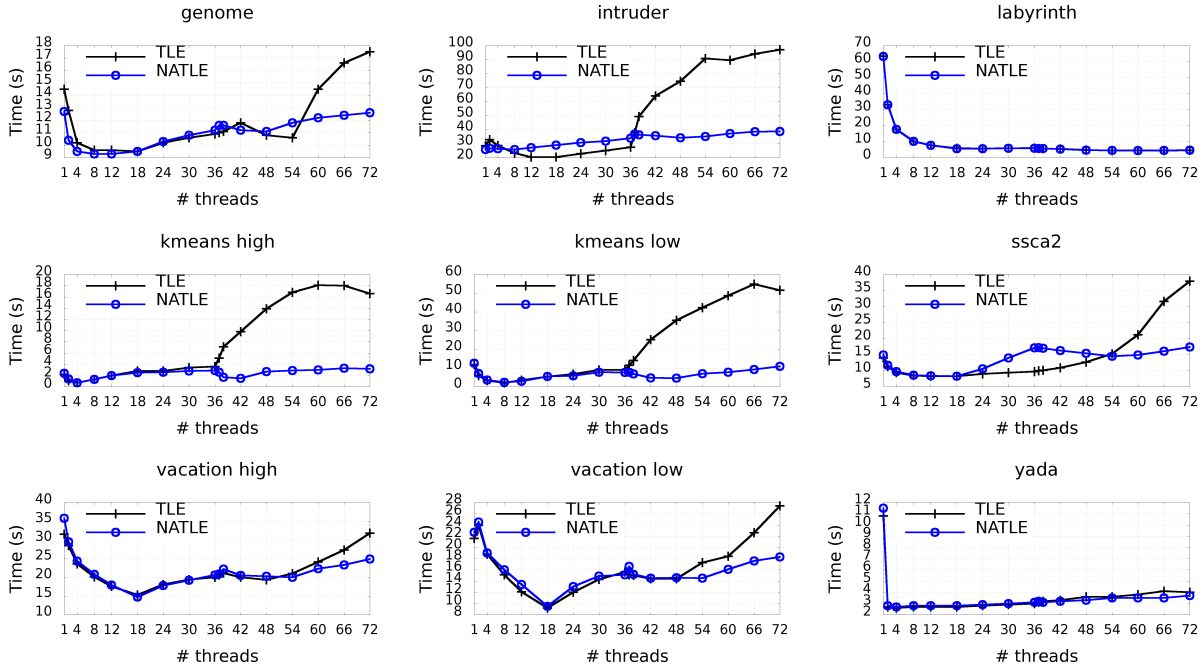
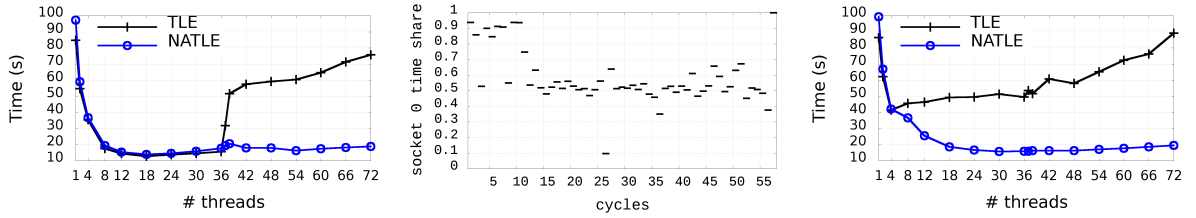


Figure 17: Experimental results with STAMP.



(a) Total runtime (with pinning) (b) Time allocation for socket 0 when running with 72 threads (with pinning) (c) Total runtime (without pinning)

Figure 18: Experimental results with ccTSA.

transactified version of ccTSA [11], which simplifies the design of the original implementation and uses a single (lock-protected) hash-map. Comparing to the original lock-based implementation, this version achieves better single-thread performance when used with locks, and performs more than twice better at any thread count when the lock is elided using TLELib (see [11] for details).

For our evaluation we use DNA segments from the E.coli organism that were provided with the original software, and configure ccTSA (e.g., the size of subsequences) with the same configuration as reported in [11]. Figure 18(a) compares the performance of ccTSA with TLE and NATLE using our usual pinning policy. Both variants scale well up to 18 threads and then keep stable performance up to 36 threads. When the number of threads exceeds the capacity of one socket, however, the runtime of TLE jumps, almost reaching the level of the single thread. In contrast, NATLE decides to throttle periodically one of the sockets, maintaining the same level of performance up to 72 threads.

Data shown in Figure 18(b) sheds some light on the decisions taken by NATLE in a run with 72 threads. The x-axis corresponds to runtime cycles (as a reminder, NATLE divides the running time into cycles and profiles performance at the beginning of each cycle), while the y-axis corresponds to the ratio of running time allocated in each quantum of the corresponding cycle for socket 0. Based on these results, one can see that for the majority of cycles, NATLE decides to allocate roughly half of the quantum time to each of the two sockets. The decision is taken based on the profiling, which shows that for these cycles, socket throttling is more effective than using both sockets at the same time.

