Consistency Oblivious Programming

Thesis submitted for the degree of Doctor of Philosophy
by
Hillel Avni

This work was carried out under the supervision of
Professor Yehuda Afek and Professor Nir Shavit

Submitted to the Senate of Tel Aviv University
April 2014
© 2014
Copyright by Hillel Avni
All Rights Reserved
Abstract

The concurrency control mechanism known as transactional memory (TM) has been accepted as mainline. It is now present in GCC, arguably the most important open-source C/C++ compiler today, as well as in most popular commodity hardware. Although many innovations can be expected to emerge from TM research, TM is already a standard feature in the toolkits of many programmers.

However, the naive programmer who tries to use TM will be disappointed to find that its limitations render it inefficient in many common workloads. To address these limitations, this thesis proposes the use of consistency oblivious programming (COP) [1] methodology to design data structures with TM, both in hardware and in the compiler, so as to yield good performance and scalability in programs where TM alone had not previously been useful.
Contents

1 Introduction 1

2 Background 4
   2.1 Terms in Concurrent Programming 4
   2.2 Concurrent Data Structures 6
      2.2.1 Fine Grained Locking 6
      2.2.2 Non-Blocking 6
      2.2.3 Lazy Synchronization 7
   2.3 History of TM 8
   2.4 Properties of TM 9
   2.5 TM’s Inherent Limitations 10
   2.6 Previous Approaches to Overcoming TM Limitations 10
   2.7 COP in a Nutshell 12

3 The COP Methodology 13
   3.1 The COP Template 13
      3.1.1 Operation Structure 14
      3.1.2 Correctness Proof Method 15
   3.2 A COP Based RB-Tree 16
      3.2.1 Algorithm 17
      3.2.2 Correctness 19

4 STM with COP 21
   4.1 Introduction 22
   4.2 TM-Pure and Suspended Semantics 23
   4.3 STM COP Template 24
4.4 Suspended Transactions and COP Composing .......................... 25
  4.4.1 Safety ......................................................... 27
  4.4.2 Memory Reclamation ........................................ 27
4.5 STM COP RB-Tree .................................................. 28
4.6 Integrating COP and Applications ................................. 29
4.7 Evaluation ......................................................... 30
  4.7.1 COP in Applications ....................................... 30
  4.7.2 STM with COP vs. Plain TM ................................. 32
4.8 Summary .......................................................... 34

5 HTM with COP ......................................................... 35
  5.1 Improving HTM Scaling with COP ............................... 36
  5.2 HTM COP Template ............................................. 36
  5.3 HTM COP RB-Tree ............................................... 38
  5.4 Cache-Oblivious B-Tree ...................................... 41
  5.5 Evaluation ....................................................... 44
    5.5.1 RB-Tree Performance .................................. 46
    5.5.2 PMA Performance ....................................... 48
    5.5.3 Cache-Oblivious B-Tree Performance .................. 49
  5.6 Conclusions .................................................... 50

6 Summary ............................................................ 51

7 Future Work ......................................................... 52

Bibliography .......................................................... 54
List of Figures

1.1 Locking vs. TM usage ........................................... 1

3.1 Generic COP template ........................................ 14
3.2 COP and TM search for key 26 in an RB-Tree ................ 16
3.3 RB-Tree insert operation Fit in generic COP template ....... 17
3.4 RB-Tree insert ROP in generic COP template ............... 18
3.5 RB-Tree insert ROP verification in generic COP template ... 18

4.1 STM COP Template ............................................. 25
4.2 COP RB-Tree in STAMP and in various configurations ...... 31
4.3 STAMP Vacation Statistics ..................................... 31
4.4 RB-Tree transactional loads .................................. 33

5.1 HTM COP template ............................................. 37
5.2 An example dynamic cache-oblivious B-tree. .................. 43
5.3 Associativity limits on Haswell HTM .......................... 45
5.4 RB-Tree various transaction sizes ............................. 46
5.5 RB-Tree contentious workload ................................. 46
5.6 PMA performance for read-only operations. .................. 48
5.7 Cache Oblivious B-Tree performance .......................... 49
Chapter 1

Introduction

As increasing microprocessor clock speed becomes energy inefficient, and instruction level parallelism, which also is transparent to the programmer, is becoming too complex, due to pipeline dependencies, new general purpose chips are evolving by accommodating more cores. In this situation, software can only benefit from newer hardware by exploiting parallelism.

If data can be partitioned by the programmer or by the program so that each process operates on its own part of the data, parallelism is free. If data cannot easily be partitioned, however, or if there could be concurrent operations by different threads on the same part of the data, then threads will have to synchronize their operations. The simplest way to synchronize is to use a single global lock to protect accesses to shared data, as demonstrated in Figure 1.1. Locking is simple, but it limits scalability. Amdahl’s law [6] shows an upper limit on the program speedups one can achieve:

\[
S = \frac{1}{(1 - P) + P/N} \tag{1.1}
\]

P is the proportion of the program that can execute in parallel and N is the number of CPUs. In other words, one can obtain a speedup that grows with the number of CPUs.

