THE UNIVERSITY OF SYDNEY

SCHOOL OF INFORMATION TECHNOLOGIES

SPACE-CONSTRAINED DATA STRUCTURES FOR HTM

TECHNICAL REPORT 708

NICK ARMSTRONG, PASCAL FELBER, VINCENT GRAMOLI

JANUARY 2018

Space-Constrained Data Structures for HTM

Nick Armstrong
University of Sydney, Australia
narm6003@uni.sydney.edu.au

Pascal Felber
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

Vincent Gramoli
University of Sydney, Australia
vincent.gramoli@sydney.edu.au

Abstract

Most of the research efforts to improve performance of programs based on hardware transactions have been devoted to designing new hybrid transactional memories and transactional lock elision algorithms to speed up software fallback paths.

Our proposal is instead to redesign the data structures accessed by these programs. To this end, we propose a novel class of concurrent data structures, called *space-constrained data structures*, especially designed to boost programs based on hardware transactions. To illustrate this idea we specify a concurrent van Emde Boas tree with insertions and deletions of time complexity $O(\log\log M + \dot{c})$ where M is the size of the key range and $\dot{c}$ is the point contention. We propose a resize operation to make the concurrent sorted tree dynamic, hence bypassing the drawback of bounded-universe structures.

Our experiments on IBM Power8 and Intel Skylake show that our space constrained tree outperforms HTM binary search trees by up to one order of magnitude on Synchrobench and STAMP benchmarks.

1 Introduction

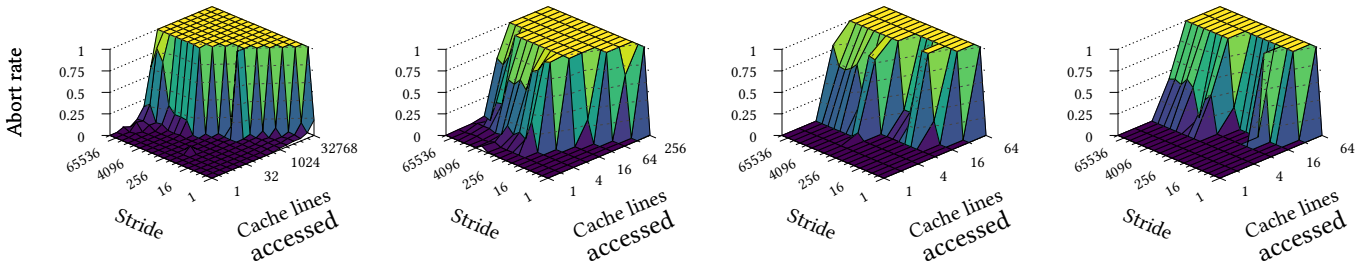
Transactional memory is now supported in hardware in modern processors, ranging from Intel Haswell and Skylake to IBM Power8 microarchitectures. The hardware support promises potential performance improvement compared to the traditional software transactional memory libraries. Since hardware transactions are inherently limited by physical capacities, a software fallback mechanism is necessary. The solution is to use either *hybrid transactional memory* [9] with best-effort hardware transactions and software transactions or *transactional lock elision* [36] that consists of trying to speculatively execute a critical section in a fast path hardware transaction before reverting potentially to a fallback path that handles the synchronization in software.

Unfortunately, the cost of the software fallback often annihilates the benefit of the hardware transaction. This led the community to focus on reducing the cost of the software path [1, 8, 10, 12, 16]. Lazy subscription [8] allows the hardware transaction postpone its subscription to the lock, however, this may lead to inconsistencies [10]. Read-write lock-elision exploits the suspend/resume features of the Power8 to avoid using hardware transactions and to guarantee progress of read-side critical sections [16]. Amalgamated lock elision exploits fine-grained locks in the fallback path to detect conflicts with the fast path [1]. Allowing only one software thread to hold the lock [12] simplifies the software path to offer a middle-ground between hybrid transactional memory and transactional lock elision. Most of these solutions have shown promising performance improvements on binary search trees, either used as micro-benchmarks [38] or as the database index of more elaborate applications [31].

In contrast with this body of work, we propose to improve performance by devising *space-constrained data structures* synchronised by hardware transactions. The challenge lies in constraining the data structure size to maximize the success of hardware transactions and avoid the need to revert to the slow software fallback path. In particular, we suggest the exploration of data structures whose universe is bounded appropriately to fit within the hardware boundaries [39]. Besides the inherent fit for hardware transactions, these sequential data structures offer asymptotically better time complexity in $O(\log\log M)$ than the classically used $O(\log n)$ data structures with n elements taken from the range of size M. This lower access complexity translates into a smaller access set that the hardware transaction needs to track, hence maximizing the chance for the transaction to commit without exploiting any software support.

Unfortunately, bounded-universe structures typically suffer from a major limitation, which is probably the reason why they are rarely implemented [34, 39]. In fact, a bounded-universe structure has by definition a limited storage capacity: if an element added to this structure does not belong to a pre-defined universe, then the structure cannot store this element. Although one could potentially encode elements in a dense universe to maximize insertions, there is always a risk that the number of elements inserted reaches a threshold after which these elements would be lost. Although bounded-universe structures could work well for caching some data items that can be found elsewhere, they do not apply to general search structures that should guarantee element retrieval.

We present a space-constrained tree resulting from a concurrent variant of the van Emde Boas tree [39] and called *CvEB*. To make CvEB concurrent, we synchronize this tree structure with hardware transactions using a classic hardware lock elision coping with the lemming effect [11]. To cope with the inherent limitation of bounded-universe data structures, we propose a dynamic version of CvEB, called *DCvEB*. DCvEB simply detects when the key to be inserted does not belong to the universe or *range* and doubles the range by merging a new empty tree to the existing one until the key falls into the range. This merge proceeds concurrently with all other operations of the tree. Surprisingly, we



(a) Intel - Read-only **(b)** Intel - Write-only **(c)** IBM - Read-only **(d)** IBM - Write-only

Figure 1. Hardware transaction capacity on Intel Skylake and IBM Power8.

observed that the resizing (sufficiently rare in various workloads of the `vacation` applications) is largely compensated by the gain of using a space-constrained data structure.

We evaluate our space-contrained tree structure on two architectures, Intel Skylake and IBM Power8, while running Synchrobench [19] and the STAMP `vacation` application [31]. Synchrobench is a micro-benchmark suite for evaluating synchronization techniques. We added support for hardware transactions to Synchrobench, which allowed us to evaluate its performance under finely tuned workloads. The STAMP [31] `vacation` application was originally designed to benchmark the performance of software and simulated memory transactions. It mimics a travel reservation system with tables implemented as red-black trees.

Our results show that CvEB outperforms the STAMP transactional red black tree instrumented with Hardware Transactional Memory (HTM) and the fastest HTM binary search tree we know [38] on both Intel Skylake and IBM Power8. First, on Synchrobench, our space constrained tree improves the red-black tree performance by one order of magnitude. Interestingly, the restructuring does not impact the performance significantly, even though it requires copying the whole data structure to resize it. Second, for the default `vacation` application workloads, our space constrained data structure is almost twice as fast as the original red-black tree. We confirmed experimentally that this gain in performance is due to a lower abort ratio than on red-black trees. This is due to the constraint our structure inherently imposes on the size of transactions.

