# Improving HTM Scaling with Consistency-Oblivious Programming

Hillel Avni
Tel-Aviv University
hillel.avni@gmail.com

Bradley C. Kuszmaul
MIT
bradley@mit.edu

## Abstract

We implemented two data structures in a consistency-oblivious programming (COP) style: a red black tree and a dynamic cache-oblivious B-tree. Unlike a naive transactional style, in which an operation such as an insertion is enclosed in a hardware transaction, in a COP-style there are two phases: an oblivious phase that runs with no transactions or locking, and an atomic phase that simply verifies that the oblivious phase operated correctly. If the verification fails, we retry, and then fallback to performing the transaction while holding a lock. We used the Intel hardware transactional memory instructions to implement the atomic phase. We found that the COP approach provides a performance improvement of more than a factor of five over a naive HTM transaction for high thread-count read-only workloads, and a factor of four for high thread-count read/write workloads. To achieve good performance when using HTM, one must distinguish between failures that are likely to succeed upon retry (for example, a conflict on the lock) from failures that indicate real conflicts (such as two threads both modifying the data structure).

***Keywords*** HTM, Transactional-Memory, Data-Structures, COP

## 1. Introduction and Related Work

A transactional-memory (TM) transaction maintains read and write sets, either by software (in software transactional memory, STM) or hardware (in hardware transactional memory, HTM). At commit time, the TM infrastructure must verify the read set is a snapshot, and update the write set atomically with respect to that snapshot [6, 7, 12]. Together the read set and write set, can be referred to as the *footprint* of a transaction. In HTM, the footprint must typically fit in hardware cache

Several relaxed consistency approaches [1, 8], exclude non-relevant addresses from the read set during the transaction. In STM, these approaches can reduce the transaction footprint, and as a result, eliminate a portion of the conflicts and the false aborts. However, every shared memory access remains instrumented. Current HTM realizations such as the Intel Haswell and the IBM Power HTM systems maintain their read set in a way which does not allow software to access or modify it in any way, which makes these approaches infeasible.

Consistency oblivious programming (COP) [2] provides one way to write TM algorithms that use a smaller footprint. In our version of COP adapted for HTM, a COP-transaction is divided into two parts: a read-only *prefix*, and an updating *suffix*. The prefix runs without any synchronization and generates a possibly inconsistent output. The suffix starts a hardware transaction which has two objectives: verify that the prefix produced an acceptable result, and perform any writes needed by the COP-transaction.

Our version of COP is similar to STM-based COP, found for example in running the read-only prefix outside transaction in [2, 3]. One of the big advantages of STM-based COP is that it reduces the overhead of STM instrumentation. This benefit is irrelevant to HTM, which does not have any instrumentation overhead. However, reducing the footprint of a transaction can make the difference between whether or not an application must execute a fallback code, such as obtaining a global lock. Furthermore, reducing the footprint of a transaction can reduce the probability that two separate transactions conflict. For example, sometimes the prefix can run, and produce an acceptable result, even though the part of the data structure examined by the prefix was not technically consistent.

COP addresses two important limitations of HTM: the limited capacity for transactional accesses and the inability to release items from its read and write sets.

COP reduces the number of memory accesses in the transaction, and thus make it more likely to fit within the limitations imposed by hardware. Since the footprint of an HTM transaction must fit in cache, and caches typically provided limited associativity, programmers may be surprised that some transactions with small footprints cannot commit. Our experiments show that using COP, we can compose many operations into a single COP transaction without violating the resources of the hardware, while in a naive HTM version can handle only a much smaller number of operations.

Since the prefix is run outside of any transaction, it cannot abort, although it may cause conflicts with other transactions that write to the locations read by the prefix. Consider, for example, a hundred-core chip, in which 99 threads run read-only transactions, each of which queries several keys in million-node red-black tree. A COP version of this transaction can get away with only holding a small constant footprint. On the other hand, a naive HTM version of that same transaction will hold the root of the tree, and the entire path down the tree. In naive HTM, if the 100th core performs an insert that requires re-balancing the root, the other 99 will abort. In COP, rebalancing the root is unlikely to disrupt a COP-transaction that is still running.

A potential problem in composing multiple COP operation into a single transaction, is that once the first operation started the transactional part, the other prefixes will run in the context of that transaction. In our code, we group all prefixes and then, in a transaction, we run all the suffixes. However, this limits us to perform updates only in the last operation of a transaction, otherwise the transaction will miss its own updates. One open problem is thus how to write composable COP-style programs.

An upcoming POWER HTM implementation [5] plans to support a suspended transactional mode, which will permit combining multiple COP-style updating operations in the same hardware transaction.

The rest of this paper is organized as follows. Section 2 describes how we adapted the COP idea to an HTM context. Section 3 explains our concurrent red-black tree implemented with

COP Template for Function $\mathcal{K}$ on GCC-4.8

```
1  retry-count = MAX-RETRY;
2  KROPOutput ← KROP();
3  status ← _xbegin;
4  if status = _XBEGIN_STARTED then
5      κVerify(κROPOutput) // _xabort (BAD_ROP) if fails;
6      κComplete(κROPOutput) // Perform updates;
7      if locked then
8          _xabort (WAS_LOCKED)
9      _xend;
10 else
11     if is_explicit(status, WAS_LOCKED) then
12         goto line 3; // Reuse ROPOutput.
13     decrement (retry-count);
14     if status ≠ _XABORT_CAPACITY then
15         if retry-count > 0 then
16             if is_explicit(status, BAD_ROP) then
17                 goto line 1; // Retry prefix
18             else
19                 goto line 3; // Reuse ROPOutput.
20             end
21     lock;
22     κ();
23     unlock;
24 end
```

Figure 1: COP Template

COP. Section 4 explains our COP-based cache-oblivious B-tree. Section 5 presents performance measurements and compares the various schemes.

