Hardware Transactional Memory Optimization Guidelines, Applied to Ordered Maps

Bonnichsen, Lars Frydendal; Probst, Christian W.; Karlsson, Sven

Published in:
Proceedings of the 13th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2015)

Link to article, DOI:
10.1109/Trustcom.2015.621

Publication date:
2015

Document Version
Peer reviewed version

Hardware Transactional Memory Optimization Guidelines, Applied to Ordered Maps

Lars F. Bonnichsen
Technical University of Denmark
lfbo@dtu.dk

Christian W. Probst
Technical University of Denmark
cwpr@dtu.dk

Sven Karlsson
Technical University of Denmark
svea@dtu.dk

***Abstract*—Synchronization of concurrent data structures is difficult to get right. Fine-grained synchronization locks small data chunks, but requires too high an overhead per chunk; traditional coarse-grained synchronization locks big data chunks, and thereby makes them unavailable to other threads. Neither synchronization method scales well. Recently, hardware transactional memory was introduced, which allows threads to use transactions instead of locks. So far, applying hardware transactional memory has shown mixed results. We believe this is because transactions are different from locks, and using them efficiently requires reasoning about those differences. In this paper we present 5 guidelines for applying hardware transactional memory efficiently, and apply the guidelines to *BT-trees*, a concurrent ordered map. Evaluating *BT-trees* on standard benchmarks shows that they are up to 2 times faster than traditional maps using hardware transactional memory, and up to 3 times faster than state of the art concurrent ordered maps.**

I. Introduction

Writing efficient parallel code is difficult. It is even more difficult if the code modifies shared data unpredictably. Consider for instance the map data structures, which store relationships between keys and values, which can be queried, added, or removed. Ordered maps store totally ordered keys, and extend normal (unordered) maps with the ability to iterate over keys according to their order. Concurrent maps, which can be modified in parallel, use synchronization to keep their contents consistent. State of the art concurrent ordered maps are almost 10 times slower than concurrent unordered maps, because the ordering on the key-value pairs makes it difficult to synchronize efficiently [**?**]. Traditional concurrent ordered maps attempt to synchronize efficiently with customized synchronization. Focusing on synchronization first, rather than the design of the ordered maps, means that the ordered maps are more complex and less efficient. Recent hardware support for transactional memory promises to enable efficient *and* simple synchronization, but applying it efficiently still has a number of pitfalls.

In this paper we present 5 guidelines for applying hardware transactional memory efficiently. The guidelines inform the design decisions of the underlying problem, rather than requiring intrusive custom tailored synchronization. We use the guidelines in the design of a new concurrent ordered map, BT-trees. We evaluate BT-trees with up to 8 threads on a set of standard benchmarks, and find that they are up to 3 times as fast and almost twice as space efficient as state of the art concurrent ordered maps. BT-trees use a simple synchronization mechanism, but are still faster than the state of the art, because they are designed with main focus on providing efficient ordered maps, rather than specialized synchronization. To validate the guidelines, we also apply hardware transactional memory to off the shelf sequential ordered and unordered maps, with traditional designs. These traditional maps have significantly worse parallel performance, typically well below that of state of the art concurrent ordered maps. Experiments show that traditional maps are dramatically affected by synchronization contention; when scaling from 1 to 8 threads traditional maps using hardware transactional memory often execute 3 times as many instructions and cause far more L1 cache misses.

Our main contributions are a list of 5 guidelines to apply hardware transactional memory efficiently, the application of these guidelines to the design and implementation of BT-trees, a concurrent ordered map data structure, and the evaluation of BT-trees, showing the validity of our guidelines.

The rest of this paper is structured as follows: In Section 2 we give a more thorough introduction to transactional memory. Section 3 reviews the guidelines to avoiding the limitations of hardware transactional memory, and Section 4 presents the design of BT-trees based on these guidelines. Section 5 evaluates BT-trees and ordered maps which do not follow our guidelines. Section 6 discusses related work and Section 7 concludes the paper.

II. Transactional memory

Transactional memory enables executing a section of code in transactions, which can fail or succeed. Successful transactions appear as if they executed atomically, with no interference from concurrently running threads. Failed transactions have no visible effects to other threads. Transactions can be used to implement the critical section of code by trying to execute it with a transaction until the transaction succeeds.

Transactional memory was introduced in 1993 [1] and is traditionally implemented in software [2]. Software transactional memory has application-specific overhead that can vary widely [3]. Recently, Intel, IBM, and Sun, have implemented hardware support for transactional memory (HTM) [4]–[6]. HTM implementations have a lower overhead than software transactional memory at the price of more limited transactions. Hardware transactions may abort for many other reasons than conflicts, for example, interrupts, system calls, page faults, large transactions, debugger breakpoints, cache evictions, TLB misses, or problematic instructions, such as division. The page fault and system call limitations mean that transactions may fail when executing a line of code the first time, or when

allocating memory. Retrying transactions that fail because of system calls or page faults will also fail, because the previous transaction rolled back its changes.

Using HTM directly is impractical because of its limitations, but it is easy to use HTM for lock-elision [4]. Lock-elision is an optimization on normal locks that initially tries to execute the corresponding critical section in a transaction:

1) Try to execute the operation in a transaction that fails if the lock is held;
2) Retry the transaction if it conflicts or fails; and
3) Use the lock if the transaction fails again.

