Towards Consistency Oblivious Programming

Yehuda Afek[1], Hillel Avni[1], and Nir Shavit[1,2]

[1] Tel-Aviv University, Tel-Aviv 69978, Israel
[2] MIT, Cambridge MA 02139, USA
hillel.avni@gmail.com

Abstract. It is well known that guaranteeing program consistency when accessing shared data comes at the price of degraded performance and scalability.
This paper initiates the investigation of consistency oblivious programming (COP). In COP, sections of concurrent code that meet certain criteria are executed without checking for consistency. However, checkpoints are added before any shared data modification to verify the algorithm was on the right track, and if not, it is re-executed in a more conservative and expensive consistent way. We show empirically that the COP approach can enhance a software transactional memory (STM) framework to deliver more efficient concurrent data structures from serial source code. In some cases the COP code delivers performance comparable to that of more complex fine-grained structures.

1 Introduction and Related Work

The need to maintain consistency when accessing data has been a major source of overhead in concurrent software and a great limitation on its scalability with respect to its matching sequential code.

There are in the literature concurrent algorithms that reduce the consistency overhead by traversing the data structure while ignoring all locks and meta-data. Examples are the concurrent lazy list [1] and skip list [2] algorithms that allow traversal operations to execute while ignoring the locks that are taken by threads modifying the structure. The traversal correctness is derived from properties of the structure and post validation. Another example is Lee's Java hash table [3], where a thread first traverses the bucket unsafely, and only if the key is not found, takes locks to guarantee consistency and then re-traverses it.

Our goal in this paper is to generalize this approach to derive a broader class of algorithms that execute without verifying consistency. We provide a methodology for designing concurrent algorithms in which we optimistically make a first attempt at executing an operation in a completely un-instrumented way. We will call this approach *consistency oblivious programming* (COP). This paper makes a first attempt at formulating COP and providing examples of its usefulness.

We base our COP generalization in part on using the software transactional memory (STM) programming paradigm. Transactional memory is a leading technique for simplifying concurrent programming. Software transactional memory systems suffer from major overheads because they maintain, in some form or another, a level of consistency among concurrently executing transactions. In [4] Shavit and Touitou allow a transaction to see an inconsistent state but from that point on the operation is considered a zombie, and will not complete successfully. Most modern STM algorithms [5–8] conservatively abort a transaction as soon as the possibility of inconsistency is detected. Others [5, 7, 9] force consistency by having even read-only operations check locks and global clocks or by maintaining multiple versions per address. Consistency, as many authors argue, simplifies the interaction of the STM algorithm with its environment, and allows

simplified correctness proofs, yet it comes at a cost. Each and every access of a transaction to a shared variable or object must be instrumented in some way or another. This instrumentation is a major source of modern STM overhead, which in some cases can be reduced by algorithms such as the NORec STM [6]. These schemes avoid per object meta data, which reduces instrumentation overhead at the price of reduced scalability.

Our approach here is to provide a set of criteria that will allow a programmer to determine if a given data structure can be converted to work in the COP framework. If the conversion is possible, COP allows us to optimistically execute various operations on the data structure in native code, without any consistency checks, then test the outcome, and either retry or resort to a traditional STM based consistent execution if the earlier consistency oblivious ones failed.

1.1 COP and Acceptability Oriented Programming

Our approach follows along the lines of Rinard's acceptability oriented programming [10], where he introduces the idea of allowing programs to execute while making errors, attempting to recover from them only eventually.

In concurrent programs we consider sections of a thread's code (such as, but not only, transactional memory transactions) that can be viewed as having two types of sequential segments. A given section starts with a segment of code that does not affect the system, consisting usually (but not always) of reads, followed by a second segment that writes and updates the memory in addition to further reads. An example could be an insert operation in a data structure where it is first traversed to find the insert location and then writes to memory take place in order to implement the actual insertion of a new item.

In COP we add a third segment of code, called a Validating White-Box, between the first part and the updating part. The idea is to execute the first part, called the Black-Box, in a concurrent environment, as fast as possible without worrying about consistency of any form. Then, in the Validating White-Box, the values returned from the first part are checked to make sure they are in a consistent state with respect to the values returned from the first part. If the validation fails, then the entire operation re-starts.