## 2. COP Template

The COP algorithms we developed should work with any HTM block, but for this paper, we implemented them for the Intel Haswell RTM, and used the intrinsics **_xbegin**, **_xend** and **_xabort**, that were introduced in the GCC-4.8. The **_xend** commits a transaction, and **_xabort** terminates it with an abort. The **_xbegin** return an error code. The codes which interest us in the context of COP, are in the following table:

| Code | Meaning |
|---|---|
| _XBEGIN_STARTED | Transaction started. |
| _XABORT_CONFLICT | There was a conflict with a concurrent transaction. |
| _XABORT_CAPACITY | Transaction is too large. |
| _XABORT_EXPLICIT | Software called xabort |
| _XABORT_CODE | The parameter given in xabort. |

### 2.1 Operation Structure

Let $\mathcal{K}$ (kappa) be a function, which is a sequential operation on a data structure. The template for a COP version of $\mathcal{K}$, using the GCC-4.8 intrinsics, is given in Figure 1.

To adapt $\mathcal{K}$ to COP, we extract the longest read-only prefix of it into $\mathcal{K}$ROP() (line 1). $\mathcal{K}$ROP() calculates $\mathcal{K}$ROPOutput, in an unsafe mode, i.e., without any synchronization. Thus $\mathcal{K}$ROPOutput might be inconsistent and wrong, due to conflicts with concurrent operations.

After calculating $\mathcal{K}$ROPOutput, we start a transaction in line 3, and call $\mathcal{K}$Verify($\mathcal{K}$ROPOutput) in line 4. $\mathcal{K}$Verify will call **_xabort** if $\mathcal{K}$ROPOutput is inconsistent. If $\mathcal{K}$ROPOutput is consistent, we will continue the transaction to run $\mathcal{K}$Complete($\mathcal{K}$ROPOutput). $\mathcal{K}$Complete($\mathcal{K}$ROPOutput) will use $\mathcal{K}$ROPOutput and do any updates, considering $\mathcal{K}$ROPOutput is correct.

Before we try to commit in line 9, we check in line 7 that the global lock is free. If its locked, we abort with a specific code. We could sample the lock in the beginning and abort for a conflict in case some thread grabbed the lock, but this could lead to a false fallbacks, as a conflict is considered as a retry, while a lock, as seen in line 12 allows us to reuse the ROP output, and not considered as a retry.

If the transaction failed and we want to retry, we will reach line 15. If the source of abort was capacity overflow, we do not retry the transaction, as it will probably fail again. If it was an explicit abort, i.e., $\mathcal{K}$Verify called **_xabort**, we must rerun $\mathcal{K}$ROP to get a correct $\mathcal{K}$ROPOutput, otherwise, the abort was due to a conflict, so $\mathcal{K}$ROPOutput may well be correct, and the transaction has a chance to commit successfully, thus we can reuse $\mathcal{K}$ROPOutput and retry the HTM transaction. If we have no more retries, we lock and run $\mathcal{K}$ sequential version.

### 2.2 Correctness Proof Method

A correct COP version of $\mathcal{K}$ will be equivalent to locking and running $\mathcal{K}$Complete($\mathcal{K}$ROP()), which in turn, is equivalent to running $\mathcal{K}$ in an HTM transaction. To have a correct COP version of $\mathcal{K}$, we need to demonstrate the following:

**Property 1.** ***Obliviousness:*** *$\mathcal{K}$ROP() completes without faults, regardless of concurrent executions, and will finish in a finite number of steps if runs alone.*

**Obliviousness** is progress related, as if $\mathcal{K}$ROP() will crash or stuck in an infinite loop, no work will be done. The following two properties imply that the COP version of $\mathcal{K}$ is correct.

**Property 2.** ***Verifiability:*** *$\mathcal{K}$ROPOutput has attributes, that can be tested locally, and that imply $\mathcal{K}$ROPOutput is consistent, and $\mathcal{K}$Verify is checking these attributes.*

**Property 3.** ***Separation:*** *$\mathcal{K}$Complete is using $\mathcal{K}$ROPOutput but is not aware of any other data collected by $\mathcal{K}$ROP().*

**Verifiability** imply that the consistency of $\mathcal{K}$ROPOutput can be checked locally, by looking at its attributes. This may require adding to the sequential $\mathcal{K}$ code, without changing its functionality. As the $\mathcal{K}$Verify and $\mathcal{K}$Complete are in the same transaction, we know $\mathcal{K}$ROPOutput stays consistent until commit, and as $\mathcal{K}$Complete runs virtually under a global lock, and according to **Separation**, $\mathcal{K}$Complete accesses only consistent data, we have a linearizable, COP version of $\mathcal{K}$.

The system model here is a global lock, i.e., a code segment that runs in a transaction is semantically protected by a global lock, and have all its necessary barriers inserted by the hardware TM.

Now, if we want to implement a COP version of a function $\phi$, we only need to show $\phi$ROP, $\phi$Verify and $\phi$Complete. For example, if we want to demonstrate a COP implementation of an RB tree Insert function, we will present InsertROP, InsertVerify and InsertComplete. After creating the COP version, we have to show it has the three properties described above.