This particular form of lock-elision is called SLR lock-elision [7].

III. Designing for HTM

In this section we provide general guidelines for how to design parallel data structures and algorithms such that they benefit from lock-elision. We focus on guidelines for using lock-elision, because using HTM directly is problematic. The main goal of these guidelines is to illuminate the benefits and pitfalls of HTM, to support reasoning about them up front.

HTM transactions can fail for 3 reasons:

1) Limitations in the hardware support,
2) Conflicts in the transactions read set, ie another thread wrote to the transactions read set, or
3) Conflicts in the transactions write set, ie another thread read or wrote to the transactions write set.

The main hardware limitations are that transactions fail on false sharing, too large transactions, system calls or page faults. These limitations can largely be avoided by (1) **optimizing spatial locality**, ie packing data tightly, and (2) **avoiding system calls and page faults in the critical section**. With traditional synchronization it is also a good idea to avoid system calls, page faults, and optimize spatial locality, but failure to do so is less dramatic: with HTM failure system calls, page faults, and poor spatial locality cause transactions to fail, while with traditional synchronization it just causes cache line evictions and slightly longer critical sections. Even if there without hardware limitations on HTM, page faults, system calls, and large transactions would still reduce transaction's probability of succeeding, because they increase the size of the transactions the read and write sets.

To avoid conflicts in the read set you should, (3) **use data structures and access memory such that the memory which is most frequently written is least frequently read**. Guideline (3) is similar to saying avoid true sharing, which is also a good idea with traditional synchronization, as it will minimize the amount of synchronization. To avoid conflicts in the write set you should, (4) **minimize the time from the first visible write to the transaction's commit**: Try to write as late as possible in the transaction, and prefer copy-on-write to long in-place writes. Guideline (4) is specific to transactional memory which use eager conflict resolution, ie detect conflicts in transactions before they commit. With eager conflict resolution, a transaction can fail from a conflicting write after the write has executed. Delaying the write will reduce the time that the transaction is exposed to such conflicts. Current HTM implementations from Intel and IBM presumably use eager conflict resolution because it can built on top of existing cache coherency protocols [8], [9].

The guidelines can also be used to estimate if using HTM will be beneficial: If you can follow the guidelines, *ie* have a very short duration from the first write to commit, avoid writing to frequently read data, and avoid limitations in HTM, then you are likely to benefit, because most transaction should succeed. If most of the transactions succeed, then the size of the critical section is no longer important. As a consequence you should (5) **worry less about the size of critical sections**: lock-elision on a single *coarse-grained* lock can scale well if most transactions succeed. Coarse grained locking simplifies writing parallel code to the point where it is hardly any more difficult than writing sequential code.

Using several locks per data set, or another kind of *fine-grained* synchronization, would not be efficient use of HTM: Starting a transaction with HTM on Intel Haswell processors is 3 times more expensive than acquiring a lock, and coarse-grained synchronization with HTM already permits parallelism within critical sections. Fine-grained synchronization will rarely benefit from HTM, and it multiplies the overhead of starting transactions; it also causes new problems, for instance, concurrent operations must not deallocate the memory they share. Typical solutions to this problem imply a hefty performance and space penalty, and may limit which memory locations can be accessed in parallel algorithms. If fine-grained synchronization is necessary, one should use regular locks or atomic operations, not HTM.

IV. BT-trees

In this section we present the design of BT-trees, a design which is largely driven by the guidelines in Section III. BT-trees are ordered maps supporting the operations SEARCH, INSERT, and REMOVE, for querying, updating, or removing key-value associations. BT-trees are search trees, where the root node represents the entire range of keys, and each of its children represents a smaller subset of the keys. The children are ordered from lowest (child 0) to highest, such that the lowest child can hold the lowest keys, and the highest node can hold the highest keys. While this design decision is not

TABLE I. Guidelines for using HTM efficiently

Guideline	Motivation
(1) Optimize spatial locality and avoid false sharing	Reduces the size of the read and write sets
(2) Avoid system calls and page faults	Hardware transactions fail on page faults and system calls
(3) Minimize true sharing	Avoids conflicts in the read set
(4) Reduce time from first visible write to commit	Increases success rate under current HTM
(5) Use coarse grained synchronization	If HTM works well, the critical sections size is not important

```
class E<K, V> {K key; V value; }; // Key-value pairs

class alignas(64) L<K, V> { // Leaf nodes
  E<K, V> e[L_C]; // Unordered key-value pairs
};

class alignas(64) I<K> { // Internal nodes
  I* child[I_C]; // Pointers to children
  int size; // Number of children
  K key[I_C - 1]; // Internal node keys
};

class BT { // BT trees
  int height; // The tree's height
  I* root; // Pointer to the tree's root
  Lock lock; // The tree's lock
};
```

Listing 1. Type definitions for BT-trees.

related to the guidelines from the previous section, the rest of the design decisions are.