We allow a transformed section of code to encompass multiple black and white boxes, as long as each black box is followed by its corresponding white box. It is interesting that a black box may include writes, as long as each write, by itself, is a valid addition to the system. We show an example for such writes later, in a union-find compression algorithm.

We say that the output of a black-box is *acceptable* if there exists a consistent execution of the black-box in which exactly the same outputs are generated by the black-box. As we do not check the consistency of the execution, we must extract other properties of the output that imply it is consistent with the system. We call these properties *acceptability properties*.

The programmer must first determine the acceptability properties of the output of the black-boxes and then devise tests that verify that these properties are satisfied given a set of outputs. If any of these tests fail, it means the output of the black-box is not acceptable. We call these tests *acceptability rules*.

We define our approach following Rinard and in the same way. Several activities characterize this approach:

- **Acceptability Property Identification:** The programmer identifies what in the output of the black-box marks it valid, i.e., what are the acceptability properties of the output. These properties are specific to each algorithm and implementation.

- **Enforcement:** Minimal set of acceptability rules is constructed to verify that the output of the black-box is acceptable. Once a rule is violated, it means an acceptability property is violated, and an action which fixes it is taken.
- **Monitoring:** The programmer produces components that enable the acceptability rules testing. These components must indicate whether a rule is violated in the output.

As mentioned, we split the concurrent code to boxes. The boxes terminology is borrowed again from Rinard, into COP, and relates to the above activities:

- **Black Box:** A native, sequential code section. We know the code does not crash or corrupt the system, yet we cannot trust its output.
- **White Box:** The code of this box monitors and enforces the acceptability rules on the outputs of the black box. It must enforce continuous correctness and undo any error that may occur.
- **Gray Box:** The code is not modified, but it is recompiled with synchronization, such as STM. Gray boxes are self contained as they are synchronized and are thus concurrency safe.

The performance advantage of COP is that the black boxes run without any synchronization. Our goal is to have as much code as possible in the black boxes, and as little of it as possible in the white and gray ones. The size of a box is its execution time.

The remainder of the paper is structured as follows. In Section 2 we present the conditions and justifications for the use of COP, and we use an example to demonstrate where it can be most affective. Then in section 3 we show several COP algorithms. In section 4 we evaluate the performance of the COP, and conclude in Section 5.

2 When Can We Use COP

To determine if one can use the COP approach, we must check if a breakup into the three boxes is possible. When checking a sequential section of code, to see if it can be in a black box, we need to examine each shared variable access in that function. If all accesses are reads If the access is a write, we must make sure that it is:

- Based only on committed data, and
- any single write is a valid modification, so we do not need synchronization. If a write is tentative, it must be reversible, and can not be visible, thus it must be instrumented.

The COP un-instrumented write accesses are immediately published and can not be undone. Thus, if the written values are calculated according to un-instrumented reads from transactional addresses, we have to use a commit time locking STM in the system. This means when a value is recorded in a transactional address, the value will be final and ready to use. We also need to see that any subset of the un-instrumented writes is a valid addition to the current state.

An un-instrumented read can be added if its encapsulating data structure precludes:

- permanent loops, even in deleted objects.
- un-initialized pointers, even for an object that is not yet fully connected.

Since COP does not validate consistency, an infinite loop may go undetected, unless it is caught by the serial algorithm. We notice that the second condition applies partially to any STM: this is the privatization problem of [11]. If transaction T_1 reached a node N during traversal and went to sleep, and then another transaction, T_2 freed N, then T_1 could wake up and read N's next pointer. If N is actually freed by T_2, T_1 may prompt an exception. Currently this problem is best solved by quiescence barriers [12].

When inspecting a data structure to see if it is fit for COP, the challenging part is to identify the set of acceptability checks that it requires. In the next section we show useful and common data structures that benefit from COP.

Any COP code has the layout of Algorithm 1 below. It executes a black box (BB) which generates some output V. Then a white box (WB) is used to verify V is acceptable, and finally it executes a gray box (GB), which is compiled with full synchronization using V to complete the operation. Multiple such operations may reside in the same transactions.

Algorithm 1 COP

```
1: V ← BB
2: if failed(WB(V)) then
3:     restart
4: end if
5: if exists(GB(V)) then
6:     GB(V)
7: end if
```

The gray box may be empty, but if we have a black box we must have a white box following it. If there are no black-boxes in a transaction it is not related to COP.