## 3. COP Red-Black Tree

In Figure 2, we see two concurrent search operations that start a search for the key 26 in an unbalanced RB-Tree. One is a COP operation which is doing this read-only prefix in non transactional context, and the other is a plain TM operation, which is in transactional mode. When both searches reached 27 the tree was balanced and 27 became the root of the tree. Now the COP search, which is not in transactional context continues and reaches the leaf which

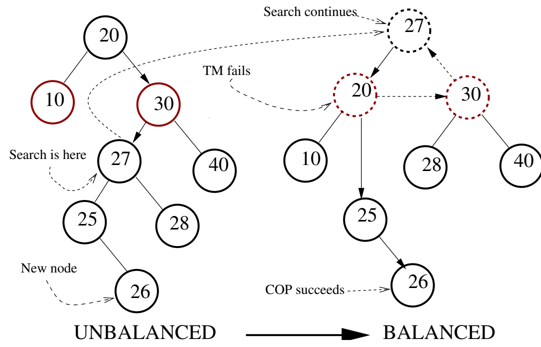



Figure 2: COP and TM search for key 26 in an RB-Tree

holds 26. Plain TM search on the other hand, which is in transactional context from the start, fails right after balancing. The reason is that the search traversed the right pointer of 20 in the beginning of the search, and balancing modified that pointer. In addition, balancing changed the color of 20 from black to red. As the color and the pointer are in the same node, and thus probably in the same cache line, changing the color by itself was enough to fail the TM search. After COP completes the non transactional search, it will resume the transaction to verify it got a valid result. Note that when TM failed, it lost the whole transaction and not only the search for 26 operation. If for example, 26 was a product of a heavy prior operation, that operation is lost as well, while the transaction that used a COP operation continues.

We ported the COP Red-Black Tree with chained leaves from [2] to our HTM COP template. Listing 1 shows the code for inserting into a red black tree, using C notation to make the exposition as close to the real code as practical.

The algorithm for insertion, which was introduced and proved in [2], looks for a key $K$ and returns a node $N$. If K is found, $N$ holds $K$. Otherwise $N$ is a leaf which is either the potential predecessor or successor of $K$. If $N$ is the potential predecessor, $K$ should be inserted in its right pointer, which must be `NULL`, and if $N$ is the potential successor, K should be inserted in its left pointer which must be `NULL`.

The code first performs the read-only prefix with no locking or synchronization (at line 4). We employ a type-preserving node recycler of the node, and keep the nodes within the same tree, so that arbitrary pointers will not lead us to undefined memory, which could crash our code or fool it with locations which look like valid nodes, but are not. Our RB-Tree implementation recycles nodes within a thread, and if a thread accumulates more then a threshold of idle nodes, it uses an epoch based memory reclamation [9] scheme to free them.

The verification performs a straightfoward lookup, as shown in Listing 2.

Returning to Listing 1, the code next waits until the tree is not locked (at line 6). The fallback code acquires a mutex on the tree. As we shall see, to make progress we will require that the lock is not held, so there is no point in trying to start a transaction to operate on the tree until the lock is released.

Next, the code begins a transaction (at line 7). The `_xbegin()` function either returns `_XBEGIN_STARTED`, in which case the code is running in a transaction, or the system attempted the transaction and failed, and status tells us something about why it failed.

In the case that we are running the transaction, we must finish the insertion. Since the read-only prefix ran without any synchronization, it could yield an inconsistent result and we must verify its correctness (at line 10). The verification code, shown in Listing 3,

```
1   void rb_insert(RBT *s, int K, int V) {
2     int retry_count = 0;
3   retry:
4     node_t *place = RBROP(t, K);
5   retry_verify:
6     while (tree_locked) pause();
7     int status = _xbegin();
8     if (status==_XBEGIN_STARTED) {
9        RBVerify(place, K);
10       RbInsertComplete(t, K, V, place);
11       if (tree_locked) _xabort(WAS_LOCKED);
12      _xend();
13    } else {
14      if (is_explicit(status, WAS_LOCKED))
15        // Lock was held. Always try again.
16        goto retry_verify;
17      // Other failures prejudice us.
18      // Allow only RETRY_COUNT retries.
19      if (retry_count++   RETRY_COUNT) {
20        if (is_explicit(status, BAD_ROP))
21          // Must redo the whole prefix
22          goto retry;
23        if (can_retry(status))
24          goto retry_verify;
25      }
26      // Fallback code.
27      lock_tree();
28      place = RBROP(t, K);
29      RBInsertComplete(t, K, V, place);
30      unlock_tree();
31    }
32  }
```

Listing 1: RBInsertComplete function.

```
33  node_t* RBROP(RBT *s, int K) {
34    node_t * p = s- root;
35    node_t *pp = NULL;
36    while (p != NULL) {
37      if (K == p- k) return p;
38      pp = p;
39      p  = (K   p- k) ?  p- l : p- r;
40    }
41    return pp;
42  }
```

Listing 2: RB-Tree COP Lookup (ROP)

checks that the node is the right place to insert a key $K$. To facilitate the verification, we always insert two sentinel nodes with the keys $\infty$ and -$\infty$. The node must exist and be allocated (lines 46–47). If the node has a key matching $K$ then we've got a good result. Otherwise, the previous node must have a smaller key and the next node must have a larger key. Furthermore, if the node has a key larger than $K$ then there must be no left child (line 49), otherwise there must be no right child. If the verification fails, it calls `_xabort()` to explicitly abort the transaction with a code indicating that the verification failed.. If the verification succeeds, we call code to complete the insertion at line 10. Finally we check to see if the tree is locked If it is, then some other code may be modifying the data structure in a way that is inconsistent with our transaction. In this case, we explicitly abort with a code indicating that the lock was held.

Because the tree has the sentinel nodes, there is no need to check predecessor and successor pointers are not `NULL`. When the tree is empty, for example at the first insertion, the verification will fail by following a NULL, and eventually fallback to the lock and skip the verification. This is acceptable, as it will happen once, and once for the other sentinel node, as the predecessor will be `NULL`, and never happen again. On the other hand, it saves conditions in the rb_rop_verify which is called frequently.

```
43  #define BAD_ROP 1
44  inline void RBVerify(node_t* p, int K) {
45    node_t *next;
46    if (!p)                 _xabort(BAD_ROP);
47    if (!p- live)               _xabort(BAD_ROP);
48    if (p- k != K) {
49      if (p- k   K) {
50       if (p- l != NULL)     _xabort(BAD_ROP);
51       if (p- prev- k  = K) _xabort(BAD_ROP);
52      } else {
53       if (p- r != NULL)     _xabort(BAD_ROP);
54       if (p- succ- k  = K) _xabort(BAD_ROP);
55      }
56    }
57  }
```

Listing 3: RB-Tree COP ROP Verify

In the case that the transaction failed, there are four interesting kinds of failures handled in the `else` clause at line 13.