BT-trees have two features to improve spatial locality (recall Guideline 1): (1) BT-trees' nodes are aligned to cache line boundaries, making cache line references more local, and eliminating false sharing, and (2) BT-trees are multiway search trees: each internal node is aligned to cache line boundaries and can have multiple children and keys, such that we read keys adjacent to each other, and take full advantage of the cache lines being read.

BT-trees are external trees: BT-trees store all the key-value pairs they represent in their leaf nodes. Finding the leaf node which may hold a key mostly involves reading from internal nodes, as well as occasional writes when balancing the tree. Because all associations of a BT-tree are in its leaf nodes, the BT-tree operations all follow 3 steps:

1) Find the leaf node l for the given key.
2) Inspect l's keys.
3) Balance l or operate on l based on its keys.

As a result, the INSERT and REMOVE operations always write to leaf nodes, typically without writing to any internal nodes, reducing true sharing (Guideline 3). BT-tree operations only write to internal nodes when balancing the trees, making it important to reduce the frequency of balancing.

BT-trees' operations rarely balance the tree, because of their balancing scheme: unbalanced nodes are balanced while searching through the tree.

- Full nodes are split into two nodes, and nodes with 2 elements are merged with their siblings;
- If balancing would merge the two only children of the root, we instead replace the root reducing the trees height; and
- If balancing would split the root, we introduce a new root node increasing the trees height.

From an algorithmic point of view, this scheme ensures that all leaf nodes have the same depth, and that all non-root nodes have at least 2 elements, giving a worst case height of $log_2(n/2)+1 = log_2 n$. Given that the nodes have capacity C, the expected height is $log_{\frac{2+C}{2}} \frac{n}{2} + 1$. The scheme also provides a control knob for the frequency of balancing: increasing

```
REMOVE(T, k)
 1  // Step 1: Find the leaf node l
 2  repeat
 3      PREALLOC6NODES()
 4      LOCK()
 5      l = FIND-LEAF(T.root, T.height, k)
 6  until l ≠ RESTART
 7  // Step 2: Inspect l's keys (l.key[1 .. L_C])
 8  u = 0   // Used key-value pairs
 9  m = 0   // Matching key-value pair index
10  for i = 1 to L_C
11      if l.element[i].key ≠ EMPTY
12          u = u + 1
13      if l.element[i].key == k
14          m = i
15  // Step 3: Balance l or remove k from l
16  if m == 0
17      return NO_MATCH
18  if T.root == l or u > 2
19      v = l.element[m].value
20      l.element[m].key = EMPTY
21      UNLOCK()
22      return v
23  BALANCE(l)
24  UNLOCK()
25  return REMOVE(T, k)

FIND-LEAF(x, h, k)
 1  if h == 0
 2      return x // x is the leaf
 3  if ¬IS-BALANCED(x)
 4      BALANCE(x)
 5      UNLOCK()
 6      return RESTART
 7  for i = 1 to x.size
 8      if x.key[i] ≥ k
 9          return FIND-LEAF(x.child[i], h − 1, k)
10  return FIND-LEAF(x.child[x.size], h − 1, k)
```

Listing 2. BT-tree remove operation pseudo code.

C reduces the frequency of node balancing, reducing the frequency of write to internal nodes, and ultimately reducing the risk of true sharing and the number of writes.

BT-trees replace the parent of unbalanced nodes when balancing them, rather than replacing the unbalanced nodes inside of their parent node. Replacing the parent node means that balancing has to copy the old parent node, which is more expensive, but it reduces the number of writes to internal nodes. Perhaps more importantly, it ensures that the write to the internal node is the last thing in the transaction, minimizing the time from the first visible write, to the transactions commit, as per Guideline 4.

BT-trees further reduce the number of writes and the time from the first write to the transactions commit, by representing leaf nodes as unordered arrays of key-value pairs. REMOVE and INSERT operations only have to write to one memory location in the leaf nodes. By comparison, if the leaf nodes stored keys and values separately, the operations would have to write twice, and if the leaf nodes stored key-value pairs in an ordered fashion, then the operations would have to write

several times to preserve the ordering.

Memory allocation typically involves both page faults and system calls. BT-trees operations avoid performing system calls and page faults in the critical section (Guideline 2), by preallocating 6 nodes before entering a critical section. Preallocating 6 nodes ensures that no additional memory allocation is needed in the critical section. The allocated nodes are stored on a stack created from the allocated memory, thereby touching the page of the allocated nodes, causing the page fault before the transaction starts.

Listing 1 illustrates how BT-trees, key-value pairs, and nodes are represented in pseudocode resembling C++. The classes `I` and `L` represent internal and leaf nodes respectively, while `E` represents key-value pairs. Internal nodes have other internal nodes or leaves as children. Leaf and internal nodes are aligned to cache line boundaries by using the C++11 `alignas` keyword, and allocating with `new`. Each leaf node can store up to L_C key-value pairs, and internal nodes have up to I_C children, where $L_C, I_C \geq 6$. The lower bound node capacities of 6 ensure that we can split a full node into two nodes with at least 3 children or key-value pairs.

We normally use BT-trees with up to 32 children for each internal node, and 32 key-value pairs for each leaf node. When using 64 bit pointers and 32 bit keys and values, this corresponds to 384 bytes for internal nodes and 256 bytes for leaf nodes, or exactly 6 and 4 cache lines respectively.