<table>
<thead>
<tr>
<th>Lock synchronization</th>
<th>TM synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong> : parameters</td>
<td><strong>output</strong>: rc</td>
</tr>
<tr>
<td><strong>output</strong>: rc</td>
<td><strong>input</strong>: parameters</td>
</tr>
<tr>
<td>1 lock;</td>
<td>4 tx_start;</td>
</tr>
<tr>
<td>2 rc ← operation(parameters);</td>
<td>5 rc ← operation(parameters);</td>
</tr>
<tr>
<td>3 unlock;</td>
<td>6 tx_end;</td>
</tr>
</tbody>
</table>

Figure 1.1: Locking vs. TM usage
CPUs only if the fraction $P$ can be made sufficiently large; any sequential execution caused by synchronization will be an obstacle and decrease the maximum speedup.

There are two ways to make $P$ larger, i.e., decrease the serialization caused by locking, in the context of data structure design. One is to develop efficient algorithms per data structure that either break the global lock into multiple fine grain locks, or else eliminate the locks altogether. In this option, every data structure requires a special and complicated treatment, and the operations are not composable, i.e., it is not possible, or it is very difficult, to execute multiple such operations as one atomic block without reintroducing the global lock.

The second option is to keep the semantics of the single global lock, but minimize the serialization involved by monitoring the protected code execution and serializing only operations that actually connect with one another. This is the TM option, which implements single global lock atomicity (SGLA) [34]. TM permits the composition of operations, as it implements SGLA, and, as seen in Figure 1.1, it is simple to use. However, monitoring all accesses involves high overhead, even when done in hardware, and to treat all operations as unstructured sets of memory accesses makes it hard to identify false connections, i.e., benign situations that occur when one operation is writing an address that was read by another operation.

COP can integrate these two options. On one hand, COP uses TM to achieve SGLA compositability and simplicity. On the other hand, COP uses insights from data structure research to reduce the overhead and false connections of TM.

**Contributions.** This dissertation presents *consistency oblivious programming* (COP), a methodology for using the power of TM in designing data structures to overcome the limitations of the TM paradigm. COP is introduced as a method to work with any TM, and then tailored to use compiler STM support and to meet the constraints and architecture of HTM.

With COP, we manage to leave the read-only prefix of an operation in a non-transactional context. This eliminates a major part of the conflicts, and eliminates the overhead associated with them.

To show its power, we use COP to build concurrent versions of an RB-Tree with chained leaves, a cache oblivious B-Tree (COBT), based on a COP packed memory array (PMA)[7], and a Leaplist [8], which is a data structure based on a Skiplist and tailored
for consistent range queries. For PMA, plain HTM is not capable of accommodating even a single lookup for a range of 1M keys, while the COP version scales almost perfectly.

For the chained RB-tree, the PMA, the COBT and the Leaplist, we are not aware of any non-transactional concurrent algorithm. For all these data structures, plain TM either has an unacceptable overhead in software or scalability in hardware, while COP versions scale nicely and efficiently.

**Roadmap.** In Chapter 2 we explain the basic terminology in synchronization that is used throughout the thesis. It also contains a short review of concurrent data structures, followed by a brief history of TM. Then we focus on the inherent limitations of TM and explain in detail how COP resolves them.

In Chapter 3, we fully describe the COP methodology and provide a generic template for a COP operation, independent of the specific data structure and TM setup. This template will facilitate the conversion of sequential operations to COP operations, as well as the confirmation of the correct functioning and the progress of the COP operations.

Chapter 4 explains how the COP idea is connected with the STM support in the compiler, i.e., how the COP operation will be executed using the features available in the compiler and how these COP operations can be composed and integrated into an arbitrary STM transaction.

Chapter 5 describes COP adjustment to HTM of commodity hardware, i.e., how to use the current structure of an HTM transaction to create COP operations, and how these operations are more efficient than their plain HTM counterparts.

Chapter 6 summarizes the significance of COP in rendering TM useful, and thus COP’s contribution to concurrent programming.

In Chapter 7 we describe the work to be done in theory, TM infrastructure and TM based applications, so as to gain the advantages of COP.
Chapter 2

Background

In this chapter we provide information about the two lines of research that are integrated in COP: concurrent data structures and TM, and then explain how COP synergizes them.

2.1 Terms in Concurrent Programming

This section explains some concepts that are used throughout the thesis. Other concepts, such as caches, NUMA, and specific types of locks, which also are important in concurrent programming, but are not relevant to this thesis, are not specifically mentioned; nonetheless they can be learned from [46].

**Correctness.** The notion of linearizability [40], means an operation (or set of operations) that appears, to the rest of the system, to occur instantaneously. Linearizability is a common correctness criterion that implies an algorithm is operating according to its sequential specification.