- The transaction could have failed because the lock was held. In this case, at line 14 we always retry the transaction, since when the lock is released, we have every reason to hope that our transaction will succed.
- The transaction could have failed because the read-only-prefix gave a bad answer. In this case, at line 22, we retry a limited number of times, since we know there are actually conflicts occuring from other transactions in the tree.
- The transaction could have failed in some other a way that gives us hope that retrying will help. It turns out that almost all failures have a chance of succeeding on retry, even failures marked as failing because the buffers overflowed (capacity). At line 24 we retry if the status has the retry bit set, if it has the capacity bit set, or if the status is zero incication some other failure (such as a time slice interrupt occured).
- Finally either we have exhausted our retry budget or we believe that retrying won't be helpful for some other reason, and at lines 27–30 we throw in the towel: we lock the tree, redo the prefix, complete the insertion, and unlock the tree.

We don't show the code for completing an insertion or deletion.

### 3.1 Correctness

We assume there is some safe memory reclamation, which ensures a node is not recycled until all tasks which access it terminate [9]. In addition, when we recycle a node we set the left and right pointers to `NULL`, as well as the live field to **false**, so there are no cycles in the garbage nodes. We need to show our COP version of RB tree has the three correctness properties.

**Lemma 1.** *RBROP has the **Obliviousness** property.*

*Proof.* As the HTM transaction and the fallback path, have global-lock semantics, ROP sees only pointers that were part of the tree. Thus, when it reaches a node N, either N is in the tree, and then there is a finite path from N to a leaf or N was in the tree and was removed in a deletion. As a result, after a finite number of solo steps, ROP will reach a leaf or a deleted node.

ROP stops when it sees a `NULL`pointer. All pointers exiting a leaf or a deleted node are `NULL`, thus, ROP will stop after a finite number of solo steps.

As ROP visit nodes that are either deleted or in the tree, they will point to `NULL`or to a valid node, and in both cases ROP will not hit uninitialized pointers or unallocated memory and will not crash. □

When a ROP for tree $T$ and key $K$ completes, it returns a node $N$ and a flag $F$.

**Lemma 2.** *$N$, $F$ and `RBVerify` have the **Verifiability** property.*

*Proof.* If N is live and holds K, we know it is part of T and has the correct key.

If N is live and holds key $K_1$, and N points to successor S which holds key $K_2$, and $K_1 > K > K_2$, we know K is not in the tree, and $K_1$ is the closest key to K from above. This is true, because the successor-predecessor doubly linked list is accessed only in transactions, and thus must be consistent. If N→l is `NULL`, we know we can encapsulate K in a node and connect it as N left son. The case we got the successor is symmetric. □

It is left to prove that the completion is not using values seen during the ROP:

**Lemma 3.** *RBInsertComplete has the **Separation** property.*

*Proof.* The parameters for both RBInsertComplete are the global pointer to the tree, which is constant, and a pointer to the node and flag which are the output of ROP and are verified. As the ROP ran in a separate function than the complete, and did not write any global data, the only information it can pass to the complete function is the parameters. □

The same way, with trivial modifications, we can show the delete has the above properties.

As we proved all COP RB-Tree functions have the **Obliviousness**, **Verifiability** and **Separation** properties, in conclusion, we have shown the following.

**Theorem 1.** *COP RB-Tree is linearizable.*

## 4. Cache-Oblivious B-Tree

We also tested a dynamic *cache-oblivius B-tree (COBT)* [4]. A COBT comprises two parts: a packed memory array (PMA) and an index tree. The PMA holds all of the key-value pairs in a sorted array with some empty slots. By judiciously leaving empty slots in the array, the average cost of an insertion or deletion can be kept small.

The index tree is a uniform binary tree. Rather than providing a binary tree to index every element of the PMA, a COBT indexes *sections* of the PMA. The COBT partitions the PMA into *sections*, typically of size about $\log^2 N$ for an array of size $N$. Thus, the index tree is of size about $N/\log^2 N$.

The index tree is stored in an array. Unlike the usual breadth-first ordering of a tree, in which a node stored at index $i$ has children at indexes $2i+1$ and $2i+2$, the COBT employs a Van Emde Boas order in which the index calculations are a little more complex: the layout recursively lays out the top half of the tree in the array (that is of size approximately $\sqrt{N}$), and then recursively lays out each of $\sqrt{N}$ subtrees in the bottom of half of the tree, one after another. We used code from [11].

Figure 3 shows an example COBT containing 18 elements in an array of size 32. At the bottom of the figure is a PMA containing values, which are the letters 'A', 'C', 'F', 'G', etc. In this example, the sections are of size 2, but in a real implementation the sections are typically larger. Shown in the middle of the figure is the index tree. Each node of the index tree is shown with a dotted line that shows how the node partitions the array into left and right. The node contains the largest element in the left of the partition, so that for example the root node contains an 'N' indicating that the left half of the array contains elements that are all less than or equal to 'N'. The right child of the root contains 'U', indicating that the left $3/4$ths of the array contains values less than or equal to 'U'.

To understand the Van Emde Boas layout, notice that the top half of the tree contains 'N', 'H', and 'U', and there are four subtrees rooted at 'F', 'L', 'R', and 'W' respectively. First the top tree is laid out ('N', 'H', 'U'), then each subtree is laid out starting with 'F', 'C', and 'G'.

The advantage of a COBT is that it can perform insertions and deletions in amortized time $O(\log_B N)$ without knowing the cache line size $B$. Thus this data structure is optimal and cache oblivious. Although the average cost is low, our implementation has a worst-case insertion cost of $O(n)$. It turns out that one can build a COBT in which the worse-case cost is also $O(\log_B N)$, but we haven't implemented it.