Listing 2 summarizes how the REMOVE operation works. The REMOVE operation first traverses the tree from its root to the leaf node which may hold k, while balancing any unbalanced node on the path (Step 1). Upon arriving at a leaf node, the REMOVE operation balances the leaf if it is unbalanced. Otherwise, the operation iterates over the keys in the leaf node, looking for a match (Step 2). If it finds a match, it returns the keys value and removes from the tree. Otherwise there was no match, and REMOVE returns NO_MATCH (Step 3). INSERT and SEARCH work similarly to REMOVE, but for brevity's sake we defer their description to a technical report [10]. Listing 2 is a summary, and it glosses over some technical details, such as tracking the parent of the visited node, and how to balance. The techical covers these details and some implementation optimizations which improve performance significantly.

V. EVALUATION

A. Experiment setup

We evaluate BT-trees, Chromatic trees, and Java `ConcurrentSkipListMap` on the machine described in Table II. Chromatic trees are a state of the art lock-free ordered map, implemented as a relaxed red-black tree, which we acquired from Brown's homepage [11]. Java `ConcurrentSkipListMap` is a well established lock-based ordered map, implemented as a skip list. We also evaluate GCC's STL implementation of `map` and `unordered_map` (v4.9.1) where we synchronize using SLR lock-elision. The C++ map implementations all use the memory allocator provided with Intel TBB v4.3_20141023.

TABLE II. EXPERIMENTAL MACHINE

Processor	Intel Xeon E3-1276 v3@3.6GHz
Processor specs	4 cores, 8 threads
Processor specs(2)	32KB L1D cache, 8 MB L3 cache
C++ Compiler	GCC 4.9.1
Java Compiler/Runtime	Oracle Server JRE 1.8.0_20
Operating system	Ubuntu Server 14.04.1 LTS
Kernel	3.17.0-031700-generic
libc	eglibc 2.19

We use the experiment from Brown *et al.* [12], and port the experiment to C++ to evaluate the C++ maps. The experiment has been reproduced in several recent papers [13], [14]. The Java were tested with the test infrastructure hosted on Brown's website.

In the experiment up to 8 threads operate in parallel on one map for 5 seconds, after pre-filling the map with n key-value pairs. After the 5 seconds, we record how many operations the threads completed. The C++ implementations also record several performance metrics, such as cache misses. The operation's keys are 32 bit integers, uniformly sampled from 1 to k, where k is either 100, 10,000, or 1,000,000. We evaluate 3 workloads with different proportions of insert, remove, and search operations:

1) Update, with 50% insertion, 50% removal ($n = k/2$);
2) Mixed, with 70% searches, 20% insertion, and 10% removal ($n = 2k/3$); and
3) Constant, with 100% searches ($n = k$)

Each experiment is run in separate processes, which repeat the trial 50 times. We pre-fill the map with n key-value pairs, because it is the expected number of elements in a map after infinitely many operations.

To minimize any overhead in Java implementations we use an up to date Java Server runtime, which compiles early, and allow the Java virtual machine to consume up to 3 GB memory. By comparison, all of the C++ implementations consumed less than 80 MB memory. It might seem strange to compare the performance of data structures implemented in Java with other data structures implemented in C++, but it is in fact quite commonplace [12]–[14]. As of August 2014 Intel only supports HTM "for software development", since "software using the Intel TSX (Transactional Synchronization Extensions) instructions may result in unpredictable system behaviour" [15]. The news site techreport.com reports that Intel intends to reintroduce support for hardware transactional memory in future processors [16]. We expect that resolving the bug present in Haswell processors will not affect the performance of the HTM.

B. Results

Figure 1 shows the throughput of maps as a function of the number of threads under 9 different workloads and key ranges. The plots are labeled with their workloads and key ranges (k). Figure 2 shows the number of L1 cache misses per operation, peak memory consumption, and energy consumption per operation, respectively, as measured by PAPI, version 5.40, `getrusage`, and Intel RAPL. We were only able to measure this data for the C++ map implementations because the measurement interfaces have APIs in C.