3 COP Algorithms

This section shows the viability of the COP approach by way of a series of examples, presenting COP versions of a linked list data structure, a union-find data structure, and a linked, bottom balanced variant of a red-black tree data structure. The latter structure is a fundamental data structure [13] not known to be parallelizable before.

3.1 COP Linked List

In Algorithm 2 we show the delete function of the COP linked list. In the white boxes we assume the existence of an STM and use **txld**(address) and **txst**(address, value) which are respectively a transactional load and a transactional store. The transactional operations are affecting the read / write set, are validated, and if necessary, aborted, according to the used transactional memory algorithm.

Note that this list supports the standard single node insert, delete, and lookup operations. When we reach a node which has the desired value or greater, we return that node together with its value and its predecessor at line 2. The addition we make to the serial algorithm is the setting of the node's deleted value to MAXVAL in line 13 to prevent BBLookup from continuing looping in the deleted node. Deleted nodes are pointing to themselves, and if their value are less than the looked up one, the code would get trapped in them.

Algorithm 2 COP WBLookup, which deletes a node from a linked list.

```
1: val, n, prev ← BBLookup(ValToDelete))
2: //n is the last traversed node, val is its value and prev is its predecessor
3: if txld(n.val)≠ val then
4:     Abort
5: end if
6: if txld(prev.next) ≠n then
7:     Abort
8: end if
9: if txld(prev.val) ≥ValToDelete then
10:    Abort
11: end if
12: if txld(n.val) = ValToDelete then
13:    txst(n.val, MAXVAL)
14:    txst(n.next, n)
15:    GBDelete(n)
16: end if
```

The acceptance rules are that a successfully found node has the value it is supposed to have in line 5 and is still pointed-to by its predecessor in line 8. The predecessor, in turn, must have a lower value than the one to be deleted, as checked in line 11. If we would not have modified the deleted node to point to itself, it would pass an STM load as consistent, and no white box will be able to notice the node is deleted. If the deleted value would not be set to MAXVAL, the lookup would hit an infinite loop when a lookup is trapped in a concurrently deleted node.

3.2 COP Union Find

The union-find algorithm maintains disjoint sets under union. Each set is represented by a rooted tree whose nodes are the elements of the set. The root is called the representative of the set. The representative may change when the tree is updated by a union operation. The data structure provides two operations: find, which follows the path from the element to the representative and returns it, and union, which links two set representatives by making one point to the other.

The union function calls find for the two elements to be unified and then uses symmetry breaking to decide the direction of the link to be installed. We will look in the find. Actually it is called the find-compress function as it also compresses the path from the element to the representative.

The way FindCompress works is that it follows the next pointers of nodes to their parents until it hits a representative. If a node points to itself, then this node is the representative of its set. Otherwise, if the parent is not the representative, a compression occurs. In compression, the nodes' next pointer is rewritten to point to the grandparent of the node.

Algorithm 3 is searching for the representative of x and compresses the path of x to its representative. Forest is an array of nodes where each node has a next field which is the index of its representative, or a member from its set that is closer to the representative. The interesting thing is that as long as all data read is valid, the write in line 5 does not need any instrumentation, because no matter what value is written by a correct reader, it will perform some helpful compression. However, if transaction T_1 wrote a link and that link was used by T_2 in compression, and then T_1 is aborted, the compression performed will be wrong. Thus, for algorithm 4, we use a commit time STM, which keeps written

Algorithm 3 Transactional FindCompress in union-find.

```
1: next = txld(forest[x].next)
2: while x ≠ next do
3:    t = next
4:    t_next=txld(forest[next].next)
5:    txst(forest[x].next, forest[t].next)
6:    x=t
7:    next=txld(forest[t].next)
8: end while
9: return x
```

values in a buffer. BBFindCompress is the transactional Algorithm 3, replacing each instrumented access with an uninstrumented one.

Algorithm 4 COP WBFindCompress in union-find.

```
1: repeat
2:    x← BBFindCompress(x)
3:    y←txld(forest[x].next)
4:    if x≠y then
5:       x←y
6:       continue
7:    end if
8: until x=y
9: return x
```

Transaction T which calls WBFindCompress(x) gets the representative of x, but then verifies it is the representative by reading the next link transactionally in line 3. Note that there is no read-after write hazard, as any transactional write will replace a self pointing pointer, and a self pointing pointer is always read transactionally eventually.