**Progress.** Concurrent algorithms can be identified according to the progress they guarantee. We say an algorithm is wait-free [43] if it ensures that every thread will continue to make progress in the face of an arbitrary delay (or even a failure) of other threads. It is lock-free if it ensures only that some thread always makes progress. If an algorithm guarantees progress only to a thread that eventually executes in isolation, it is obstruction-free. The lowest level of progress is present when locking is involved. At this level, there is no progress guarantee.
2.1. TERMS IN CONCURRENT PROGRAMMING

**Locks.** The most intuitive way to prevent concurrent threads from interfering with each other is to protect sections of the code that access shared data with a lock. A thread must *acquire* the lock before entering, and must *release* it after exiting the hazardous code. There are many different types of locks that are tailored for different hardware architectures and application requirements, but each of them supplies mutual exclusion, i.e., only one thread manages to acquire the lock at a time and enter the code. Using a single global lock is simple and allows arbitrary code to execute atomically.

Splitting the single global lock into multiple locks that protect different objects or states requires an effort by programmers and is prone to error. Furthermore, objects either have to expose locks that are stored in the object, or callers have to manage locks for components that they use; both strategies break information hiding. If operations composition is desired, a global order on all locks must be chosen in order to avoid deadlocks.

In the context of data-structures, there are fine-grained locking techniques that are based on the shape of the data, as is detailed in Section 2.2. These methods are efficient, but they are encapsulated in the algorithms, which makes composition even more challenging.

**Compare and Set (CAS).** Simple Load and Store instructions are atomic, in the sense that all read bits are from the same write. Atomicity, however, is not guaranteed for more than one instruction, thus it is easy to see that threads cannot safely modify memory locations without the possibility of overwriting concurrent writes at the same location. This fact prevents multiple threads from performing a consensus operation, or even from maintaining a lock or a shared counter. The ultimate solution for this is the construct of a hardware transaction, wherein any compound statement can be atomically executed, but, most architectures and algorithms in the literature use the *CAS* atomic instruction. CAS receives as parameters an address, an expected value and a new value. Then, if the address holds expected value, it sets the address to the new value and returns success, otherwise it returns a failure. The reading and the writing of the address are made atomic by the hardware, i.e., no concurrent instruction could have overwritten the expected value if the CAS succeeded.
2.2 Concurrent Data Structures

Perhaps the most scalable concurrent data structure is the hash table, in which different operations naturally operate on disjoint memory locations and thus do not disturb one another. Unfortunately, hash tables always use their maximum capacity of memory, and are unordered. Whenever we need to maintain many sets or ordered sets, it therefore is better to use a linked list, which always uses memory proportionate to the number of items that occupy it. If fast searches and range queries are required in the ordered sets, different types of binary, and more generally, k-ary, trees are desired. We will discuss locking and non blocking algorithms, and then look in lazy data structures, which are also the inspiration for the COP method.

2.2.1 Fine Grained Locking

Assigning locks to small parts of the data structure, allows the creation of scalable concurrent data structures. Locking the minimum set of locks per operation, aims to let as many operations progress simultaneously. In hash tables, it is easy to see how locking a bucket or a small number of buckets, while accessing them, can yield good results. However, with more complicated operations, more sophisticated schemes are involved, to ensure both performance and deadlock freedom.

A well-known lock-based technique for creating such operations is hand-over-hand locking which decreases the duration over which locks are held by releasing locks on nodes who can no longer affect the correctness of the operation. Applying this technique to B-Trees is described in [10], but it can also trivially be applied to simpler concurrent DAGs, such as linked lists and binary trees. In [32], they explain how to use shape analysis to automate the fine-grain-locking algorithms,

2.2.2 Non-Blocking

When the number of threads in the application is passing the number of processors, the algorithm must handle unpredictable long delays, due to OS scheduling. If a thread is blocked while holding a lock, for example, on the head of a linked list, or the root of a tree, all other threads can not use the data structure. Lock-free algorithms were crafted to handle these situations. In a lock-free, operation, the traversal of the nodes is done without any locking, and auxiliary nodes [58] or marks [50] are incorporated to
allow verification of their validity. Updating operations, such as insert or delete, are using one CAS operation to change the data structure. In this way, the data structure is left in a valid state, and can be accessed continuously, regardless of the workload or the state of the working threads. The scope of lock-free data structures evolved to include unbalanced binary trees [26], and k-ary trees [16], that were adjusted to support lock-free range queries in [14]. Lately, a technique for lock-free balanced trees was introduced in [15].

### 2.2.3 Lazy Synchronization

A hybrid of the lock-free and fine-grain-locking is the lazy synchronization. The lazy operations are locking only a minimal set of nodes and only for updates.

Like lock-free data structures, the lazy implementations maintain the data structure in such states that an unsynchronized traversal will never crash for an uninitialized pointer, nor enter an infinite loop. This feature allows a search operation to ignore all concurrent operations. Furthermore, the data-structure is designed in such a way, that when an unsynchronized search completes, it returns a linearizable output, e.g., if it found the a key k, than k was in the set sometime during the search.

In case of an update, a minimal set of nodes in the data structure is locked, before the algorithm is writing. In lazy-list [39] algorithm, the traversal is executed with no locking, and only the found location, where the update will occur, finally is locked. The lazy algorithms, like the lock-free ones, add a mark to the data structure node [39, 42], which is used, in the lazy algorithms, to verify the nodes are still part of the shared data structure.