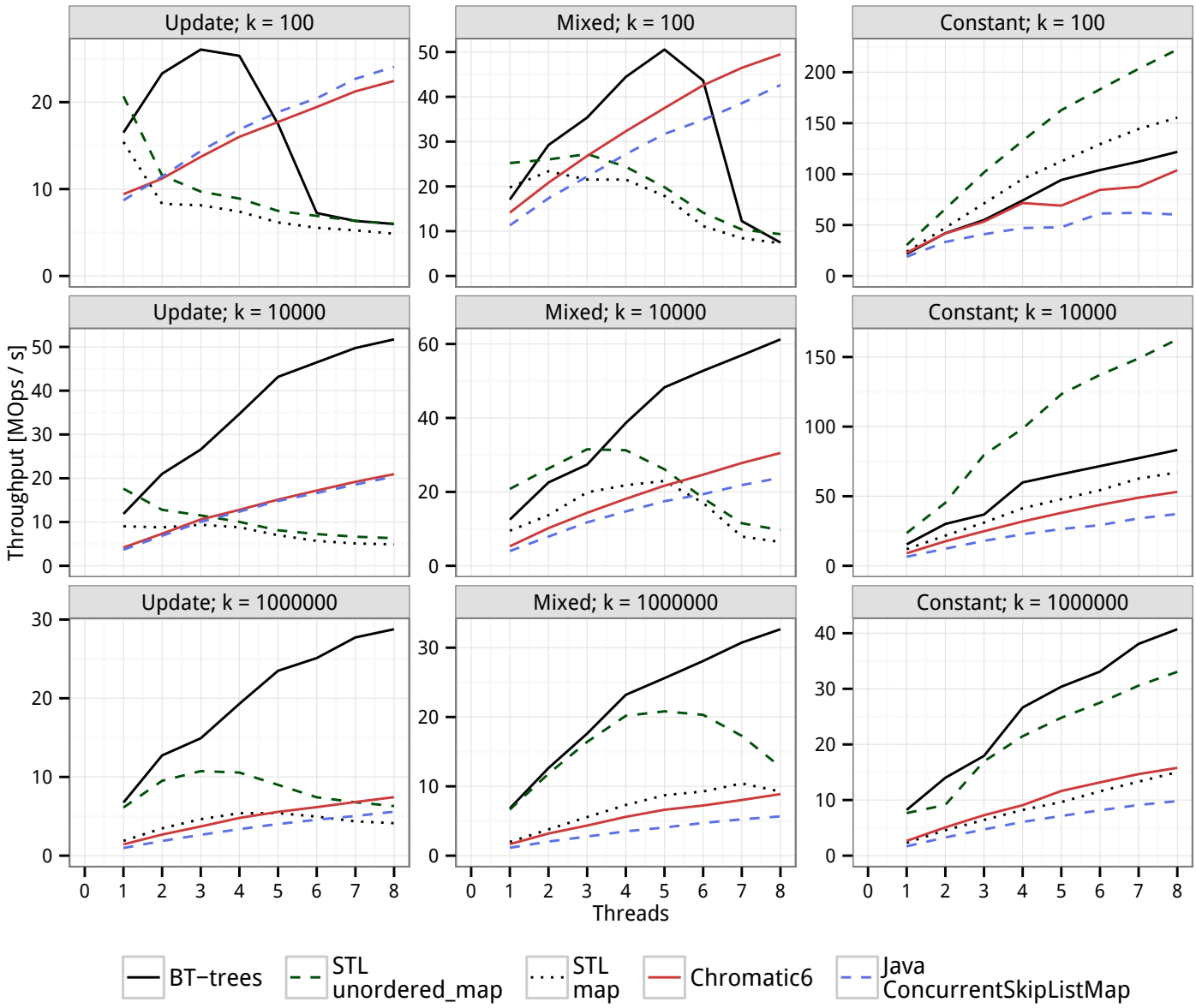

Fig. 1. Mean map throughput as a function of threads for 3 workloads and 3 key ranges.

BT-trees have the highest peak throughput out of the ordered maps in all workloads. BT-trees advantage is particularly high on large workloads ($k = 1,000,000$). The traditional concurrent maps are competetive with BT-trees on small workloads ($k = 100$), and even have higher throughput at 8 threads in the "Mixed; k = 100" and "Update; k = 100" workloads.

BT-trees do not scale well to more than 3 threads in the *Update* and *Mixed* workloads for the smallest key range $k = 100$ because the data structure is highly contended, as can be seen in Figure 2. The number of instructions per BT-tree operation increases when scaling beyond 3 threads. The increase is caused by two factors (1) the map operations are retried transactionally, and (2) acquiring the underlying lock executes more instructions when the locks are contended. By comparison, STL maps and unordered maps using lock-elision are contended on all of the *Update* and *Mixed* workloads. For instance the number of instructions executed per STL map operation triples when using 8 threads in the *Mixed* workload with $k = 1,000,000$ and the results are worse for the *Update* workloads and lower values of k.

The lock-elision based maps scale poorly when contended because of the lemming effect [5]: When operations fail to execute transactionally, they fall back to using the underlying lock. Using the underlying lock increases the risk that the following transactions fail. Once most operations use the lock, their combined throughput performance will decrease below that of single-threaded execution.

When $k > 100$ BT-trees achieve a 2.9-3.4 speedup with 4 threads, and a 4.3-5.6 times speedup with 8 threads. Using more threads does not significantly increase the number of executed instruction or L1 cache misses per BT-tree operation, indicating (1) transactions are rarely retried, and (2) there is little cache contention, hence BT-trees are not contended. We believe the sublinear scaling is caused by sharing the compute resources inherent to multicore (turboboost) and SMT