Now, either x is still pointing to itself, or it was changed within T, or T read it transactionally for the first time, and it is different. This will keep the function running, or it will be read transactionally for the second time by T. Now if x was changed by another transaction, T will abort.

As we will show empirically, the COP version of union-find performs well when there are many unions which introduce memory contention, because in that situation it saves cycles. When there are mostly finds, the path length to a representative becomes one, and there the COP and STM have the same overhead and performance. However, in applications, usually there is a burst of unions when a new network structure is constructed and then the structure becomes read only, where instrumentation can be eliminated by a barrier [14]. Thus, the high contention scenario seems more important.

3.3 COP Red-Black Tree

The next example for COP is a balanced binary red-black (RB) search tree, which is balanced bottom up. Balancing the tree from the bottom, makes the balancing effect unpredictable, so no locking mechanism, except global lock, can be used to parallelize it. The algorithm is taken from [13], where the interface is:

- **Lookup**(K, Tree) which searches the tree and returns the node with key K, or its successor if it is missing as a left son, or its predecessor in case K is missing as a right son.

– **Insert**(K,Tree) which runs Lookup and inserts the node with key K in case it is missing.
– **Delete**(K, Tree) which runs Lookup and deletes K in case it is present.

As can be seen, Lookup is the main part of all functions. Luckily, we can fit it into a black box. We verify that in the serial implementation of the RB tree, no uninitialized pointers are visible and no permanent loops are created. The challenge is then to add the appropriate white box with fitting acceptance rules.

During balancing there are two problematic states that should be addressed. A node can be temporarily detached and missed, or a successfully found node can be deleted by a concurrently executing transaction.

To solve the first issue we exploit the fact that a binary search tree has a total order on its keys and that a lookup in any binary search tree always arrives to the target node from the node with the predecessor or the successor key, depending on whether the key is the left or right son of its parent. In the white box we connect all nodes in a doubly linked list of successor - predecessor. Than we add the acceptance rule that if a node is found missing, then its predecessor must be connected to its successor or vice versa. We maintain the list with synchronization so we know it is correct. The list is not changed during balancing, and takes only a few accesses to maintain. We also have to verify that the node is missing in the parent we found. To determine this we verify the place where the node was supposed to be is still null. The problem of successfully found deleted nodes is solved with a transactionally maintained live mark, which is set by Insert and reset by Delete operations.

Algorithm 5 is the code for the Lookup operation which uses the original RB tree Lookup as a black box and adds the acceptance rules in order to make it always return a node that is valid, and which can be used safely by the other operations. In the code, we first check that the node was not removed by reading its live indication in line 8. Then in lines 14, 25, 25 and 28 we check the predecessor or the successor of the node, to verify that the requested key falls in the gap between them and is thus definitely missing. In lines 20 and 31 we verify the node is not only missing, but is missing in the found location.

In Figure 1 we see a lookup for key 26 that starts when the tree is unbalanced, and goes from the root which is 20 to 30 and to 27, then the tree is balanced, and the lookup continues through 27 back to 20, then to 25 and finally to 26. During the lookup we go through the right link of 20 to two different nodes. Before balancing we go to 30 and after we go to 25. This operation would abort on all known single version STM algorithms. The only known STM that will endure it is the multi version [9] which will introduce high overhead. As said, no existing locking protocol, including the new DL [15], will tolerate such a rebalancing move during lookup either.

In Algorithms 6 and 7, we see that Delete and Insert in the RB tree, which use the WB lookup and then insert or delete the node in the list in line 4 and set or unset its mark in line 5. Though it is a subjective matter, we found that the amount of work required to convert the bottom balancing serial RB tree we started from, into an efficient concurrent algorithm, is small and the correctness of it is easy to verify. If we wanted to keep all these features in a fully hand crafted algorithm we would end up spending a lot of work on and would involve a complex proof. Thus another good side of COP is that it is engineer friendly and saves work as well as inevitable bugs.