The paper is organized as follows. Section 2 illustrates the problem. Section 3 introduces the concept of space constrained data structures and the concurrent van Emde Boas tree. Section 4 presents the experimental evaluation while Section 5 presents the time and space complexities of the CvEB and DCvEB. Section 6 presents the related work and Section 7 concludes the paper.

2 The Capacity Problem

A memory transaction [27] exploits metadata to record its sequence of accesses and decide whether to commit or abort upon conflicts. These access sets are called the *read set* and *write set* of the hardware transaction. The read and write sets are stored in hardware structures, like caches, whose storage capacity is limited. These caches are hierarchical and organized into sets that contain a number, typically 8 or 16, of ways. This number is known as the *associativity* of the cache. When data is loaded into a processor cache, it is mapped to a set based on its address and within the set it can occupy any of the ways. At runtime, if a new access cannot be recorded in the structure, then the corresponding hardware transaction aborts. To avoid continuously aborting the same transaction, a software fallback path is needed.

To better understand how one can limit the use of the software path to improve performance, we measured the frequency of aborts of sequential hardware transactions on the Intel Skylake (i5-6500) and the IBM Power8 machines, using the capacity benchmark of Hasenplaugh, Nguyen and Shavit [23]. The frequency of aborts was recorded while increasing the number of cache lines accessed in a transaction and altering the cache line stride. A stride of k indicates that only the first of k consecutive cache lines are accessed. Depending on the associativity of the caches used, a stride of $k > 1$ can increase the chances of abort as it may cause cache lines to be mapped to fewer sets.

Figure 1a shows that an Intel Skylake processor is able to successfully read at least 65536 consecutive cache lines in a single transaction, suggesting that the read set is tracked in the L3 cache, as already suggested for the Intel Haswell processor [23]. Unlike the result for Haswell, we find that even with a very large stride, 64 cache lines can still be consistently read which is more than the associativity of the L3 (i.e., 16). However, the results for the write-only transactions shown in Figure 1b match the expected outcome. The L1 cache consists of 512 cache lines, so it is unlikely that a transaction consisting of writes to 512 consecutive cache lines (the next size tested above 256) would ever commit. We see the effects of associativity as the number of successfully written cache lines does not drop below 8, the associativity of the L1 cache.

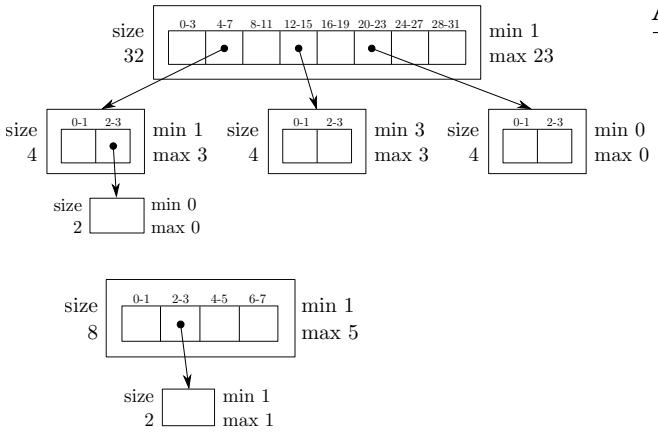


Figure 2. An vEB tree with 1, 5, 6, 7, 15, 20 and 23 inserted (top) and its associated auxiliary tree (bottom).

The IBM Power8 is particularly sensitive to the length of hardware transactions as it offers a relatively small data structure to store the read and write sets in comparison to the size of the caches used by Intel Skylake processors. Figure 1c and Figure 1d depict the capacity of the Power8 and confirms that the read and write sets of hardware transactions cannot span more than 64 cache lines, regardless of their placement. A Power8 core keeps transactional stores in its 8-way 512KB L2 cache. This cache has 4096 lines, but only 64 can be tracked by the HTM system per core. Additionally, at least 8 cache lines can always be read from or written to, illustrating the effects of associativity. No matter the stride, a single set in the L2 cache will always be able to accommodate 8 cache lines due to its associativity.

3 Space Constrained Structures

In this section, we present a new class of data structures well-suited for HTM, called the *space constrained data structures*. We first present our implementation of the van Emde Boas (vEB) tree, then explain how to synchronize it into a concurrent vEB (CvEB) tree and finally making it a dynamic CvEB (DCvEB) tree.

3.1 The van Emde Boas tree as an example

The van Emde Boas tree (vEB tree) [39] is a sequential data structure of size $M = 2^k$ for integral k that can hold integer elements in the range $[0, M)$. First, we introduce the *upper square root* and *lower square root* denoted by $\sqrt[\uparrow]{M} = 2^{\lceil \log_2 M/2 \rceil}$ and $\sqrt[\downarrow]{M} = 2^{\lfloor \log_2 M/2 \rfloor}$, respectively. Then, we recursively define the vEB tree structure as follows. A vEB tree of size M consists of $\sqrt[\uparrow]{M}$ child vEB trees, each of size $\sqrt[\downarrow]{M}$. Additionally, each tree maintains an auxiliary vEB tree of size $\sqrt[\uparrow]{M}$ to keep track of the indices of nonempty children. The minimum and maximum values stored in a tree are only stored in the root of the tree and not in any of its children. See Figure 2 for an illustration of a vEB tree and its associated auxiliary tree.

Algorithm 1 vEB tree: contains, insert, remove.

```
1:  function Contains(tree, val)
2:    if tree is empty then                                  # tree already empty
3:      return Failure
4:    if val ∉ [tree.min, tree.max] then                     # value out of range
5:      return Failure
6:    if val = tree.min ∨ val = tree.max then                # in tree
7:      return Success
8:    child_size ← ⌈√tree.size⌉
9:    child_idx ← val/child_size
10:   return Contains(tree.children[child_idx], val%child_size)
11: end function

12: function Insert(tree, val)
13:   if val ∉ [0, M) then                                   # value out of range
14:     return Failure
15:   if val = tree.min ∨ val = tree.max then                # in tree
16:     return Failure
17:   if tree is empty then                                  # 1st in tree
18:     tree.min ← val
19:     tree.max ← val
20:     return Success
21:   if tree.min = tree.max then                            # 2nd in tree
22:     tree.min ← min(tree.min, val)
23:     tree.max ← max(tree.max, val)
24:     return Success
25:   if val < tree.min then
26:     swap(tree.min, val)
27:   else if val > tree.max then
28:     swap(tree.max, val)
29:   child_size ← ⌈√tree.size⌉
30:   child_idx ← val/child_size
31:   if tree.children[child_idx] is empty then              # 1st in child
32:     Insert(tree.aux, child_idx)
33:   return Insert(tree.children[child_idx], val%child_size)
34: end function

35: function Remove(tree, val)
36:   if tree is empty then                                  # tree already empty
37:     return Failure
38:   if val ∉ [tree.min, tree.max] then                     # value out of range
39:     return Failure
40:   if tree.min = tree.max then                            # removing the last value
41:     mark tree as empty
42:     return Success
43:   child_size ← ⌈√tree.size⌉
44:   child_idx ← val/child_size
45:   child ← tree.children[child_idx]
46:   remove_val ← val mod child_size
47:   if val = tree.min then                                 # removing minimum value
48:     if tree.aux is empty then                            # ...and it is 2nd last value
49:       tree.min ← tree.max
50:       return Success
51:     child_idx ← tree.aux.min
52:     child ← tree.children[child_idx]
53:     tree.min ← child_size × child_idx + child.min
54:     remove_val ← child.min
55:   else if val = tree.max then                            # removing maximum value
56:     if tree.aux is empty then                            # ...and it is 2nd last value
57:       tree.max ← tree.min
58:       return Success
59:     child_idx ← tree.aux.max
60:     child ← tree.children[child_idx]
61:     tree.max ← child_size × child_idx + child.max
62:     remove_val ← child.max
63:   r ← Remove(child, remove_val)
64:   if r = Success ∧ child is empty then                   # last from child
65:     Remove(tree.aux, child_idx)
66:   return r
67: end function
```