The performance evaluation of [39] shows that adding the lazy searches to the lock-free list from [50] improved its performance by an order of magnitude.

The basic technique of lazy linked-list, described above, was developed in order to design a high performance lazy Skiplist [42], which allows fast searches.

[12] presents a lazy, relaxed balance AVL tree. The tree traversals are not locking, but do perform synchronization, by checking a time stamp for each node they encounter. Before modifying a node, the algorithm is locking it and increments its time stamp. A similar method was later used in [4] to create an efficient splay tree.

In COP we manage to extend the lazy technique for more data structures, such as B-Tree and Leaplist, while avoiding the continuous verifications of [4, 12]. In addition,
as we use TM, the operations of the COP data structures are composable.

### 2.3 History of TM

The TM algorithms and API will be given in detail, wherever necessary in Chapter 4 and Chapter 5. This section only highlights some milestones and directions in TM research and industry adoption. A more comprehensive survey can be found in [36].

TM research started in 1993, in [45], where TM was suggested as a hardware feature. In 1995, STM was proposed by [56]. These two papers initiated TM research. The notion of transactions in these papers was static, however, i.e., all accessed locations must have been declared at the beginning of the transaction.

In 2003, the first dynamic STM was offered in [44], and [37], which also represented the first step toward TM compiler support by adding TM constructs to Java. These STM realizations placed no constraints on addresses accessed in a transaction, which allowed them to become applicable to mainstream software.

A characteristic of many scalable STM algorithms is that they tag every memory location with a version number. The theory behind this method is found in [55]. Tags, coupled with a global version clock, allow invisible readers to verify consistency with relative efficiency, and are used in contemporary compiler support [2].

In 2006, [27] showed that efficient STM may use locks in its implementation, and [25] coupled the lock with the version to create the TL2 algorithm.

In 2007, several authors [9, 31, 52] integrated TM into the C and C++ programming languages, relying upon the compiler to translate the developer marks of a transaction, to the TM realization, so that the application developer does not directly call the TM library. In [31] the first publicly-available TM support for a C/C++ compiler was introduced.

Although the idea of HTM was introduced more than twenty years ago in [45], it only materialized quite recently in Sun’s Rock CPU [18] (which was later canceled), by Intel in 2013 [47], and is scheduled to be released by IBM in 2014 [17].

Transactional language constructs for C++ have recently been proposed for standardization; their draft specification has been the result of a collaboration between several companies since 2008, and, since 2012, is being developed under the umbrella of Study Group 5 of the ISO C++ committee. GCC supports most of this draft speci-
2.4 PROPERTIES OF TM

The TM paradigm provides the programmer with an abstraction, which is the transaction [33] itself. Transactions make concurrent programming as easy as using critical sections, and, potentially as efficient as fine-grained locking. Various TM realizations attribute the transactions with different safety and correctness characteristics, such as:

**Strong isolation** (SI) [49], in which transactions are isolated, both from other transactions and from concurrent non-transactional accesses, is the strongest correctness guarantee. SI implies the opacity and privatization qualities that are discussed below. It is, however, impractical to force SI in an STM [21], as it would imply that every non-transactional access will be converted into a small transaction, which, in turn will introduce fat too high an overhead. In HTM, SI is the standard guarantee, because hardware monitors all accesses, transactional and non-transactional, and can efficiently react to conflicts between them. If a non-transactional write hits a location that is concurrently being used in a transaction, the transaction aborts.

**Serializability** [53] means that all committed transactions in a history H issue the same operations and receive the same responses as in a sequential history S that only consists of the transactions committed in H. (A sequential history is, intuitively, one with no concurrency between transactions.) Serializability is a commonly required property in data-bases, as well as in TM, but with TM, we also need to restrict the internal states of the transactions.

**Opacity** [35] is serializability with the additional restriction that live, non-committed transactions are prevented from accessing inconsistent states. Capturing opacity in STM is not trivial, and involves overhead of maintaining versions and revalidations. Sometimes, opacity is compromised to improve performance [20].

The **privatization** property [57] for a transaction T means that if T detached a node from a shared data structure, once T successfully committed, the node is no longer shared. A side effect of privatization is that a detached node can be reclaimed. With HTM and SI, privatization is implicitly given by the TM, but the standard STM algorithms [25, 29] require that a writing transaction must execute a quiescence barrier [23] after commit, for privatization.
2.5 TM’s Inherent Limitations

The introduction of TM into compilers and hardware might seem to imply that transactions are easy to use, and that a programmer only needs to mark the atomic sections with transaction delimiters. It is implied that creating efficient concurrent data structures is especially easy. Simply take a good sequential implementation of the data structure, and put each operation in a transaction, as indicated in Figure 1.1. Moreover, it is implied that the many techniques in concurrent data structures, developed throughout the past thirty years of research, can each be dismissed and replaced by TM. These implications might be true in theory, but, in practice, TM has fundamental limitations, especially in hardware and in the compiler.