4 Evaluation

We first evaluated COP for the union-find algorithm. We started with an implementation of the classical union-find serial algorithm from [13] and transactified

Algorithm 5 COP WBLookup in RB tree.

```
1: n ← BBLookup(K, Tree)
2: if txld(n.live)≠ TRUE then
3:    Abort
4: end if
5: if n.key = K then
6:    if txld(n.key) ≠ K then
7:       Abort
8:    end if
9:    return
10: else
11:    if n.key < K then
12:       if txld(n.key) > K then
13:          Abort
14:       end if
15:       if txld(txld(n.successor).key) < K then
16:          Abort
17:       end if
18:       if txld(n.right) ≠ ⊥ then
19:          Abort
20:       end if
21:    end if
22: else
23:    if txld(n.key) < K then
24:       Abort
25:    end if
26:    if txld(txld(n.predecessor).key) > K then
27:       Abort
28:    end if
29:    if txld(n.left) ≠ ⊥ then
30:       Abort
31:    end if
32: end if
33: return n
```

Algorithm 6 Insert in RB Tree.

```
1: n ← WBLookup(key)
2: GBAddToTree(new_node)
3: GBBalanceTree
4: WBInsertToList(new_node)
5: txst(new_node.live, true)
```

Algorithm 7 Delete in RB Tree.

```
1: n ← WBLookup(key)
2: GBRemoveFromTree(new_node)
3: GBBalanceTree
4: WBRemoveFromList(new_node)
5: txst(new_node.live, false)
```

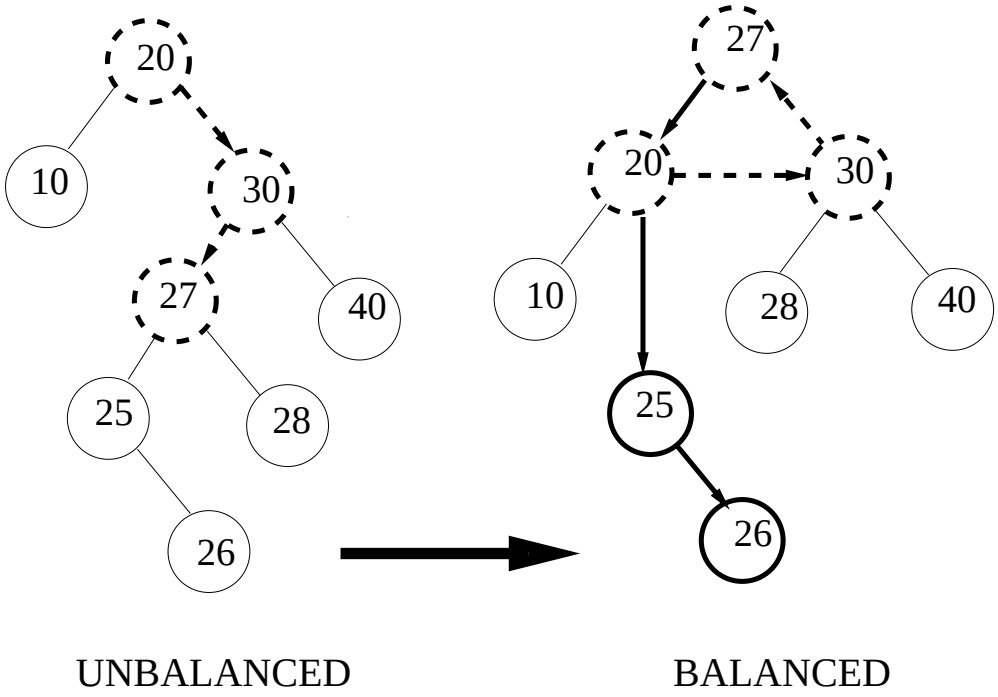

Fig. 1. COP RB Tree is finding a key 26 during rebalancing.

it using the TL2 STM [5]. We then created the COP version of the union-find according to the description in section 3.2. We compare the above to an optimized implementation of the parallel union-find algorithm from [16].

Next we examined the performance of COP for RB tree. We started with the classical red-black tree from [13] as it was transactified by Dave Dice. We derived the COP RB tree as described in 3.3. We compared the COP to the TL2-Enc [5] and NORec [6] STM, early release elastic transactions [17], and to Hanke's concurrent RB tree algorithm [18]. Later we will focus on comparing the COP RB tree to STMs and elastic transactions, because Hanke's tree is relaxed and top-down balanced, so its comparison to the bottom up and perfectly balanced RB tree from [13] is misleading.

Experimental setup: We collected results on two hardware platforms: a Sun SPARC T5240 and an Intel Core i7. The Sun is a dual-chip machine, powered by two UltraSPARC T2 Plus (Niagara II) chips. The Niagara II is a chip multithreading (CMT) processor, with 8 1.165 GHz in-order cores with 8 hardware strands per core, for a total of 64 hardware strands per chip. The Core i7 920 processor in the Intel machine holds four 2.67GHz cores that each multiplex 2 hardware threads.