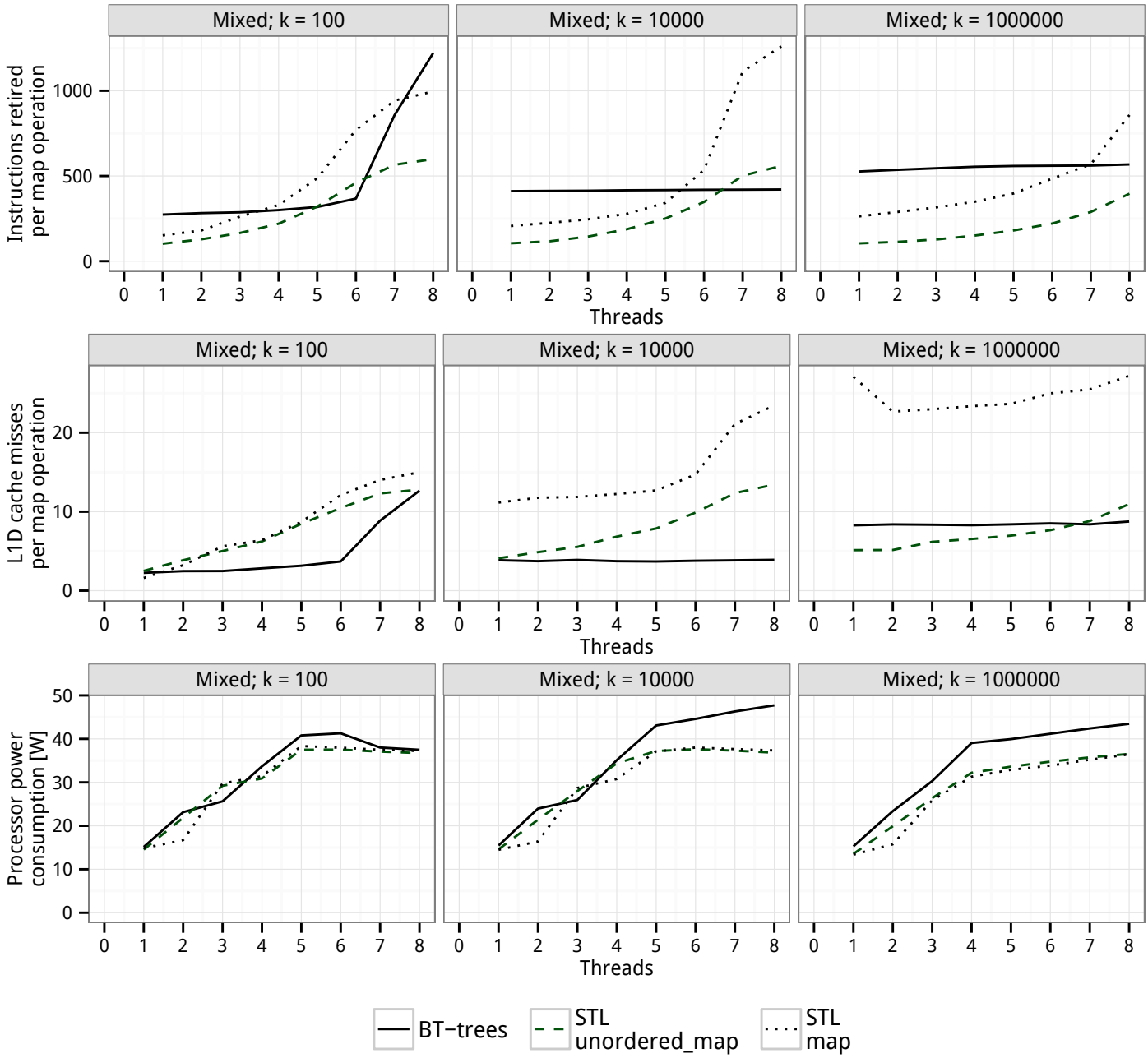

Fig. 2. Instructions retired, L1 cache misses, and processor power consumption as a functions of the number of threads for the *Mixed* workload on 3 key ranges.

(hyperthreading). Sharing compute resources is less attractive for BT-trees than the other maps, because BT-tree operations are compute bound and not memory bound: Compared to STL map operations, BT-tree operations execute 2-3 times as many instructions, but cause 1/4 as many cache misses, in the sequential case. BT-tree operations spend most of the time computing results, rather than waiting for cache misses, so their operations do not benefit as much from SMT, and they are more affected by lowering the processor's operating frequency. In summary, BT-trees are not significantly affected by contention when $k >= 10,000$, but they do not scale linearly because of hardware constraints.

Both BT-trees perform well when $k = 1,000,000$ compared to Chromatic trees and STL maps, because it benefits from being an external multiway trees, which helps its cache performance. Cache performance is more important for larger maps, which cause more cache misses per instruction. Figure 2 illustrates the cache performance for the *Mixed* workloads, where STL maps cause 3 times as many L1 cache misses as BT-trees, when $k > 100$. STL maps also cause 6 times as many L3 cache misses when $k = 1,000,000$, while the number of L3 cache misses are insignificant for lower k.