A TM transaction maintains read and write sets, either through software (in software transactional memory, STM) or in hardware (in hardware transactional memory, HTM). At commit time, the TM infrastructure must verify that the Read set is a snapshot, and must atomically update the Write set, relevant to that snapshot [22, 25, 56]. This order of operation implies certain limitations, both in performance and in adaptation. As the TM must log every access, it must either fit in the hardware cache for HTM or be explicitly logged by software. This logging forces STM to call a function per access, i.e., instrumentation. If a transaction overflows the cache in HTM, it fails, while, in STM, an instrumentation makes a memory access consume many more resources than the original load or store had demanded.

Once the TM transaction has logged the accessed address, it continuously monitors all future accesses in the system in order to verify that the address is not externally modified. If a monitored address is written by another concurrent thread, the monitoring transaction fails, due to the conflict. If the written address is no longer being used by the failed transaction, this failure is unjustified.

2.6 Previous Approaches to Overcoming TM Limitations

TM’s practical problems have motivated various research efforts, including TM algorithms and tailored TM-friendly data structures.

The boosting [41] family of STM algorithms only uses TM as an operation composition method. It assumes that every method has an inverse, and it creates transactions
2.6. PREVIOUS APPROACHES TO OVERCOMING TM LIMITATIONS

that are built by these methods. This approach relies upon the efficiency of existing
data structures to bypass the overhead of TM. It resolves conflicts by semantic locking,
in which each method protects the area in the data that it is going to access. In this
way, TM is only dealing with semantic conflicts, while the actual interleaving of the
accesses is being managed by underlying methods. This approach yields high perfor-
mance, but it is limited to reversible methods, and it does not benefit from hardware
and compiler support.

Another STM algorithm that only supports composition is transactional predication
[13]. Like [41], it relies upon existing concurrent libraries, but, instead of logging reverse
operations, it logs specific locations at which data should be updated by the transaction.
If the transaction fails, their value remains unchanged or is replaced by an empty slot,
in the case of insertion. This approach yields highly complicated algorithms, and, as
no method has yet been offered for releasing the empty slots, it uses an uncontrollable
amount of memory.

A set of algorithms was developed to reduce the overhead of STM. These algorithms
relax some of the correctness requirements of the original TM, and, in exchange, save
some of the overhead and the false conflicts. Elastic [30], and view transactions [5] do
not log some of the accessed addresses, thus avoiding a part of the transactional work.
These algorithms improve performance, but they preserve much of the overhead and
grant an application access to transactional logs, which places a burden on the devel-
oper, and is not possible with GCC compiler architecture. These algorithms therefore
are unlikely to be part of a practical STM solution.

In [24], they use small HTM transactions to create concurrent algorithms for queues
that are simpler than their non transactional counterparts. While this method demon-
strates the power of HTM, it is not generally used.

TM-friendly data structures are using other techniques to reduce conflicts. For
example, [19] is decoupling the balancing of the binary tree from the updates, which
manages to avoid some of the conflicts in highly contentious workloads. These tech-
niques, however, are not mitigating the overhead associated with TM.

To benefit from the compiler support for STM and from HTM, COP [3] and, later,
[60] and [59] are leaving the read-only prefix of the atomic operation out of the trans-
action. This lazy approach stretches the usability of TM, while utilizing the hardware
and compiler support.
2.7 COP in a Nutshell

COP takes advantage of both TM and contemporary research developments of data structures. In a sense, it uses TM to generalize the lazy approach, mentioned in Section 2.2. Consequently, it permits a relatively simple conversion of sequential operations to efficient, scalable and composable concurrent operations. COP enables the developer to design such operations for complex data structures that do not yet have any known concurrent version.

The developer uses knowledge of the data structure algorithm to extract a read-only prefix (ROP) of the operation, and to verify that this prefix does not crash or hit an infinite loop, when no synchronization is involved. This prefix returns an output that either is the output of the operation or the input to the completion of the operation. Completion, here, means any updates necessary to finish the operation. In an insert function of an RB-Tree, for example, the updates may include connecting a new node to the tree and balancing it.

After extracting the ROP, the developer uses a TM transaction to perform two actions atomically, to verify that the ROP output is valid, and to complete the updates. At this point, the transaction may continue to execute any other code.
Chapter 3

The COP Methodology

The principle behind COP is simple: Just execute the read-only prefix (ROP) of a data structure operation as part of a transaction, but without the overhead of the transaction. This implies that the ROP will perform un instrumented accesses to shared memory in STM, and that its accesses will not leave a transaction footprint in HTM, and will not subsequently be monitored in the transaction. Conversely, the ROP must see any value that had been written in the transaction before the COP operation started. After the ROP has run and generated output, a transaction starts or continues, verifies the output, and uses it to perform any updates.

This chapter provides a general template for a COP operation algorithm and correctness proof. The following chapters will explain how to port this template to specific HTM and STM implementations. Each chapter will fit an RB-Tree with chained nodes into the template. This data structure, as far as we know, has no other concurrent version, except placing its operations in plain TM transactions.

The results in this chapter were published, coauthored with Afek and Shavit, in OPODIS'11 [3].