To search for a key-value pair in a COBT, first traverse the index tree to find the section in which the pair may reside, then perform a linear search through the section to find the key.

To insert a key-value pair into a COBT, first find the location where the pair belongs as though for a search. If there is already a matching key, then replace the value. Otherwise slide pairs slightly to the left or right, if needed, to make a space for the new pair, and store the pair.

To convert to the COP style, we add a global lock, which is used for the fallback code: If a COP transaction fails, grab the lock and perform the operation.

The (hopefully) common case, when a COP transaction succeeds operates as follows.

The read-only prefix identifies the key's location (without holding the lock). The memory allocation is simpler than for the red-black tree, since the data structure comprises two arrays. The only time that a pointer changes would be if the array were reallocated. We allocate big enough arrays that the arrays are never reallocated, and rely on the operating system's lazy memory allocation scheme to avoid using more physical memory than we need. This works fine on a 64-bit machine, where we can afford to waste part of the virtual address space.

The verification step has two cases:

1. For a successful search (the key was found), we check that the key we want is in the location returned.
2. For a search of an object that is not present, we scan to the left and right of the identified location to find the first nonempty slot, and verify that the search key is greater than and less than the respective nonempty slot keys. The data structure maintains the invariant that each section is nonempty, so the scan to the left and to the right is guaranteed to look at only $O(\log^2 N)$ slots, and require only $O((\log^2 N)/B)$ cache misses.

Just as for the red-black tree, we must take care about to perform retries. We check that the tree is not locked before attempting a transaction (which will verify that the lock is not held). If the transaction aborts because the lock was held, we always retry. Otherwise we retry a few times (each time waiting for the lock to free before retrying). If the verification fails, we must redo the prefix. To execute multiple query operations within a single transaction, one accumulates all the verification steps and performs them at the end.

For lack of space, we do not include here a proof that COBT is linearizable.

Debugging transactions is painful. We'd like to get some information out of the transaction besides the abort code. The abort code seems useless except for the case of explicit aborts. The other codes, such as conflict and capacity all seem to call for retrying the transaction.the same. For example, conflict misses sometimes work on retry, and sometimes the retry status is zero, which we suspect is caused by a TLB miss or an interrupt.

```
58  volatile int dummy;
59  int test (volatile char *A, int stride) {
60    for (int txsize=1; 1; txsize++) {
61       for (int trial=0; trial 20; trial++) {
62         int sum=0;
63         for (int i=0; i txsize; i++) {
64           sum+=A[i*stride];
65         if (_xbegin() == _XBEGIN_STARTED) {
66           A[0]++;
67           for (int i=0; i txsize; i++) {
68             sum+=A[i*stride];
69           _xend();
70           dummy=sum;
71           goto next_txsize;
72         }
73       }
74       // 20 trials failed.
75       // Return the last txsize that worked.
76       return txsize-1;
77    next_txsize:;
78    }
79  }
```

Listing 4: Code for determining the capacity of a transaction.

## 5. Evaluation

We use a Core i7-4770 3.4 GHz Haswell processor, running Linux 3.9.1-64-net1 x86_64. This processor has 4 cores, each with 2 hyperthreads, and hyperthreads enabled. Each core has a private 32KB 8-way associative level-1 data cache and a 256KB 8-way level-2 data cache. The chip further includes a shared 8MB level-3 cache. The cache lines are all 64-bytes.

The TLB may also affect the success of a transaction. In Haswell, the level 1 data TLB has 64 entries, 4-way associative and the level 2 unified data/instruction TLB has 1024 entries, 8-way associative [10].

All benchmarks code was written in C and compiled with GCC-4.8. We use HTM intrinsics that were introduced in that compiler version.

Before we evaluate our algorithms,we want to better understand the behavior of the HTM in practice. We initiate a test that reads cache lines from a practically infinite array. We read the array with power-of-two strides, i.e., we read a byte, skip a number of bytes, read the next one, and so forth.

We found that if a transaction is read-only and the data is already in level 3 cache, the system can accommodate very large transactions. However, if there is even one instance of an address that is written and then read, the capacity drops to level-1 cache size, and is bounded by level-1 associativity. Since we expect most transactions to perform a write, the meaningful transaction size is whatever fits in level-1 cache.

Listing 4 shows the code for testing transaction size. One problem we faced on these experiments was to make sure the compiler does not optimize our loop away. We declared `dummy` and `A` to be `volatile` to convince the compiler that our needs to run. In each transaction we perform one read-after-write as in line 66.

It turns out that if you write to a different location, you get strange artifacts. For example if you write to $A[128]$, then for strides of 128 and less, the size is limited by level 1 cache, but strides of 256 and larger do not read the written value, and the limit appears to be from level 2 or level 3 cache. The blue line in Figure 4 shows what happens in this case, as the capacity drops from 32KB as expected until the stride equals $2^7$, and then for a stride of $2^8$, the capacity jumps up again.

Figure 4 shows the size of the largest observed transaction with a given stride. For 64-byte stride (that is one cache line), we manage to access about 512 different cache lines in a successful transaction.

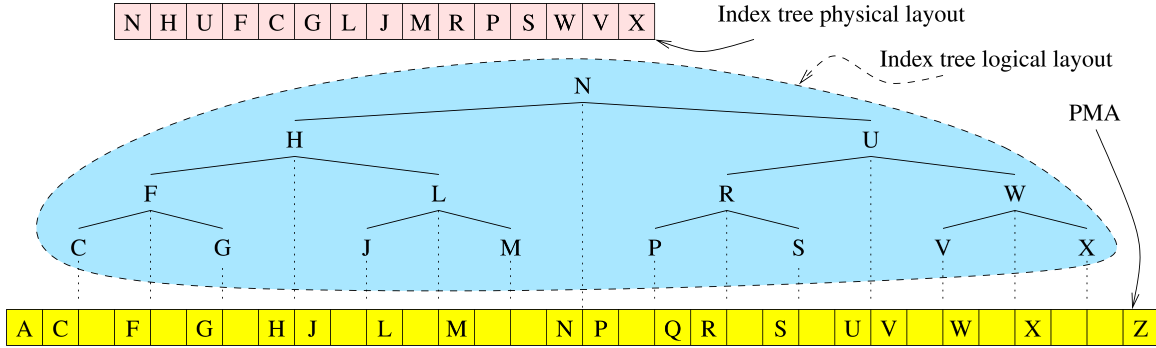



Figure 3: An example dynamic cache-oblivious B-tree. The bottom array is a PMA containing values. The middle tree is an index structure on the array. Each node of the tree contains the largest value to the left of the node. The top array shows the same index tree stored using a Van Emde Boas physical layout.