STL maps, and binary trees in general, cause more cache misses than multiway trees, because they are higher, and do

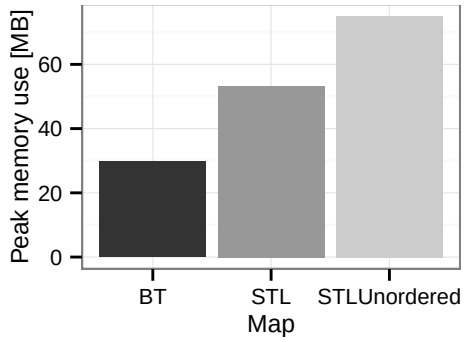

Fig. 3. Peak memory consumption when $k = 1,000,000$.

not fit as well into the cache. Sedgewick [17] observed that the average successful search in left-leaning red-black trees, such as STL maps, traverses $log_2(n)$ nodes, corresponding to 20 nodes in the *Constant* workload when $k = 1,000,000$. Our results show such operations cause approximately 21 L1 cache misses, which is higher than the expected number of nodes traversed, indicating that traversing red-black tree nodes references more than 1 cache line. By comparison the expected height of BT-trees is $h = log_{17.5}(\frac{n}{2}) + 1$, corresponding to 5.7 nodes in the *Constant* workload when $k = 1,000,000$, because the expected non-root node has $b = \frac{3+32}{2} = 17.5$ elements. Assuming that traversing BT-tree internal and leaf nodes both cause 2.5 cache line references ($\frac{1+2}{2} + 1 = 2.5$, $\frac{1+2+3+4}{4} = 2.5$), a successful search operation will reference $4.7 \cdot 2.5 + 2 = 13.75$ cache lines. Our results show that a successful search operation on such a BT-tree causes 8 cache line misses. BT-trees have a lower cache line miss to cache line reference ratio than red-black trees because they benefit from hardware prefetching and have better temporal locality. Internal BT-tree nodes change rarely, so they are less likely to be evicted, and more likely to stay cached by the cores.

We expect that Chromatic trees cause approximately as many cache misses as STL maps, but we do not have the infrastructure for fine grained measurement of Java code cache performance. STL maps and Chromatic trees have very similar structures, and as such they have similar sequential throughput, but chromatic trees have far better scalability. They mainly differ in how they are balanced: Chromatic trees are less balanced, and are more expensive to balance, but require less synchronization. Chromatic trees have a constant running time balancing, and the red-black tree property is violated for at most 6 nodes on any path. As a consequence, the sequential performance of the data structures mostly differs for small data sets.

Figure 2 shows the processor's power consumption. in the *Mixed* workload. Uncontended maps tend to be more energy efficient when using more threads. In the sequential case, BT-trees are more energy efficient than STL maps when $k >= 10,000$, and almost as energy efficient as STL `unordered_maps` and Intel TBB `concurrent_hash_maps` when $k = 1,000,000$. Generally BT-trees are slightly more attractive in the *Update* workload, and slightly less attractive in the *Constant* workload. When using 8 threads and $k > 100$, BT-trees generally consume 25% more power than STL maps, while still being far more energy efficient. BT-trees' higher power consumption comes from the cores, while the memory controller power consumption is lower. This is as expected, because operations on BT-trees execute many instructions, but incur few L3 cache misses.

Figure 3 shows the peak memory consumption we measured in the benchmarks. BT-trees' peak memory consumption is approximately 40MB lower than that of STL maps, which is approximately 25% lower than that of STL `unordered_maps`. When $k < 1,000,000$ all of the C++ maps have similar peak memory consumption, varying from 6MB to 15MB. We believe the maps have similar peak memory consumptions because most of the memory is consumed by factors other than the maps. Theoretically we would expect the binary trees use at least 32,000,000 bytes to represent 1,000,000 key-value pairs, as representing each key-value pair takes up to 32 bytes: 8 bytes for key-value pair, 16 bytes for child pointers, and 8 bytes for memory allocator data structures and 16 byte alignment. The estimate closely resembles 37 MB, the lowest memory consumption we observed for STL map in the *Constant* workload with $k = 1,000,000$. We would expect BT-trees to use 17,000,000 bytes to represent 1,000,000 key-value pairs: Every leaf node represents $\frac{3+32}{2} = 17.5$ key-value pairs, and there are approximately 17.5 times as many leaf nodes as internal nodes, giving the estimate $n\frac{\frac{384+16}{17.5}+(256+16)}{17.5} \approx 16,800,000$. The estimate closely resembles 18MB, the lowest memory consumption we observed in the *Constant* workload with $k = 1,000,000$.

In general, BT-trees have excellent space, time and energy performance compared to state of the art concurrent ordered maps, despite using much simpler coarse-grained synchronization.

VI. Related work

Transactional memory has been quite a hot topic in computer science since its introduction in 1993 [1]. Most of the research surrounding it has focused on how to expose transactional memory to programmers, and its hardware and software implementations [2]–[5], [18]. There has also been a several studies on the impact on software which applies transactional memory. Recently, emperical studies have confirmed that applying transactional memory is a simpler than traditional fine-grained synchronization methods [19], [20], but not neccesarily simpler than applying coarse grained synchronization.

Our work differs from prior work on transactional memory, by focusing on how to design software, in order to apply HTM efficiently. Our general approach, is to apply simple coarse grained synchronization based on SLR lock-elision [8] and redesign data layout to avoid limitations in HTM. On a related note, Bobba *et al* investigated performance pathologies for different simulated hardware implementations of HTM [21]. They found that different implementations of HTM will pathologically degrade performance under different workloads. In particular, transactional memory implementations which resolve conflicts with requester-wins policies, similar to the Intel and IBM implementations, are have particularly pathological case called *Friendly Fire*: transactions which conflict with other transactions are likely to fail themselves. We use HTM in the form of lock-elision, which suffers from the Lemming Effect, giving pathological performance in the same situation [5].