3.1 The COP Template

The COP algorithms work with any HTM and STM implementation, but the actual TM realizations have their own limitations and characteristics that demand specific tailoring. The template in this section is for an ideal TM block which has a suspended mode as defined in Section 4.2, and where every transaction eventually succeeds.
CHAPTER 3. THE COP METHODOLOGY

General COP Template for Function $\kappa$

```
start_transaction;
ANY CODE;
suspend_transaction;
$\kappa$ROPOutput ← $\kappa$ROP();
resume_transaction;
if $\neg$($\kappa$Verify($\kappa$ROPOutput)) then
    abort_transaction;
$\kappa$Complete($\kappa$ROPOutput);
ANY CODE;
end_transaction;
```

Figure 3.1: Generic COP template

3.1.1 Operation Structure

Let $\kappa$ (kappa) be a function, which is a sequential operation on a data structure. $\kappa$ can be written as a sequential function, as $\kappa$Complete($\kappa$ROP()), where $\kappa$ROP() is the read-only prefix of $\kappa$ and it generates $\kappa$ROPOutput.

The template for a COP version of $\kappa$ is given in Figure 3.1. In Chapter 4, Figure 4.1 shows this template ported to GCC STM interface, and, in Chapter 5, Figure 5.1 is the porting to Haswell HTM block.

To adapt $\kappa$ to COP, we extract a read-only prefix of it into $\kappa$ROP() (line 10). $\kappa$ROP() calculates $\kappa$ROPOutput, in an unsafe mode, i.e., without any synchronization, even though it resides in a transaction. Thus $\kappa$ROPOutput might be inconsistent and wrong, due to conflicts with a concurrent transactions.

After calculating $\kappa$ROPOutput, we resume the transaction in line 11, and call $\kappa$Verify($\kappa$ROPOutput) in line 13. If $\kappa$Verify sees $\kappa$ROPOutput is inconsistent, it will abort and retry the transaction. If $\kappa$Output is consistent, the transaction continues to execute $\kappa$Complete($\kappa$ROPOutput). $\kappa$Complete($\kappa$ROPOutput) will use $\kappa$ROPOutput and performs any updates, assuming that $\kappa$ROPOutput is correct.

If the transaction aborts, due to explicit `abort_transaction` or because of a conflict, it will automatically retry, and, if there are too many retries, the TM mechanism must execute it solo in order to verify progress, as if it were any other transaction, i.e., transactions that do not include any COP operations.
3.1.2 Correctness Proof Method

A correct COP version of $\kappa$ requires that the underlying TM and the the $\kappa$ROP() will not produce arbitrary executions:

**Property 1. Transactional Regular Registers:** transactional locations are regular, in the sense of regular-registers [46], i.e., if a thread reads a location $L$ in non-transactional context concurrently with a transaction $T$, which writes $V$ to $L$, it will read from $L$, either $V$, or the value that was in $L$ when $T$ started, but not an arbitrary value.

All variables, parameters and return value of $\kappa$ROP() are transactional regular registers.

Transactional regular registers are safety related, in the sense that the ROP can not read arbitrary values, thus, it is possible to reason about its output. In addition, if the COP version of $\kappa$ demonstrates the following properties, it is correct and will not deadlock.

**Property 2. Obliviousness:** $\kappa$ROP() must maintain that: It completes without faults, regardless of concurrent executions, and will finish in a finite number of steps if executes alone.

Obliviousness is progress related, as if $\kappa$ROP() will crash or get stuck in an infinite loop, no work will be done. The following two properties imply the correctness of the COP operation.

**Property 3. Verifiability:** $\kappa$ROPOutput has attributes, that can be tested locally, and that imply $\kappa$ROPOutput is consistent, and $\kappa$Verify is checking these attributes.

**Property 4. Separation:** $\kappa$Complete is using $\kappa$ROPOutput but is not aware of any other data collected by $\kappa$ROP().

Verifiability imply that the consistency of $\kappa$ROPOutput can be checked locally, by looking at its attributes. This may require adding to the sequential $\kappa$ code, without changing its functionality. As the $\kappa$Verify and $\kappa$Complete are in the same transaction, we know that $\kappa$ROPOutput stays consistent until commit, and as $\kappa$Complete executes in a transaction, and according to Separation, $\kappa$Complete accesses only consistent data, thus, we have a serializable, COP version of $\kappa$. 
The system model here is a global lock, i.e., a code segment that executes in a transaction that is semantically protected by a global lock, and will 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 \( \phi_{\text{ROP}} \), \( \phi_{\text{Verify}} \) and \( \phi_{\text{Complete}} \). If, for example, we want to demonstrate a COP implementation of an RB-Tree Insert function, we will present ROP, InsertVerify and InsertComplete. After creating the COP version, we have to show that it has the three properties described above.

### 3.2 A COP Based RB-Tree

The canonic example we use for a COP data structure is the RB-Tree with chained leaves that was introduced in [3]. After fitting the algorithm in the generic COP template, it is adjusted to GCC compiler in Chapter 4, and adapted to Haswell HTM block in Chapter 5.