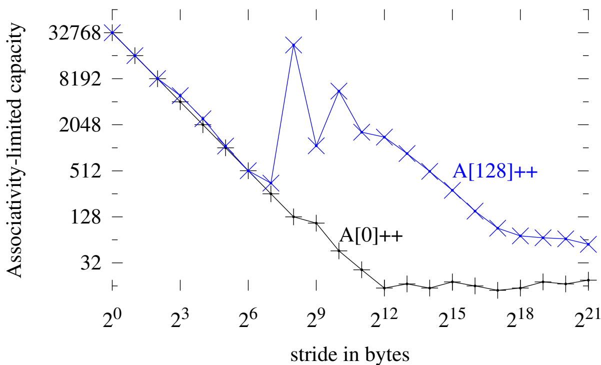



Figure 4: Associativity limits on Haswell HTM capacity obtained by measuring a read-only transaction that accesses a sequence of memory locations with a particular stride. The horizontal axis is the stride of the access. The vertical axis is the number size of the largest transaction that succeeds. The black line shows what happens when we write to location $A[0]$ at the beginning of the transaction. The blue line shows what happens if we write to $A[128]$ at the beginning of the transaction.

This is what we expected, since level-1 data cache has 512 cache lines. Since level 1 is 8-way set associative, we expect to get at least 8 accesses, for any stride size. When we double the stride, we expect the number of accesses in a successful transaction to be the maximum of $\text{CacheSize}/(\text{CacheLine} * \text{Stride})$ and 8, which is what Figure 4 shows.

To generate the data in Figure 4, we run the a given transaction several times. Each time, before running the transaction we perform all the reads (at lines 63–64) so that the cache will start out holding as much of the relevant data as we can fit. If the `_xbegin` returns success, then we try a bigger transaction. Otherwise we repeat and after 20 failures we consider ourselves to have found the largest transaction that we can run with that stride.

### 5.1 RB-Tree Performance

COP reduces the number of capacity and conflict aborts in HTM. To demonstrate these facts better on an RB-Tree, we needed to create more complex tests, because the RB-Tree operations have naturally low contention and, at least for small trees, simple transactions usually succeed. Although these tests are synthetic, they represent important scenarios.

**Capacity:** We combine multiple operations, to challenge the capacity of the HTM buffer. In the COP template in Figure 1, we see that if a transaction gets a capacity abort. it will take a lock and not retry. This means that the number of capacity aborts is bound by the number of successful transactions.

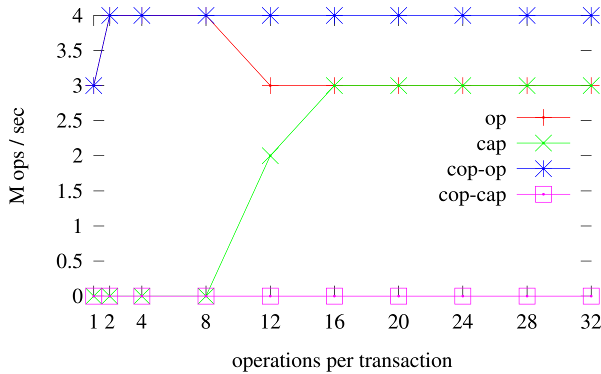
On a single thread, if a transaction will get the capacity abort early, it will take the global lock and lose some performance, however, in a parallel execution, the global lock will eliminate scalability of the performance. To make the results more readable, we count successful operations and not successful transactions, by multiplying the number of successful transactions by the number of operations per transaction. If we got a capacity abort, we also count it as the number of operations in that transaction, as it would mean this number of operations now will run under a global lock.

In Figure 5 we see a read-only workload, where the x axis is the number of operations per transaction. We can see the COP version manages to maintain almost the same bandwidth of operations, up to 32 operations per transaction and much more, while the naive HTM version hits capacity limit quickly. Note conflicts can not be a factor in this workload as it is read only. Also, if conflicts were the reason for locking, we would not see the capacity aborts line at the operations count line. Another important insight is that for single operation transaction on a small tree, capacity aborts seldom occur.

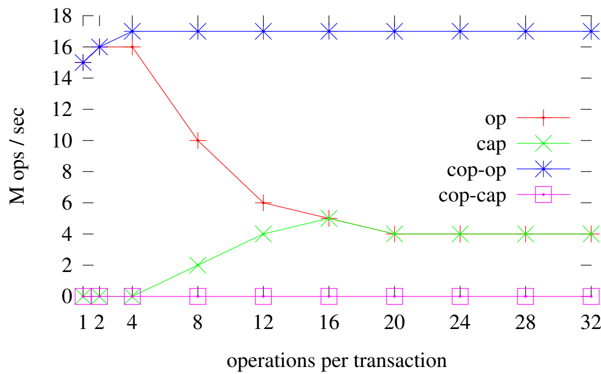
In Figure 5a we run one thread, in Figure 5b we run four and in Figure 5c eight, and as expected, the more threads we use the higher the advantage of COP. The simple reason is that capacity aborts force naive HTM to fallback to global locking, which makes it unscalable, while virtually all COP operations complete successfully within an HTM transaction.