3.2 Constraining the storage space

Naively, this structure will take $O(M)$ space as memory must be allocated for every possible value in our universe of M

Algorithm 2 vEB: successor, predecessor.

```
68: function Successor(tree, val)
69:   if val < tree.min then
70:     return tree.min
71:   if val ≥ tree.max then                    # there is no next element
72:     return ∞
73:   child_size ← ∜tree.size
74:   child_idx ← val/child_size
75:   child_val ← val mod child_size
76:   if child_val ≤ tree.children[child_idx].max then
77:     idx ← Successor(tree.children[child_idx], child_val)
78:     return child_size × child_idx + idx
79:   child_idx ← Successor(tree.aux, child_idx)
80:   return child_size × child_idx + tree.children[child_idx].min
81: end function

82: function Predecessor(tree, val)
83:   if val > tree.max then
84:     return tree.max
85:   if val ≤ tree.min then                    # no previous element
86:     return −∞
87:   child_size ← ∜tree.size
88:   child_idx ← val/child_size
89:   child_val ← val mod child_size
90:   if child_val ≥ tree.children[child_idx].min then
91:     idx ← Predecessor(tree.children[child_idx], child_val)
92:     return child_size × child_idx + idx
93:   child_idx ← Predecessor(tree.aux, child_idx)
94:   return child_size × child_idx + tree.children[child_idx].max
95: end function
```

elements. We improve on this by only allocating memory to child and auxiliary trees when they are needed, so in practice the memory usage is reduced for trees holding few elements. However even in these cases there remains a rather large overhead of $O(\sqrt{M})$ due to the root node. The algorithms used to implement our space-constrained tree are described in Algorithm 1 and Algorithm 2. We can show, as stated in Theorem 3.1, that the time complexity of operations is $O(\log\log M)$ by forming a recurrence relation and then solving. The minimum and maximum operations are exempt as they are trivially implemented in constant time.

Theorem 3.1. *The time (and access) complexity of any operations in Algorithm 1 and Algorithm 2 is* $O(\log\log M)$.

3.3 CvEB: Synchronizing vEB with HTM

To add HTM support to Synchrobench and `vacation`, basic `TM_START` and `TM_END` macros similar to [33] were added with the architecture specific transactional memory functions provided by GCC.[1] A transaction has a maximum number of retries before it gives up and falls back to locking.

```
int vebtree_htm_op(tree, value) {
  TX_START;
  int ret = vebtree_seq_op(tree, value);
  TX_END;
  return ret;
}
```

Figure 3. Using HTM to protect a sequential operation.

Each of the contains, insert, remove, successor and predecessor operations is encapsulated in a transactional wrapper

[1] https://gcc.gnu.org/onlinedocs/gcc/Target-Builtins.html.

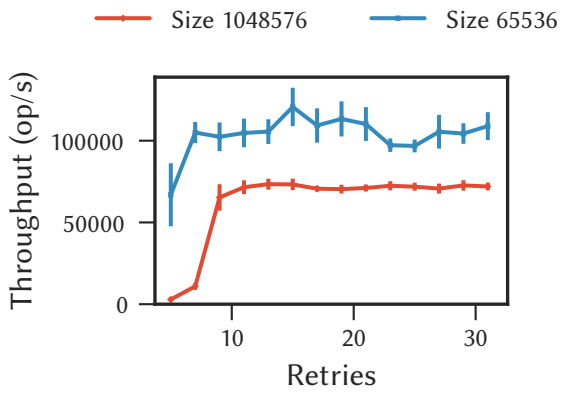


Figure 4. The impact of the number of retries on performance on Power8 with 80 threads running Synchrobench.

whose implementation is a classic hardware lock elision: the wrapped sequence of accesses tries executing as a hardware transaction several times, before falling back to a software path that acquires a global lock to execute the sequence as a critical section. To avoid the lemming effect when a lock is taken and aborts all the other threads, threads must first wait until the lock becomes free before retrying transactional execution once more. We tested the performance of the resulting CvEB tree in two different sizes with 80 threads running Synchrobench [19] on the Power8 machine. We measured the impact of the maximum number of retries before falling to the fallback path. As depicted in Figure 4 we observed that too few retries would unnecessarily trigger the software path whereas too many retries would not lead to performance improvements. This is why we set the maximum number of retries to 15.

3.4 DCvEB: Dynamic growth tree

It is possible to extend the CvEB tree to support dynamic range increases. When an attempt is made to insert an element greater than the maximum possible the range is doubled.

There are two cases to consider when doubling the size of a CvEB tree of size M. In the case when $\log M$ is even, the children in the doubled tree will all be the same size as the original children, but double in number. We can simply copy the children into a new CvEB tree of size $2M$ and double the size of the original auxiliary tree. When $\log M$ is odd, the doubled tree will have the same number of children but they will be twice the size of the original children. We must instead merge consecutive pairs of children and place the merged children into a new CvEB tree of size $2M$, maintaining the new auxiliary tree in the process.

Algorithm 3 contains this doubling algorithm and updated insert operation. The ability to merge two trees is required for doubling in the odd case, so tree doubling is implemented by merging the original tree with an empty tree. As for the CvEB tree, the concurrent version of the DCvEB is obtained

Algorithm 3 Dynamic CvEB (DCvEB): merge, double, insert.

```
96:  function MERGE(left_tree, right_tree)
97:    size ← left_tree.size
98:    new_tree ← an empty vEB tree of size size × 2
99:    num_children ← left_tree.num_children
100:   if log size is odd then                    # merge consecutive children pairs
101:     pairs ← num_children/2
102:     for i ← 0, pairs do
103:       left_child1 ← left_tree.children[2 × i]
104:       left_child2 ← left_tree.children[2 × i + 1]
105:       left_merged ← MERGE(left_child1, left_child2)
106:       new_tree.children[i] ← left_merged
107:       if left_merged is not empty then          # update auxiliary
108:         INSERT(new_tree.aux, i)
109:       right_child1 ← right_tree.children[2 × i]
110:       right_child2 ← right_tree.children[2 × i + 1]
111:       right_merged ← MERGE(right_child1, right_child2)
112:       new_tree.children[pairs + i] ← right_merged
113:       if right_merged is not empty then         # update auxiliary
114:         INSERT(new_tree.aux, pairs + i)
115:   else                                         # copy children in new tree
116:     COPY(left_tree.children, new_tree.children[ : num_children])
117:     COPY(right_tree.children, new_tree.children[num_children : ])
118:     new_tree.aux ← MERGE(left_tree.aux, right_tree.aux)
119:   if left_tree is empty then
120:     new_tree.min ← size + right_tree.min
121:     new_tree.max ← size + right_tree.max
122:   else if right_tree is empty then
123:     new_tree.min ← left_tree.min
124:     new_tree.max ← left_tree.max
125:   else
126:     new_tree.min ← left_tree.min
127:     new_tree.max ← size + right_tree.max
128:     INSERT(new_tree, left_tree.max)
129:     INSERT(new_tree, size + right_tree.min)
130:   return new_tree
131: end function

132: function DOUBLE(tree)
133:   empty_tree ← an empty vEB tree of size tree.size
134:   return MERGE(tree, empty_tree)
135: end function

136: function INSERT_DYNAMIC(tree, val)
137:   if val ≥ tree.size then
138:     tree ← DOUBLE(tree)
139:     return INSERT_DYNAMIC(tree, val)
140:   return INSERT(tree, val)
141: end function
```

by adding the transactional wrapper to Algorithm 3 (cf. Figure 3).