In Figure 3.2, 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...
3.2. A COP BASED RB-TREE

<table>
<thead>
<tr>
<th>RB-Tree insert operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 start_transaction;</td>
</tr>
<tr>
<td>18 ANY CODE;</td>
</tr>
<tr>
<td>19 suspend_transaction;</td>
</tr>
<tr>
<td>20 InsertROPOutput ← InsertROP(RB-Tree, Key);</td>
</tr>
<tr>
<td>21 resume_transaction;</td>
</tr>
<tr>
<td>22 if ¬(InsertVerify(InsertROPOutput)) then</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>24 InsertComplete(RB-Tree, InsertROPOutput, Key, Value);</td>
</tr>
<tr>
<td>25 ANY CODE;</td>
</tr>
<tr>
<td>26 end_transaction;</td>
</tr>
</tbody>
</table>

Figure 3.3: RB-Tree insert operation Fit in generic COP template

context, continues and reaches the leaf that holds 26. By contrast, a plain TM search, 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 its 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 that it got a valid result. Note that when TM failed, it lost the whole transaction, not just the search for 26 operation. If, for example, 26 was a product of a heavy prior operation, that operation will be lost as well, while the transaction that used a COP operation continues.

3.2.1 Algorithm

The $k$, i.e., the function that is fit into the COP template in Figure 3.3, is Insert(RB-Tree T, Key k, Value v). The Insert is split to The InsertROP() that returns InsertROPOutput, InsertVerify(InsertROPOutput), which verifies the output of InsertROP(), and InsertComplete(InsertROPOutput) which is performing the updates.

The ROP part of the insertion algorithm, shown in Figure 3.4, looks for a key $K$ and returns a node $N$. If K is found, $N$ holds $K$. Otherwise, $N$ is a leaf that either is the potential predecessor or successor of $K$. The keys $\infty$ and $-\infty$ always are in the tree, so $N$ can not be null.

Because the tree has the sentinel nodes, there is no need to check that predecessor, successor or $p$ are not null in InsertVerify from Figure 3.5. The special case of the
RB-Tree insert ROP

input : RB-Tree T, Key k
output: *node pp

27 *node p;
28 p := T.root;
29 while p \neq \text{null} do
30 pp := p; if p.k = k then
31 \quad \text{return } p
32 \quad \text{if } p.k > k \text{ then}
33 \quad \quad p := p.left
34 \quad \text{else}
35 \quad \quad p := p.right
36 \quad \text{end}
37 \text{return } pp;

Figure 3.4: RB-Tree insert ROP in generic COP template

RB-Tree insert verify

input : *node p, Key k
output: boolean Valid

39 if p.live = false then
40 \quad \text{return false}
41 \quad \text{if } p.k \neq k \text{ then}
42 \quad \quad \text{if } p.left \neq \text{null then}
43 \quad \quad \quad \text{return false}
44 \quad \quad \text{if } p.prev.k \geq k \text{ then}
45 \quad \quad \quad \text{return false}
46 \quad \quad \text{else}
47 \quad \quad \quad \text{if } p.right \neq \text{null then}
48 \quad \quad \quad \quad \text{return false}
49 \quad \quad \quad \text{if } p.succ.k \geq k \text{ then}
50 \quad \quad \quad \quad \text{return false}
51 \quad \quad \text{end}
52 \text{return true;}

Figure 3.5: RB-Tree insert ROP verification in generic COP template
insertion of the two sentinels needs to be performed in a non-COP manner, so as to allow this optimization.

The InsertComplete function code is not shown here, as it is the same as the insert function from [25]. This underlines the fact that the exact same COP algorithm is also used for Delete and Lookup functions.

### 3.2.2 Correctness

After the RB-Tree algorithm was set in the COP template, it is left to show that it has the COP correctness properties, which would imply that it is serializable. It is assumed there is some safe memory reclamation, which ensures a node is not recycled until all tasks that access it terminate. This can be achieved using methods from [38].

In addition, when a node is recycled, its left and right pointers are set to `null`, so there are no cycles in the garbage nodes. Now we can prove the following:

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

**Proof.** When the `InsertROP` reaches a node N, it traverses its right or left pointer. N is either in the tree, or was in the tree and got removed. If the node was removed, its pointers were set to `null`, so the ROP will stop. If N is in the tree, then there is a finite path from N to a leaf, and the leaf pointer will be `null`, and `InsertROP` will stop. According to Property 1, if a tree update is concurrent to the `InsertROP`, it will still see either a `null` or a valid pointer. \(\square\)

**Lemma 2.** *InsertROPOutput, returned by InsertROP (Figure 3.4, and InsertVerify from Figure 3.5 have the Verifiability property.*

For the proof of Lemma 2, N is the node returned from InsertROP, i.e., InsertROPOutput and T is the RB-Tree.

InsertVerify must determine if N is in the tree, and if it holds the key that was searched, or if the key is not in the tree and if N holds its potential successor or predecessor in T. As the InsertVerify executes in the context of a transaction, it sees T updates as atomic operation. For example, if a node live mark is `false`, the node already is not in the tree.