4.1 Union Find

As explained in section 3.2, each representative lookup compresses the path from the element to the representative, so after a certain level of find operations, almost all paths are of length one. In this situation the COP and the transactional algorithm will perform the same number of instrumentations and their performance will be the same. However, usually the union-find is used to

build a network and then is used in read only mode, so the practically important scenario is when the amount of unions is high.

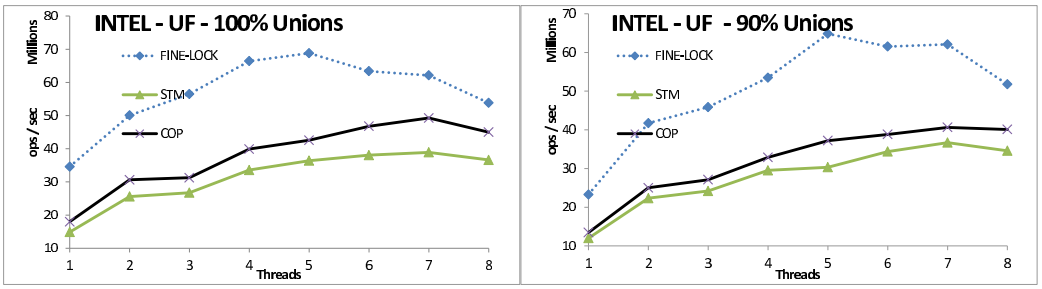

Fig. 2. Union Find with different amount of unions, benchmark results on Intel Nehalem machine with 4 cores, each multiplexing 2 hardware threads.

In Figure 2, most 90-100 percents of the operations are the mutating union. As seen, when the amount of union operations is high, the COP performance is closer to the optimized hand crafted concurrent algorithm than to the STM version. We note that the COP is simple to develop and is composable and may participate in STM transactions.

4.2 Red Black Tree

Our bottom up and perfectly balanced serial algorithm does not have a concurrent, non-transactional version. Thus, we chose the top down, locally balanced Hanke algorithm as an upper bound for the performance of the concurrent red-black tree.

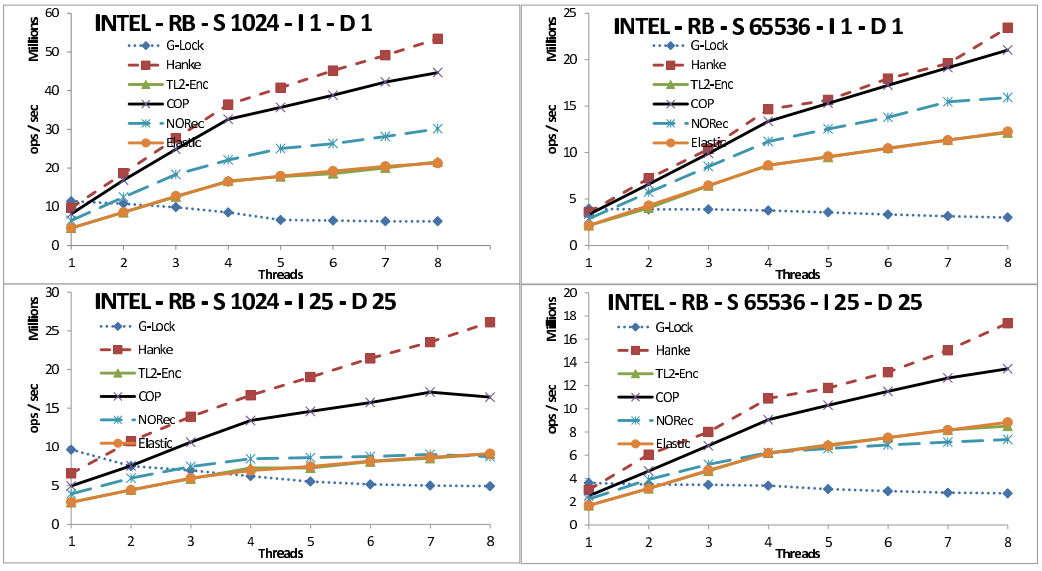

Fig. 3. RB Tree benchmark results for 1024 and 65536 nodes on Intel Nehalem machine. The benchmarks have 50% lookups with 25% inserts and 25% deletes or 98% lookups with 1% inserts and 1% deletes.