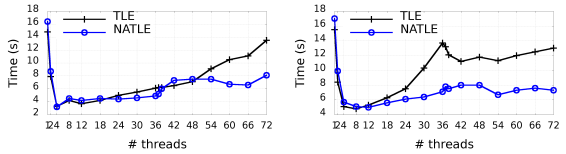
We also show the results of running this benchmark without pinning threads (see Figure 18(c)). In this case, the benefit of using NATLE becomes evident much earlier because the cctsa benchmark does not scale across sockets.

5.4 paraheap-k

paraheap-k is a small (less than 1000 lines of code) parallel heap-based application for calculating k-means clusters [21]. It has been developed in the context of studying astronomical data to process efficiently galactic spectral data with the task of finding galactic components. paraheap-k gets as in input a file with galactic coordinates and uses a configurable number of threads to calculate iteratively k centroids. The calculation process stops once the ratio of data points that maintains an association with the same centroid across subsequent iterations goes above a configurable threshold (99.9%, by default). At the end, the position of calculated centroids as well as the data point association is written to output files.

Overall, the paraheap-k application has 7 critical sections, 6 of which are very short (just update a shared counter) and another one inserts a data point into the heap. Furthermore, the application uses multiple locks, making it an interesting use case for the evaluation of NATLE. The results of this evaluation with the usual pinning policy are shown in Figure 19(a). We measured the time of the actual data processing, i.e., we did not include the time to read the provided input file and output the results.

Despite the fact that NATLE statistics show that for most parts of executions with more than 36 threads, using one socket at a time yielded substantially higher throughput then when using both sockets, the benefit of NATLE over



(a) With pinning (b) Without pinning

Figure 19: Experimental results with paraheap-k.

TLE is relatively moderate. We believe this is a result of the way this benchmark creates and uses multiple worker threads. Specifically, while all other benchmarks considered so far create their working threads once and at the beginning of their runs, paraheap-k does so in every processing iteration (there are more than a dozen iterations per run), and in fact, does so twice per iteration (to associate data points with centroids, and then to recalculate the position of those centroids once the data association phase is done). Each time a new thread is created, it is pinned according to the pinning policy, and the overhead of this pinning becomes substantial when it is done frequently, eliminating most of the benefits achieved by NATLE in reducing conflicts between threads running on different sockets. The results of the evaluation without pinning confirm this hypothesis (see Figure 19(b)). There, the benefit of NATLE is much more substantial, and it becomes evident at 18 threads.

6. 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 more 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.

7. 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 *Proc. ACM PODC*, pages 212–221, 2014.

[3] Y. Afek, A. Matveev, O. R. Moll, and N. Shavit. Amalgamated lock-elision. In *Proc. DISC*, pages 309–324, 2015.

[4] J. H. Ahn. ccTSA: A Coverage-Centric Threaded Sequence Assembler. *PLoS ONE*, 7(6), June 2012.

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

[6] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore systems. In *Proc. USENIX ATC*, 2011.

[7] 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.

[8] 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.

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

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

[11] D. Dice, A. Kogan, and Y. Lev. Refined transactional lock elision. In *Proc. ACM PPoPP*, pages 19:1–12, 2016.

[12] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir. Adaptive integration of hardware and software lock elision techniques. In *Proc. ACM SPAA*, pages 188–197, 2014.

[13] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. ACM ASPLOS*, pages 157–168, 2009.

[14] 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.

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

[16] 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.

[17] N. Diegues, P. Romano, and S. Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *Proc. ACM SPAA*, pages 224–233, 2015.

[18] 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.

[19] A. Hassan, R. Palmieri, and B. Ravindran. Remote transaction commit: Centralizing software transactional memory commits. *IEEE Transactions on Computers*, pages 26–33, 2015.

[20] M. Herlihy and E. Moss. Architectural support for lock-free data structures. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993.

[21] M. Jenne, O. Boberg, H. Kurban, and M. Dalkilic. Studying the milky way galaxy using paraheap-k. *Computer*, 47(9):26–33, 2014.

[22] 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.

[23] B. Lepers, V. Quema, and A. Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *Proc. USENIX ATC*, 2015.

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

[25] 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 *Proc. USENIX ATC*, 2012.

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

[27] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: stanford transactional applications for multi-processing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 35–46, 2008.

[28] 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 *Proc. ACM/IEEE ISCA*, 2015.

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

[30] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using dope: The degree of parallelism executive. In *Proc. ACM PLDI*, pages 26–37, 2011.

[31] W. Ruan, Y. Liu, and M. Spear. STAMP need not be considered harmful. In *Proceedings of ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.

[32] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *Proc. IEEE HPCA*, pages 188–197, 2013.

[33] 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.

[34] 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.

[35] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proc. ACM SPAA*, pages 169–178, 2008.