3.5 Other space-constrained structures

There are a number of ways to design space-constrained data structures. While our space constrained data structure relies on a concurrent and dynamic variant of van Emde Boas tree, other space-constrained data structures could result similarly from concurrent and dynamic variants of x-fast tries, y-fast tries, fusion trees or skip tries.

Arrays. We can trivially implement an integer set using indexing by value. The existence of an element v in the set S depends only on the value of $S[v]$. If we implement an integer set using an array we have the following complexities for the set operations: insertion, deletion, and contains in $O(1)$, minimum, maximum, successor, predecessor in $O(M)$. While this gives us very good time complexity in the most popular operations, we are limited by the memory cost which is $O(M)$ as the entire array must exist for any value in the universe to be stored.

***-fast tries.** The x-fast trie [40] is a structure based on a bitwise trie that has a space usage of $O(n \log M)$. Each level in the trie has a hash table that contains the nodes for that level. Leaf nodes form an ordered linked list and internal nodes maintain pointers to certain leaf nodes in place of missing child nodes. It maintains $O(\log \log M)$ successor and predecessor operations, improves the contains operation to $O(1)$, but the insert and remove become $O(\log M)$.

The y-fast trie [40] improves on the x-fast trie by reducing the space usage to $O(n)$ and matching the time complexities of the vEB tree (although insert and remove are amortized). It is essentially an x-fast trie with a collection of balanced binary search trees (e.g., red black trees) at the leaves. Each red black tree contains between $\log M/4$ and $2 \log M$ elements.

Fusion trees. A fusion tree is a tree structure that stores k-bit integers [17] (so $M = 2^k$). It uses $O(n)$ space and its operations take $O(\log_k n)$. This is achieved by compressing keys to fit in a machine word and using a B-tree structure with a branching factor of $k^{1/5}$. There exist methods to make fusion trees dynamic [2, 37].

Skip tries. The skip trie [34] is already a concurrent data structure exporting predecessor, insertion and deletion operations in $O(\log \log M)$ amortized complexity when the maximum number of concurrent operations at any point in the execution is $O(1)$. To make it a space constrained data structure, one would simply have to make it dynamic, however, the structure is more complex than a x-trie as it features a concurrent hash table, a concurrent skip list, a concurrent x-fast trie and a doubly-linked list. Hence it is unclear whether implementing a dynamic version of a skip trie would be easier than implementing a concurrent and dynamic variant of an x-trie. To date, we are not aware of any existing implementation of a skip trie.

Many extensions and improvements have been made to structures such as tries and radix trees that may also make them good space-constrained data structures to explore with HTM.

4 Evaluation

In this section we compare our space-constrained data structure against classic data structures on two multicore machines using Synchrobench [19] and STAMP `vacation` [31]. One machine is a 4-core Intel machine with a Skylake processor supporting Intel Transactional Synchronization Extensions (TSX) with 32 GB of memory, and running Fedora 21, `perf` v3.13.11-ckt35 and gcc v4.8.4. The other is a 10-core POWER8 machine also supporting HTM, each core having 8 hardware contexts, 16 GB of memory, and running Ubuntu

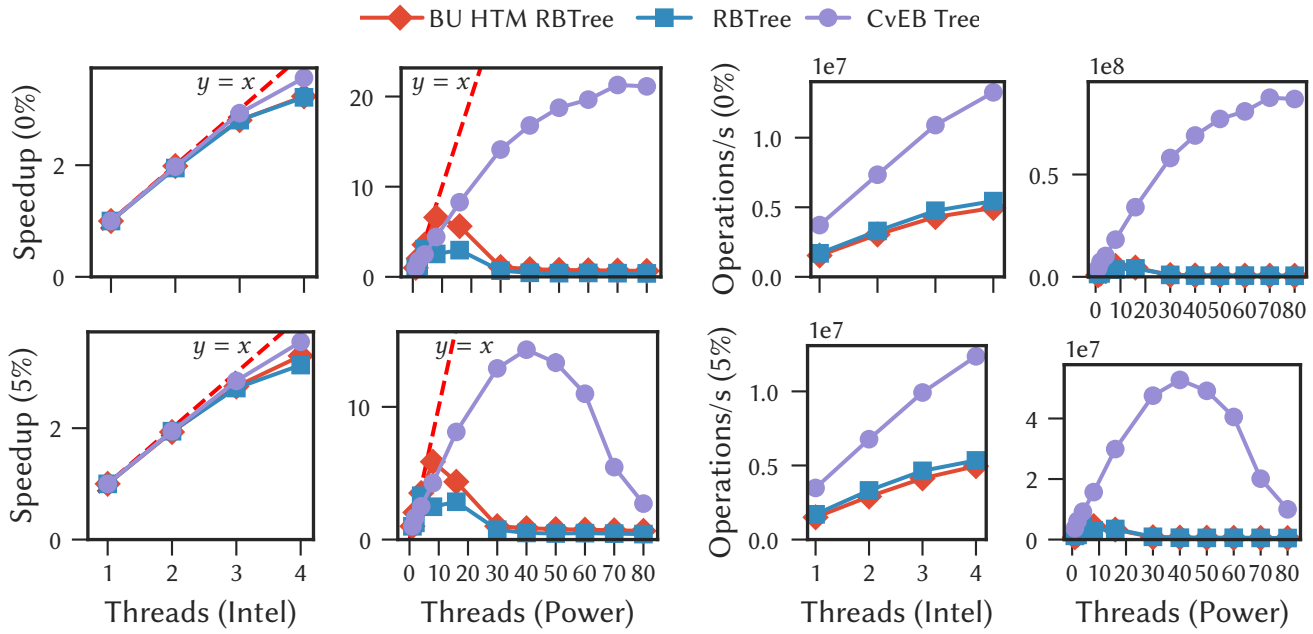


Figure 5. Speedup and absolute throughput of size 2^{20} trees with 0% and 5% updates

14.04.4 LTS, `perf` v4.7.4.200.fc24.ppc64.gf11a and gcc v6.2.1. Our experiments focus on three data structures: our DCvEB/CvEB, the classic transactional red-black tree implemented by Oracle [19] (RBTree) and its more recent bottom-up variant tuned for HTM (BU HTM RBtree) [38]. We did not use any explicit pinning or binding and on the Intel machine, we replaced the default glibc malloc by the more concurrency-friendly TCMalloc memory allocator.