Another insight is that on one and four threads, naive HTM is scalable up to 16 operations per transaction, while on eight it is scalable only to 8. The reason is hyperthreading, where each thread from the eight, is sharing the cache with another thread on the same core, so available capacity for HTM is cut to half.

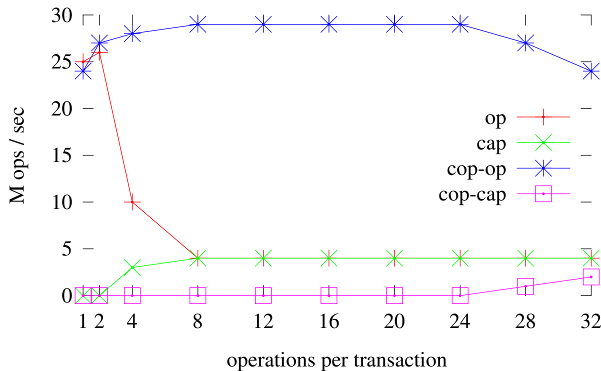
**Conflicts:** An RB-Tree has low contention, so to demonstrate how COP reduces conflicts we devised a variation of the insert that writes arbitrary data to the value field in the root node, as well as inserting the key in the tree. The value is in the same cache line with the pointers and the key, so any concurrent transaction which will traverse the root will have to abort. In Figure 6 we see a workload with 20% such inserts. In a single and two threads, we can see




(a) One thread.



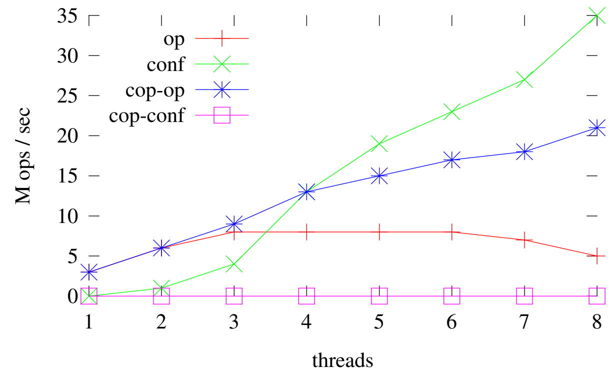

(b) Four threads.




(c) Eight threads.

Figure 5: RB-Tree Benchmark, with various number of read-only operations per transaction. We compare a simple HTM with COP, and count total number of operations and not transactions (op for simple HTM and cop-op for COP operations). We also show number of capacity aborts (cap for simple HTM and cop-cap for COP operations), to demonstrate that they are the reason of COP better performance. We present graphs for 1, 4 and 8 threads. The tree is initially populated with 100K nodes.

COP has the performance of naive HTM, but then naive HTM stops scaling while COP version keeps climbing. The reason is conflict aborts, which are accumulating from 3 threads for naive HTM while COP does not suffer from conflicts at all. All the transactions are of a single operation, so capacity aborts are insignificant as seen in Figure 5.




(a) Various threads, single operation per transaction, 20% insert operations that change value in root, 20% deletes, 60% lookups. Counting operations and conflict aborts.

Figure 6: An RB-Tree Benchmark, comparing COP and simple HTM. Counting operation (op for simple HTM and cop-op for COP operations), and conflict aborts (conf, cop-conf). We do not show capacity aborts, as Figure 5 shows capacity aborts number for a single operation transactions are negligible. We have a lot of conflicts in the simple HTM, as each updating transaction is also writing a value in the root of the tree, which does not distract COP. Each HTM transaction is retried up to 20 times before locking. The tree is initially populated with 100K nodes.

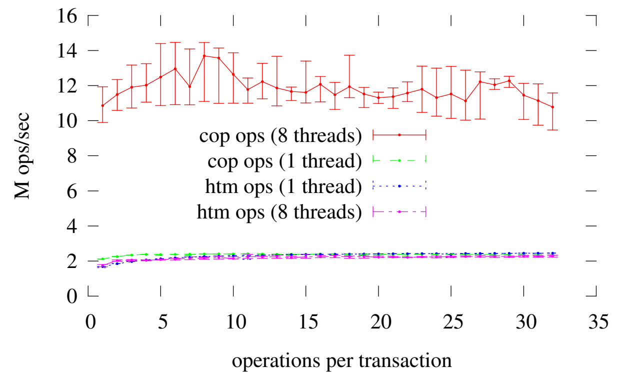



Figure 7: PMA performance for read-only operations on a PMA, for COP and naive HTM for 1 thread and 8 threads. The horizontal axis is the number of searches within a single transaction. The vertical axis is the performance (more is better), measured in number of successful searches per second. Each configuration was run ten times. The error bars show the slowest, the fastest, and the average runtime (through which the curve passes).

### 5.2 PMA Performance

Figure 7 shows the read-only performance of the PMA running both with naive HTM and with COP, for 1 thread and 8 threads. The error bars are negligible for all the runs except the 8-thread COP version, which shows more than 30% variation in runtime. The figure shows the number of successful searches per second, whether the searches were done with HTM or with a lock. The naive HTM code is running with virtually every successful search being performed by the fallback code holding the global lock. Thta is, there are essentially no successful HTM searches in the 8-thread runs. We believe that this poor performance is a result of cache associativity: the array is always a power of two in size, and a binary search on the array hits the same cache line repeatedly. A

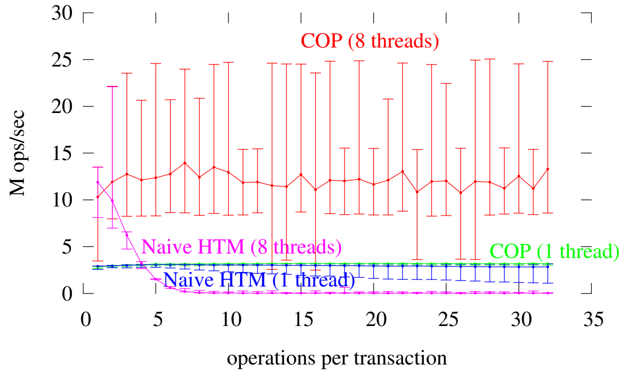



Figure 8: Cache Oblivious B-Tree performance for read-only workloads on a tree containing 100,000 values. The horizontal axis is the number of searches within a single transaction. The vertical axis is performance, measured in number of successful searches per second. The error bars show the slowest, the fastest, and the average.