BT-trees, which we designed to avoid the limitations of HTM, can be seen as a simplification and further relaxation of PO-B+trees [22]. PO-B+trees are slightly relaxed B+trees which are balanced while traversing, which in turn allows them only holding two lock while searching. BT-trees mainly differs from PO-B+trees by using coarse grained lock-elision, as opposed to fine-grained locks, representing leaf nodes as unordered arrays to avoid transactional conflicts, and allow internal nodes to have fewer children to further reduce balancing. The attempt to reduce balancing is seen in other concurrent data structures such as chromatic trees, skip lists, Ctries [12], [23], [24], and in paticular for SF-trees [25].

SF-trees are concurrent ordered red-black trees, which also optimize for transactions and reduce balancing, but with different means for avoiding conflicts [25]: BT-trees reduce the frequency of balancing, whereas SF-trees defer balancing to another thread; SF-trees defer removing nodes from the tree to avoid conflicts near the root, while BT-trees are multiway external search trees, making writes near the root infrequent. The main similarity of BT-trees and SF-trees is that they both use individual transactions for balancing, rather than allowing an operation and balancing in the same transaction.

VII. Conclusion

Modern multi core processors offer increasing parallel power, but writing efficient parallel code is still difficult. In this paper we illustrated a simple way of writing efficient parallel code applying hardware transactional memory. The main idea is to reason about how the code affects the synchronization, rather than custom tailoring new synchronization schemes. We presented 5 guidelines that can help detect scalability pitfalls, and applied the guidelines to the design and implementation of BT-trees, a new ordered map. BT-trees are 3 times faster and twice as space efficient as state of the art concurrent ordered maps. Unlike other state of the art ordered maps, BT-trees use very simple synchronization. Using the same synchronization on traditional maps, which were not designed according to our guidelines, results in massive synchronization contention, and limited scalability.

Acknowledgment

This article presents the result of a research and development work carried out in the European collaborative project PaPP (Portable and Predictable Performance on Heterogeneous Embedded Manycores) funded jointly by the ARTEMIS Joint Undertaking and national governments under the Call 2011 Project Nr. 295440.

References

[1] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.

[2] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[3] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui, "Why stm can be more than a research toy," *Communications of the ACM*, vol. 54, no. 4, pp. 70–77, 2011.

[4] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust architectural support for transactional memory in the power architecture," in *ISCA*, pp. 225–236, ACM, 2013.

[5] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski, "Early experience with a commercial hardware transactional memory implementation," tech. rep., Sun Microsystems, Inc., 2009.

[6] Intel, *Programming with Intel Transactional Synchronization Extensions*, June 2014.

[7] Y. Afek, A. Levy, and A. Morrison, "Software-improved hardware lock elision," in *PODC*, pp. 212–221, ACM, 2014.

[8] A. Levy, "Programming with hardware lock elision," Master's thesis, Tel-Aviv University, September 2013.

[9] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *ISCA*, pp. 81–91, ACM, 2007.

[10] L. Bonnichsen, C. Probst, and S. Karlsson, "Implementation of BT-trees," tech. rep., arxiv.org, 2015.

[11] T. Brown, "Personal homepage." [Online; Last accessed 12/20/2014].

[12] T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," in *PPoPP*, pp. 329–342, ACM, 2014.

[13] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *PPoPP*, pp. 317–328, ACM, 2014.

[14] D. Drachsler, M. T. Vechev, and E. Yahav, "Practical concurrent binary search trees via logical ordering," in *PPoPP*, pp. 343–356, ACM, 2014.

[15] Intel, "Intel® xeon® processor e3-1200 v3 product family specification update," August 2014. [Online; Last accessed 09/08/2014].

[16] S. Wasson, "Errata prompts intel to disable tsx in haswell, early broadwell cpus," August 2014. [Online; Last accessed 09/08/2014].

[17] R. Sedgewick, "Left-leaning red-black trees," in *Dagstuhl Workshop on Data Structures*, p. 17, 2008.

[18] K. Fraser, *Practical lock-freedom*. PhD thesis, University of Cambridge, 2004.

[19] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?," in *ASPLOS*, pp. 47–56, ACM, 2010.

[20] V. Pankratius and A.-R. Adl-Tabatabai, "A study of transactional memory vs. locks in practice," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pp. 43–52, ACM, 2011.

[21] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *ISCA*, vol. 35, pp. 81–91, ACM, 2007.

[22] Y. Mond and Y. Raz, "Concurrency control in b+-trees databases using preparatory operations.," in *VLDB*, pp. 331–334, 1985.

[23] W. Pugh, "Concurrent maintenance of skip lists," Tech. Rep. TR-CS-2222, Dept. of Computer Science, University of Maryland, 1990.

[24] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *PPoPP*, pp. 151–160, ACM, 2012.

[25] T. Crain, V. Gramoli, and M. Raynal, "A speculation-friendly binary search tree," *Acm Sigplan Notices*, vol. 47, no. 8, pp. 161–170, 2012.