Synchrobench. The Synchrobench benchmark was used to determine the scalability of our space-constrained data structure and compare it to that of the two red black trees. Combined with `perf`, this allows us to see the effect of aborts on throughput. We compare the scalability of the different data structures and the absolute performance. We present results with tree sizes of $2^8 - 2^{26}$ and various workloads (0% to 100% update ratio).

Figure 5 (left) depicts the speedup in terms of the throughput scaled to single thread performance on the Intel machine and the Power8 machine. As the number of threads are increased we observe a nearly linear speedup for all three data structures on Intel but our space-constrained data structure shows slightly better scalability with more threads. On the Power8 (Figure 5, 2nd column), the speedup for the CvEB tree is not as close to linear as it is on the Intel machine. We also see the red black trees outperforming the CvEB in terms of speedup at low thread counts. The bottom up red black tree manages to outscale the CvEB tree up to 8 threads. However after that, the CvEB tree exhibits strong scalability up to 80 threads in a read only workload while the red black trees drop quickly in performance.

Although the speedup of the data structures is rather similar, the absolute performance of the CvEB tree far exceeds that of the red black trees as shown in Figure 5 (right). In particular at 5% update, the peak performance obtained by the CvEB on the Power8 (on 40 threads) is 17× higher than the peak performance of the regular red-black tree (on 16 threads) and 11× higher than the peak performance of the bottom-up red-black tree (on 8 threads). With any thread count, the CvEB tree is capable of performing twice as many operations per second. This is probably due to the asymptotic runtime of the CvEB tree ($O(\log \log M)$) compared to the red black trees ($O(\log n)$). If n is a reasonable fraction of M then the CvEB tree should have a lower runtime. Synchrobench is designed so that n remains close to the initial size parameter. In these experiments, this is set to $M/2$ so this condition definitely holds. In terms of raw throughput on Power8, the CvEB tree once again outclasses the red black trees. In the read only workload we see the CvEB tree peaks at around 87 million operations per second on 70 threads while the bottom up red black tree manages 5 million operations per second on 8 threads. As the update ratio is increased, the peak throughput of the CvEB tree drops and also occurs at a lower thread count. The drop in throughput is expected as update operations introduce contention which causes transactions to retry when conflicting memory reads and writes occur.

On Intel, the number of aborts per operation is calculated as the (#abort / throughput) plotted against the thread count (cf. Figure 6 left) where #aborts corresponds to the counter `RTM_RETIRED.ABORTED`. We plot the aborts per operation rather than the aborts per second as a data structure with a higher throughput (operations per second) may have more chances to cause an abort than a slower structure. The abort rate for all structures appears rather low. However, it can be

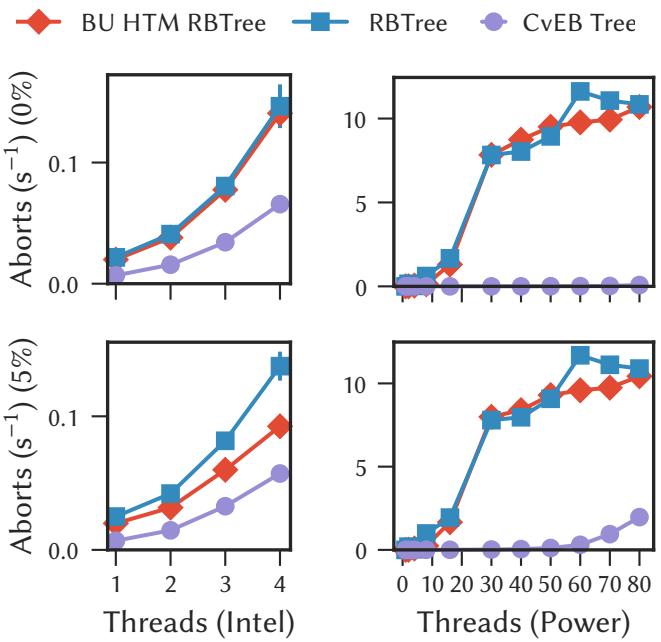


Figure 6. Transactions aborted per operation on size 2^{20} trees with 0% and % updates

interpreted as around one in ten operations on a red black tree causes an abort, compared to just under one in twenty for the CvEB tree. This may not be strictly true, as a single operation can cause up to 15 aborts before falling back to the software path.

We see that the abort rate does not directly relate to the absolute throughput (the RBTree has a higher throughput than the BU HTM RBTree, but a higher abort rate) but it may influence the scalability of the data structure. Structures with lower abort rates have speedups that are closer to linear.

One of the keys to the scalability shown by the CvEB tree on Power8 may be the very low abort rates it causes during operations. Figure 6 (right) shows that the CvEB tree has effectively zero aborts per operation in a read only workload, and the abort rate stays below 2 aborts per operation in a 5% update case. On Power8, we calculated (#capacity aborts + #transactional conflict aborts + #non-transactional conflict aborts) / throughput, which corresponds to `PM_TM_FAIL_FOOTPRINT_OVERFLOW` + `PM_TM_FAIL_TX_CONFLICT` + `PM_TM_FAIL_NON_TX_CONFLICT` / throughput. The red black trees show very different behavior, reaching over 8 aborts per operation on just 30 threads. This means that potentially every second operation is falling back to acquiring a lock, severely limiting performance as a global lock would be highly contended in such a situation.

As the update ratio increases, there is a trend towards lower peak throughput at lower thread counts, as shown briefly in Figure 5 (right). This trend is presented further in Figure 7. For this CvEB tree of size 2^{20}, the peak performance drops to just over 20 million operations per second on 30 threads. The effect also exists on Intel, but a shift in the peak

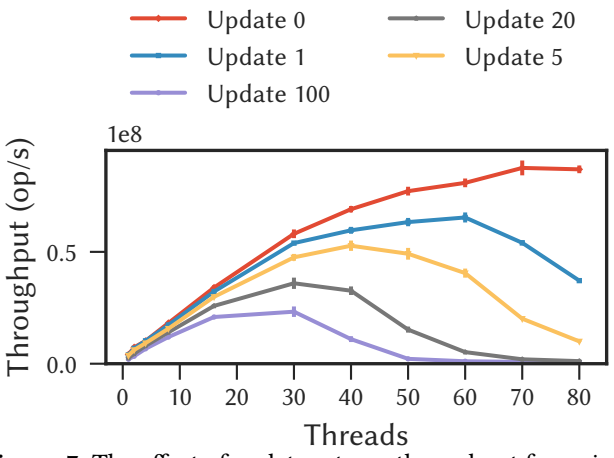


Figure 7. The effect of update rate on throughput for a size 2^{20} CvEB tree on the Power8.

number of threads is not observed as only thread counts up to 4 are tested with maximum throughput occurring always on 4 threads. A similar effect appears to exist when the size of the data structure is changed, however, this is expected as it directly relates to the running time of the data structure operations.

Structure size	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
Optimal thread count	80	80	80	80	60	40	40	20	20	20

Table 1. Ideal thread counts among {20, 40, 60, 80} for various sizes on Power8 with 5% updates

To get a better understanding of how the structure size impacts the performance, we also tested various structure sizes and measured the throughput obtained at thread counts {20, 40, 60, 80} on Power8 when running Synchrobench with 5% updates. We observe that higher thread counts cause performance to drop at lower sizes as depicted in Table 1, most likely due to higher contention that increases the frequency of aborts.