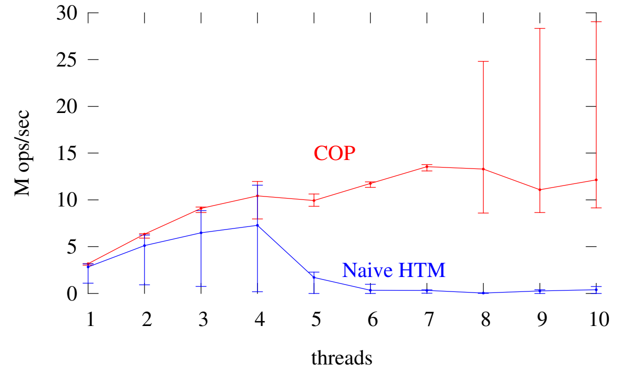



Figure 9: Cache Oblivious B-Tree performance for read-only workloads on a tree containing 100,000 values performing 32 searches per transaction. The horizontal axis is the number of threads. The vertical axis is performance, measured in number of successful searches per second. The error bars show the slowest, the fastest, and the average.

binary search on a one-million element array requires 20 cache lines, 9 of which are on different pages, and 9 of which all reside in the same associativity set, and so even single searches often fail under HTM. The COP code runs between almost exclusively with transactions succeeding in the suffix code, rather than with locks.

The naive HTM version usually fails due to capacity problems when the threadcount equals one. For larger threadcounts, there are a mix of capacity aborts, conflict aborts, and explicit aborts triggered by the suffix code failing validation. For the explicit aborts, we used the 24-bit abort code available in the Intel **_xabort** instruction to determine why the abort happened. Usually the transaction failed because the lock was held. Basically, the transactions failed, the code reverted to the fallback code which grabbed the lock, and then the system was never able to get back into transaction mode, because the lock prevents any transaction from succeeding. This runaway lock problem appears tricky: one way to attack runaway locks is to use backoff, but it is not clear how to do this to get the system back into an HTM mode. In the case of the naive HTM code, it's not clear that there is any alternative, since the transactions usually fail due to capacity.

Under COP, the performance achieved is much better. The verification step typically needs to look at only one cache line.

We do not yet have PMA measurements for a mixed read/write workload, but expect to have that for the final paper.

### 5.3 Cache-Oblivious B-Tree Performance

Figure 8 show the performance of the COBT on a 100,000-node tree. As we can see, the COP implementation outperforms the Naive version both for single threaded and multithreaded workloads. For single threaded workloads, the COP behavior remains essentially flat at about 3.1Mop/s. On single threads, Naive HTM does about the same on average, but has some slow outliers which are about half as fast.

For an 8-threaded workload, the Naive HTM starts quite well for a single query per transaction, but then performance collapses. The COP approach achievs between 10 and 13.5Mop$/s$. The largest speedup seen is about 4.4 compared with a single thread.

Single threaded COBT is a little faster than the PMA without the index tree, and multithreaded COBT is on average similar to the PMA, with much higher variance.

We found that 1,000,000-element trees, the graphs were similar, but that the naive transactions essentially never succeed for more than 15 lookups per transaction.

Figure 9 shows for 32 searches per transaction how the performance of COP and Naive HTM varies with the number of threads. COP dominates HTM, and interestingly HTM has high variance when it works well (sometimes giving very poor perforamnce), whereas until the thread count becomes relatively as large as the number of hardware threads, the COP has little variance. The COP variance at high threads is a good kind of variance: sometimes it runs much faster (getting near linear speedup), rather than the HTM's variance which makes it sometimes run much slower.

## 6. Conclusions

We show COP can make HTM useful in scenarios and data structures which it could not improve in its simple usage pattern.

The PMA is an important part of the infrastructure of some leading in-memory data bases. Without COP, HTM can not complete even a single lookup operation on a quite small, 1M size data-base. With COP, we produce an almost perfectly scalable PMA.

Combining operations is a key feature of TM, and in plain HTM it is limited. For single operations, there are efficient algorithms in the literature, while composing the operations scalably, is the contribution of TM. We show COP greatly improves the scalability of transactions that compose red-black tree operations. It also allows writing values in in the root, as we do in another benchmark. This can be useful, for example, to count the population of the tree. Yet, in plan HTM, keeping this data in the root, will abort all read-only transactions that are concurrent with an update, while With COP, all the read-only transactions can complete successfully.

## References


[1] Y. Afek, A. Morrison, and M. Tzafrir. Brief announcement: view transactions: transactional model with relaxed consistency checks. In *PODC*, pages 65–66, 2010.

[2] Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *OPODIS*, pages 65–79, 2011.

[3] H. Avni, N. Shavit, and A. Suissa. Leaplist: lessons learned in designing tm-supported range queries. In *PODC*, pages 299–308, 2013.

[4] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM J. Comput.*, 35(2):341–358, 2005.

[5] 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*, pages 225–236, 2013.

[6] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPOPP*, pages 67–78, 2010.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

[8] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, pages 93–107, 2009.

[9] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.

[10] D. Kanter. Intel's Haswell CPU microarchitecture, 13 Nov. 2012. `http://www.realworldtech.com/haswell-cpu/5/`.

[11] Z. Kasheff. Cache-oblivious dynamic search trees. M.eng. thesis, MIT, June 2004. `http://people.csail.mit.edu/bradley/papers/Kasheff04.pdf`.

[12] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.