In Figure 3 we show the COP RB Tree throughput compared to Hanke and a global lock as upper and lower performance bounds. In addition, we compare to the NORec, TL2-Enc and Elastic STM algorithms. The results show that COP consistently outperforms all other transactional variations, for all tested combinations of tree size and contention level. However, it is not as good as the more loosely balanced Hanke algorithm.

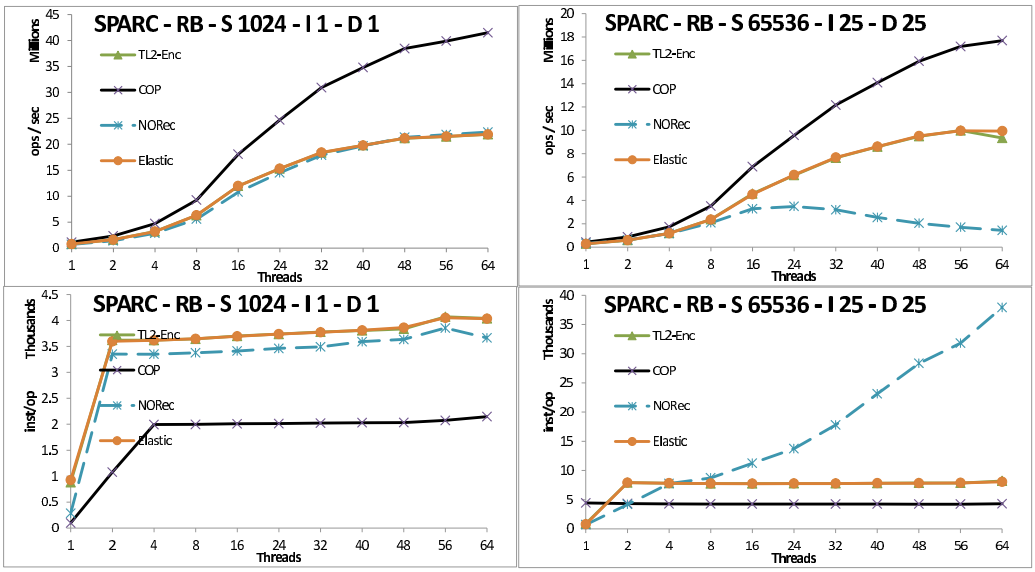

Fig. 4. RB Tree of 1024 and 65536 nodes. Graphs show rate of completed operations per second and hardware performance instruction counter per operation. The benchmarks have 50% lookups with 25% inserts and 25% deletes or 98% lookups with 1% inserts and 1% deletes.

In Figure 4 we compare throughput in the first row and cycles per operation in the second row, and see a perfect match. The lower the count, the higher the throughput. In this case the COP algorithm has the lowest count and respectively has about twice the throughput.

As our COP RB tree contains links, which can be useful for range queries, we present its comparison to the STM version of RB tree with links, i.e., trees whose nodes are chained in a predecessor - successor doubly linked list, in Figure 5. As expected, the COP wins by a large margin.

5 Conclusion

In this paper we introduced consistency oblivious programming, an approach for efficient parallelization of serial code. It manages to reduce significantly the footprint of synchronization compared to automatic transactification or coarse locking.

The reasoning needed to parallelize an RB tree using COP is negligible, while its performance compares, in some cases, to that of the celebrated concurrent algorithm of Hanke [18]. In contrast, locking protocols such as hand over hand or DL [15] are not as flexible as COP and are not applicable, for example, to the RB tree we converted.