Vacation. The `vacation` application is used to evaluate the CvEB and DCvEB trees in a more realistic environment where multiple data structure operations are contained within a single transaction as the benchmark encapsulates operations within software transactions. Vacation is a program from the STAMP benchmark suite [31] that simulates a travel reservation system. It uses data structures to implement key-value stores for various travel related items and bookings. A single coarse-grained transaction is used to encapsulate all modifications to multiple data structures and maintain their integrity. The amount of contention can be varied by changing the parameters passed to `vacation`. The static CvEB tree was first tested with the low contention (with recommended parameters `-n2-q90-u98-r1048576-t4194304`) and high contention (with recommended parameters `-n4-q60-u90-r1048576-t4194304`) workloads [31] and compared to the red black tree implementation of `vacation`.

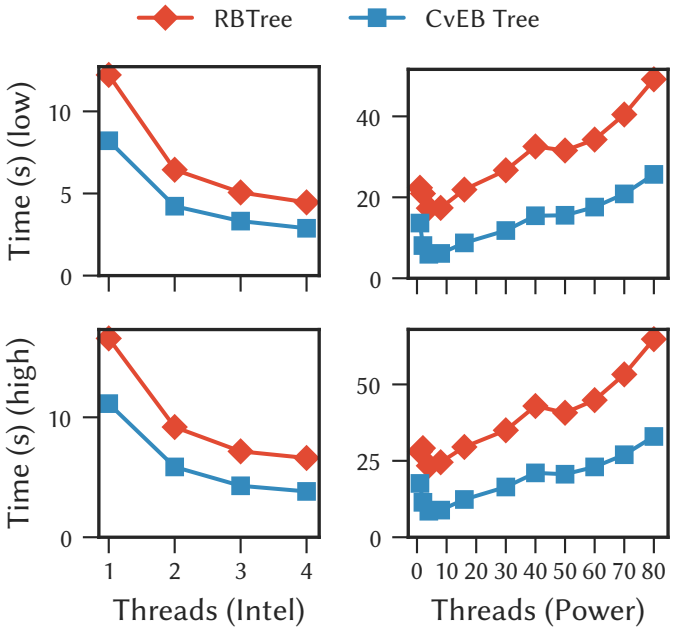


Figure 8. Time taken for `vacation` benchmark

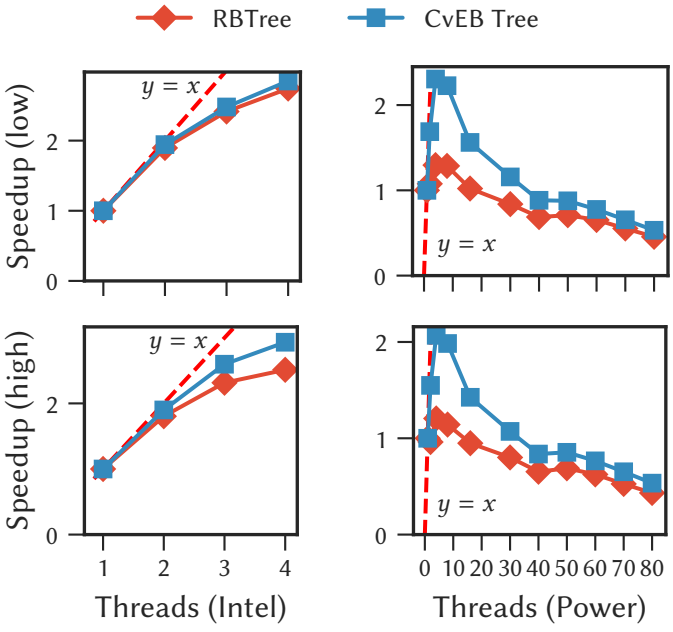


Figure 9. Speedup relative to sequential for `vacation`

On the Intel machine, the CvEB tree has a clear performance advantage over the red black tree as shown in Figure 8. In both workloads the CvEB tree completes the benchmark in less time than the red black tree for any thread count.

Figure 9 depicts the results in terms of the speedup over sequential execution. The CvEB tree shows fairly good scalability on up to 8 threads, after which the performance never improves on any of the structures. This may be a limitation of the `vacation` benchmark, but shows that in more complicated programs, it is still not trivial to gain performance from extra threads even with HTM. We see that somewhere between 30 and 40 threads the CvEB tree performance drops below that of a sequential execution. Note that the red black tree would reach this point at just 16 threads.

Dynamic growth space-constrained data structure. The DCvEB expands our static space-constrained data structure by allowing the range to be doubled as required during runtime. This allows the tree to start with a small range and only expand when an element outside the range needs to be inserted. The performance of the dynamic version is compared against the static version using Synchrobench. In `vacation`, the low growth workload (`-n4-q800-u80-r2097152-t4194304`) and high growth workload (`-n4-q6400-u80-r262144-t4194304`) were designed so that the tree would definitely resize during execution of the benchmarks. The low growth workload issues queries from a range that is 8 times larger than the oroginal range of 2^{21} (for a size of 2^20), so there should be 3 doubling events. Similarly, the high growth workload issues queries from a range 64 times larger than the original range of 2^{18}, so there should be 6 doubling events. The low growth workload causes 3 resizes to be made at some point during the execution and the high growth workload causes 6 resizes.

Figure 10 presents the results of the growth workloads on the Intel machine. There is an obvious difference between the static and dynamic versions of the CvEB tree, with the dynamic version being consistently one second slower in the low growth case. The difference in the high growth workload is not as pronounced but there still appears to be a fairly consistent time difference between the static and dynamic CvEB trees. In both cases, the dynamic tree still outperforms the red black tree. The results are similar for the Power8 system, depicted in Figure 10. In this case the difference in time taken between the static and dynamic trees appears to grow with increasing thread count. The results on both systems are consistent and show that the DCvEB tree outperforms the red black tree at all thread counts. Finally, the throughput as measured on both the Intel and Power8 machines is presented in Figure 11. While no differences are visible on Intel, the Power8 machine does show a lower throughput for thread counts between 30 and 80.

Instrumentations. The Intel machine features 8 general purpose counters per core that can be set to monitor any event. This allows all of the required transactional events to be counted at once. The Power8 features only 4 counters per core that can be freely set while 5 events need to be monitored. It is possible for `perf` to multiplex events as it monitors a process and then extrapolate to the estimated total number at the end. To avoid this, we instead ran two different groups of events to get the actual counts from the performance monitoring unit (PMU). We instrumented the

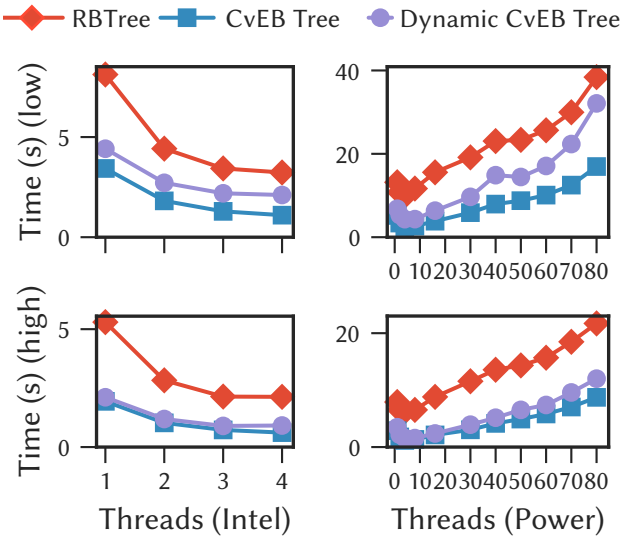


Figure 10. Performance on growth workloads for `vacation` benchmark

HTM macros in Synchrobench to keep track of the number of aborts and their abort reasons for comparison with the counters obtained with `perf`.

Figure 12 shows the total abort counts of `perf` against the instrumented HTM counts on Intel (left) and IBM (right). We see that both `perf` counts and the instrumented counts give the same value on the Intel machine but not on Power8. We look at the transactional performance monitoring unit (PMU) events on Intel [28] in more details and observed `TX_MEM.ABORT_CONFLICT` and `TX_MEM.ABORT_CAPACITY` events seem to overcount, which may be because a transactional abort is signaled more than once before it actually aborts.

In contrast, the counters on the Power8 machine are not as accurate. There is not a single counter that provides a total abort count on the Power8 so we calculated it by adding all the other abort counts together. Figure 12(right) shows that the addition of the capacity and conflict aborts undercounts the total number of aborts, which may be because of other uncounted abort conditions. We also compared the total number of aborts to the difference in the number of transactions started and the number that completed successfully. We observed surprising results (like `PM_TM_END > PM_TM_BEGIN`) that could be due to the highly aggressive out-of-order execution performed by the Power8 causing events to be generated from speculative executions. Colleagues noted that these discrepancies stop with thread pinning. These slight anomalies do not alter our earlier conclusions.

5 Complexities

In this section, we analyse the space and time complexites of the static and dynamic variants of the CvEB tree.

[1] http://www.ibm.com/support/knowledgecenter/SSFK5S_2.2.0/com.ibm.cluster.pedev.v2r2.pedev100.doc/bl7ug_power8metrics.htm

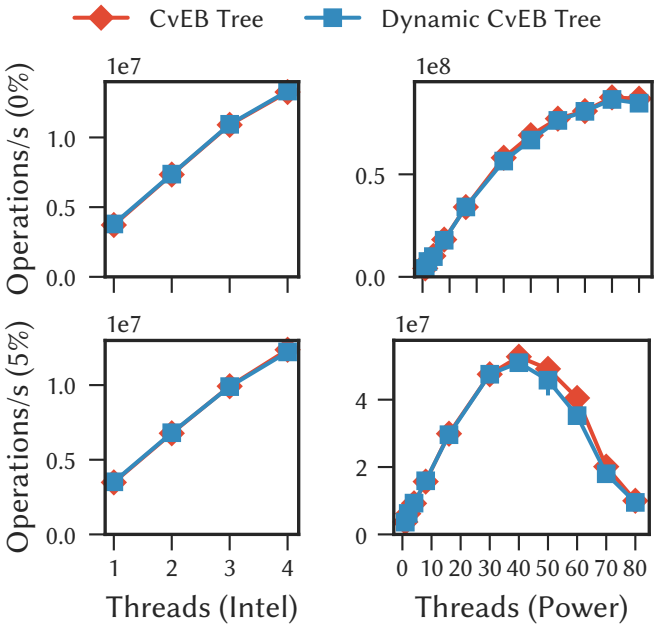


Figure 11. Static CvEB and dynamic DCvEB tree throughput in Synchrobench

The tree features operations insert, delete, contains, successor and predecessor whose complexity is $O(\log\log M)$ (cf. Theorem 3.1) and operations minimum and maximum whose complexity is $O(1)$. The operation of the hardware transactional implementation of the concurrent vEB executes a constant number of hardware transactions before executing in software, where it acquires a lock, hence the complexity in accesses to data items is still $O(\log\log M)$. Consider the number of stalls induced by unsuccessful lock acquirements. This number is upper-bounded by the interval contention $\bar{c}$ [3], hence the amortized complexity in terms of stalls and accesses is $O(\log\log M + \bar{c})$. Given that the interval contention is asymptotically equivalent to the point contention $\dot{c}$ in an amortized context [18], we obtain $O(\log\log M + \dot{c})$.

Now for the dynamic case, we have to measure the cost of resizing DCvEB as this represents an additional cost for the operation that resizes. As the universe M_t is changing with time t, let us consider M_x as the largest universe M_t for all time t during the execution. Assuming that the cost of resizing is negligible compared to the ratio of resizing operations over non-resizing operations, then one can bound the amortized complexity of the dynamic concurrent vEB to $O(\log\log M_x + \dot{c} + \varepsilon)$.

Compared to the simple array, our concurrent vEB will use more space when completely full but this will still be $O(M)$. However, it is possible for this structure to use less space when fewer elements are inserted by not allocating space for unnecessary branches of the tree. Each element requires one entry in a size L array at each of k levels, giving an upper bound of $O(kLn) = O(nL\log_L M)$ space for n elements.

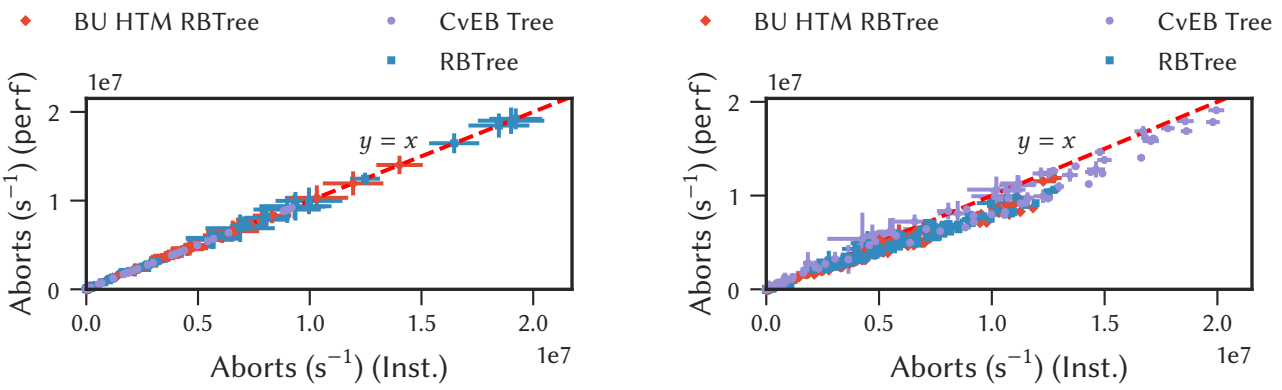


Figure 12. Differences between counters from instrumentation and reported through `perf` on Intel (left) and IBM (right).

6 Related work

In this section, we discuss the related work about hardware transactions and concurrent data structures.

Hardware transactions. There are two ways of using hardware transactions, through hybrid transactional memory [9] that uses HTM to boost the performance of Software Transactional Memory (STM) when possible or through transactional lock elision [36] that speculatively executes a critical section in a fast path hardware transaction before potentially reverting to a software fallback path. HTM have already proved useful for database applications, however, the results usually show scaling performance up to 4 threads or rely on simulations [30], because it is hard to scale to multiple NUMA nodes [4, 32].