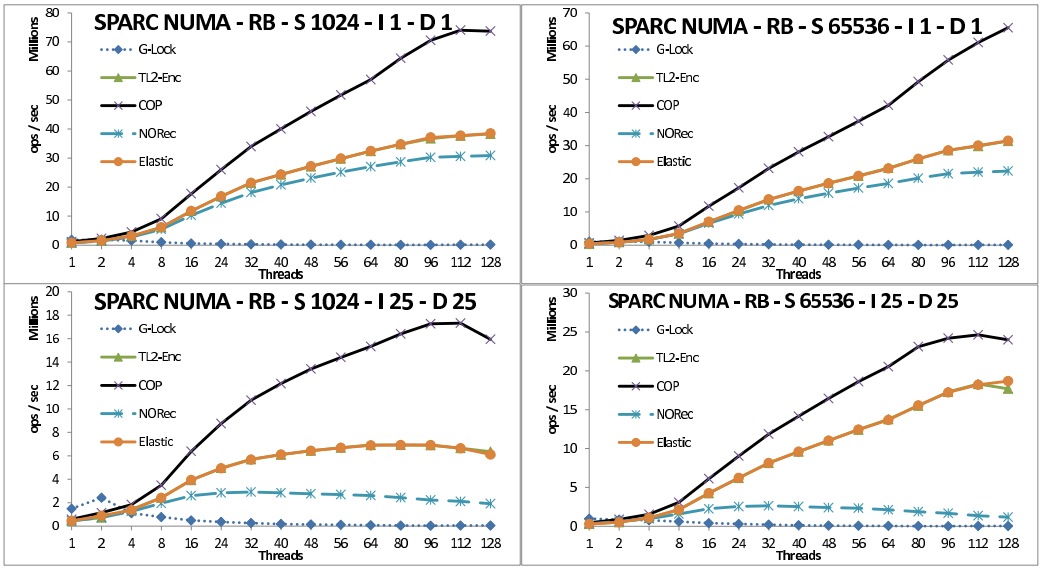

Fig. 5. RB Tree of 1024 and 65536 nodes with 50% lookups and 25% inserts and 25% deletes or 98% lookups with 1% inserts and 1% deletes when using both processors of the Sun machine (i.e., NUMA configuration). Default OS policy is to place threads on chips in round robin order.

All the COP algorithms we presented are composable and have controllable number of transactional accesses, characteristics which make COP HTM friendly.

References

1. Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W.N.S., Shavit, N.: A lazy concurrent list-based set algorithm. Parallel Processing Letters **17**(4) (2007) 411–424
2. Herlihy, M., Lev, Y., Luchangco, V., Shavit, N.: A simple optimistic skiplist algorithm. In: SIROCCO. (2007) 124–138
3. Lea, D.: The java.util.concurrent synchronizer framework. Science of Computer Programming **58**(3) (2005) 293 – 309 Special Issue on Concurrency and synchonization in Java programs.
4. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing **10**(2) (1997) 99–116
5. Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: DISC. (2006) 194–208
6. Dalessandro, L., Spear, M.F., Scott, M.L.: Norec: streamlining stm by abolishing ownership records. In: PPOPP. (2010) 67–78
7. Riegel, T., Fetzer, C., Felber, P.: Time-based transactional memory with scalable time bases. In: SPAA. (2007) 221–228
8. Herlihy, M., Luchangco, V., Moir, M., Scherer, III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the twenty-second annual symposium on Principles of distributed computing. PODC '03, New York, NY, USA, ACM (2003) 92–101
9. Afek, Y., Morrison, A., Tzafrir, M.: Brief announcement: view transactions: transactional model with relaxed consistency checks. In: PODC. (2010) 65–66
10. Rinard, M.C.: Acceptability-oriented computing. In: OOPSLA Companion. (2003) 221–239
11. Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing isolation and ordering in stm. SIGPLAN Not. **42** (2007) 78–88

12. Matveev, A., Shavit, N.: Implicit privatization using private transaction. TRANSACT (2010)
13. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press and McGraw-Hill Book Company (2001)
14. Scott, M.L., Spear, M.F., Dalessandro, L., Marathe, V.J.: Delaunay triangulation with transactions and barriers. In: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization. IISWC '07, Washington, DC, USA, IEEE Computer Society (2007) 107–113
15. Golan-Gueta, G., Bronson, N., Aiken, A., Ramalingam, G., Sagiv, M., Yahav, E.: Automatic fine-grain locking using shape properties. In: SPLASH. (2011) 194–208
16. Schrijvers, T., Frühwirth, T.: Optimal union-find in constraint handling rules. Theory Pract. Log. Program. **6** (2006) 213–224
17. Felber, P., Gramoli, V., Guerraoui, R.: Elastic transactions. In: Proceedings of the 23rd international conference on Distributed computing. DISC'09, Berlin, Heidelberg, Springer-Verlag (2009) 93–107
18. Hanke, S.: The performance of concurrent red-black tree algorithms. In: Algorithm Engineering. (1999) 287–301