As mentioned earlier, various ideas were proposed to avoid performance penalties, including lazy subscription [8], read-write lock elision [16], amalgamated lock elision [1] or requiring at most one software thread to hold the lock [12]. Some, like state teleportation [6] were even tailored to search structures. Our goal is to tune the structure to avoid the fallback path rather than reducing the synchronisation overheads.

Concurrent data structures. Transactional memory, even in its software form, has been tested [21, 22] and optimized for various data structures [15]. A first approach is to limit the transaction size [7, 14, 41]. A skip list was implemented using small software transactions to reach the performance of a lock-free skip list implementation [14]. The speculation-friendly binary search tree [7] decouples the updates affecting the implemented abstraction (e.g., key-value store, set, collection) from the updates that affect the underlying data structures (i.e., the tree itself). The partition of transaction into a read-mostly planning and a write-mostly completion increases performance by restricting the transactional part to the completion [41]. This approach does not usually reduce the step complexity of structure accesses. A second approach is to relax the transactional model like elaborate forms of nesting [5], transactional boosting [24, 26] or elastic transactions [15]. These solutions require software transactions and cannot easily be implemented with hardware transactions whose syntax follows a basic balanced open-close block. A third approach is to adjust the synchronisation techniques [25, 38]. A balance tree [25] does not use any locks during traversal and defers physical modifications and lock acquisitions to a transaction commit phase. Synchronizing red black trees bottom-up was shown more efficient than synchronizing them top-down [38], which motivated our choice of the BU HTM Tree as the baseline for our experiments. A fourth approach is to tune the contention manager or the level of contention [13, 20] rather than tuning the structure to avoid reverting to the software fallback.

Finally, data structures similar to our CvEB have already been applied to database index. A concurrent hash trie with a constant time snapshot operations was implemented in Scala [35]. The adapting radix trie [29] applies to string and offers operations that execute in $O(k)$ where k is the maximum length of all keys in the set. They were not designed for hardware transactions.

7 Conclusion

This paper demonstrates the advantage of adapting algorithmic designs to HTM rather than adapting HTM designs to algorithms. We illustrated this concept with a concurrent space-constrained tree, called CvEB, on two architectures supporting HTM and showed via the Synchrobench benchmark suite and the STAMP `vacation` application, that it outperforms the state-of-the-art HTM-based tree data structures. In order to make the space-constrained structure dynamic, we implemented a resize operation that induces a slight overhead that does not annihilate the performance gain. By shifting the longstanding research focus of optimizing hardware transactions for structures into optimizing structures for hardware transactions, this work opens up unexplored ways of optimizing programs for HTM.

Acknowledgements

This research was supported under Australian Research Council's Discovery Projects funding scheme (project number 160104801) entitled "Data Structures for Multi-Core". Vincent Gramoli is the recipient of the Australian Research Council Discovery International Award.

References

[1] Yehuda Afek, Alexander Matveev, Oscar R. Moll, and Nir Shavit. Amalgamated lock-elision. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 309–324, 2015.

[2] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3), June 2007.

[3] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, July 2003.

[4] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, 2016.

[5] Keith Chapman, Antony L. Hosking, and J. Eliot B. Moss. Hybrid STM/HTM for nested transactions on OpenJDK. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 660–676, 2016.

[6] Nachshon Cohen, Maurice Herlihy, Erez Petrank, and Elias Wald. Poster: State teleportation via hardware transactional memory. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 437–438, 2017.

[7] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2012, pages 161–170, 2012.

[8] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 39–52, 2011.

[9] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, 2006.

[10] Dave Dice, Timothy L. Harris, Alex Kogan, Yossi Lev, and Mark Moir. Pitfalls of lazy subscription. In *6th Workshop on the Theory of Transactional Memory (WTTM)*, 2014.

[11] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *3rd ACM SIGPLAN Workshop on Transactional Computing (Transact)*, 2008.

[12] Dave Dice, Alex Kogan, and Yossi Lev. Refined transactional lock elision. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2016, pages 19:1–19:12, 2016.

[13] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. ProteusTM: Abstraction meets performance in transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 757–771, 2016.

[14] Aleksandar Dragojević and Tim Harris. STM in the small: Trading generality for performance in software transactional memory. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 1–14, 2012.

[15] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. *Journal of Parallel and Distributed Computing (JPDC)*, 100(0):103–127, Feb 2017.

[16] Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. Hardware read-write lock elision. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, page 34, 2016.

[17] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424 – 436, 1993.

[18] Joel Gibson and Vincent Gramoli. Why non-blocking operations should be selfish. In *Proc. 29th International Symposium on Distributed Computing DISC'15*, volume 9363 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2015.

[19] Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, 2015.

[20] Vincent Gramoli, Rachid Guerraoui, and Anne-Marie Kermarrec. Profiling transactional applications. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*, pages 278–292, 2015.

[21] Derin Harmanci, Pascal Felber, Vincent Gramoli, and Christof Fetzer. TMunit: Testing software transactional memories. In *4th ACM SIGPLAN Workshop on Transactional Computing (Transact'09)*, feb 2009.

[22] Derin Harmanci, Vincent Gramoli, Pascal Felber, and Christof Fetzer. Extensible transactional memory testbed. *Journal of Parallel and Distributed Computing - Special Issue on Transactional Memory (JPDC)*, 70(10):1053–1067, 2010.

[23] William Hasenplaugh, Andrew Nguyen, and Nir Shavit. Quantifying the capacity limitations of hardware transactional memory. In *7th Workshop on the Theory of Transactional Memory (WTTM)*, 2015.

[24] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic transactional boosting. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 387–388, 2014.

[25] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Transactional interference-less balanced tree. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 325–340, 2015.

[26] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, 2008.

[27] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[28] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3B. June 2016.

[29] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE'13)*, pages 38–49, 2013.

[30] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In *Proc. of IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 580–591, 2014.

[31] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing.

In *IISWC*, pages 35–46. IEEE, 2008.

[32] Mohamed Mohamedin, Roberto Palmieri, Sebastiano Peluso, and Binoy Ravindran. On designing numa-aware concurrency control for scalable transactional memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 45:1–45:2, 2016.

[33] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 144–157, New York, NY, USA, 2015. ACM.

[34] Rotem Oshman and Nir Shavit. The skiptrie: low-depth concurrent search without rebalancing. In *Proc. of the ACM Symposium on Principles of Distributed Computing, (PODC'13)*, pages 23–32, 2013.

[35] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPOPP'12)*, pages 151–160, 2012.

[36] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, 2001.

[37] Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *Proceedings of the Fourth Annual European Symposium on Algorithms*, ESA '96, pages 121–137, 1996.

[38] Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Performance analysis of concurrent red-black trees on HTM platforms. In *10th ACM SIGPLAN Workshop on Transactional Computing (Transact)*, 2015.

[39] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, FOCS '75, pages 75–84, 1975.

[40] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.

[41] Lingxiang Xiang and Michael L Scott. Software partitioning of hardware transactions. In *ACM SIGPLAN Notices*, volume 50, pages 76–86. ACM, 2015.

ISBN : 978-1-74210-428-7

School of Information Technologies
Faculty of Engineering & Information Technologies
Level 2, SIT Building, J12
The University of Sydney
NSW 2006 Australia

T +61 2 9351 3423
F +61 2 9351 3838
E sit.information@sydney.edu.au
sydney.edu.au/it

ABN 15 211 513 464
CRICOS 00026A