**Proof.** If N is live and holds K, it is part of T and has the correct key.
If N is live and holds key \( K_P \), and N points to successor S which holds key \( K_S \), and \( K_S > K > K_P \), 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 \( N\rightarrow\text{right} \) is \textbf{null}, K’s node can be connected to it as N right son. If \( N\rightarrow\text{right} \) is not \textbf{null}, the InsertROP from Figure 3.4 would have traversed that node, so N is not consistent. In the case we present 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 \textit{Separation} property.

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

In the same way, with trivial modifications, we can show that the delete and the lookup have the above-mentioned properties.

As we have proven, all COP RB-Tree functions have the \textbf{Obliviousness}, \textbf{Verifiability} and \textbf{Separation} properties; in conclusion, we have shown the following:

**Theorem 4.** The COP RB-Tree is serializable.
Chapter 4

STM with COP

In consistency oblivious programming (COP), the read-only prefix of a data structure operation is performed outside of the TM transaction. Then the operation is completed using a transaction by verifying the prefix output and performing updates. In STM, this strategy effectively avoids much of the overhead and potential contention.

We emphasize the importance of transaction-suspension which enables performing non-transactional memory accesses inside a transaction. Suspension not only simplifies the use of COP, but also enables the composition of a sequence of COP-based operations into a single transaction. We add transaction-suspension support to GCC STM, and integrate COP into real applications. Transactions with COP operations, can use TM-Safe memory reclamation, because we add privatization before rollback in the GCC STM library.

Our evaluation shows that using COP data structures can greatly improve performance. We also add statistics counters to the GCC STM that show that using COP reduces the number of aborts and the number transactional memory read accesses. We compare the performance of COP-based transactional applications with applications that use Intel’s Haswell HTM, which cannot support transaction-suspension. We show that in some applications COP-based transactions yield better performance than HTM.

The results in this chapter were published, coauthored with Shavit and Suissa, in PODC’13 [8] and some later results are submitted to Europar’14.
4.1 Introduction

Software transactional memory [56] aims to parallelize sequential applications, and make them scalable and concurrent. The struggle to make STM practical and useful, proved that STM must be supported by the compiler [54], otherwise developers will consider it too complicated. In this paper, we introduce a methodology that uses GCC STM to support efficient and natural composition of COP operations in any application. This allows composable, efficient and scalable use of GCC STM, and, in some applications, make it comparable or even better than HTM.

In GCC, the transaction code is marked by the `__transaction_atomic{}` directive. Inside the transaction, all accesses to shared memory are instrumented, which means that instead of plain load or store machine instruction, a library function is called with the corresponding address (and a value upon a write access). The function can lock the location for writing, log it for later verification or rollback, or verify the time stamp version of the address, according to the STM algorithm implemented by the library.

GCC-TM supports various STM implementations, each having its own characteristics. An STM can either be a write-through or a write-back implementation. In write-through, the STM writes the newly stored value directly to the destination address in memory. In write-back, by contrast, the tentative value is kept in a redo log until commit time, only visible locally to future operations in the same transaction.

The simplest STM implementation is serial, in which a global lock is acquired and released when starting and committing a transaction, respectively. More sophisticated implementations are based on multiple-locks, where each logical lock is associated with a distinct set of addresses. When accessing some address, the STM can verify whether that address is locked or not. When writing to an address, the multiple-locks STM can either attempt to lock it (if it is not already locked) and release it during the commit (encounter-time locking), or abort or log the address for locking in commit-time.

Previous STM research [28, 29] showed that the most efficient and scalable STM is based on multiple-locks and uses write-through and encounter-time-locking semantics, and therefore is the default algorithm in GCC-TM library. Note that a write-through implementation must use encounter-time locking to maintain isolation.

During compilation, the compiler infers what addresses in a transaction may be accessed by concurrent transactions, and replaces the access with a call to a store or
load function supplied by the STM implementation.

In GCC, the transaction code can call functions that are attributed by either \_\texttt{transaction\_safe} or \_\texttt{transaction\_pure} \cite{54}. \_\texttt{transaction\_safe} functions are considered a part of the transaction, but the \_\texttt{transaction\_pure} functions never are instrumented.

In GCC-STM the programmer has to make the decision whether some code segment does not require instrumentation. As STM, unlike TM in hardware, is not aware of any non-transactional access to shared memory locations, it is up to the developer to prevent such races.

A transaction can also call \_\texttt{transaction\_cancel}, which will abort the transaction, i.e., discard and undo all this transaction updates, as if it had never been executed. The STM algorithm is best effort, in the sense that if, after a certain number of attempts, a transaction repeatedly is aborted, it moves to a serial execution mode in which it takes a global lock and pessimistically executes the code.

Section 4.2 outlines the resemblance of TM-Pure and the POWER suspended mode. In Section 4.3 the suspended mode is used to generate a COP template, and, in Section 4.4, it is used to compose COP operations. Section 4.5, demonstrates composing COP operations, and Section 4.6 explains how a COP library can be used in an application. We provide performance evaluation in Section 4.7.

4.2 TM-Pure and Suspended Semantics

When working on adding suspended mode to GCC STM, we found that the undocumented TM-Pure functions, in the default \texttt{write-through} mode, have similar semantics to future POWER architecture HTM block \cite{17} suspended state, which is marked by the newly introduced instructions \texttt{tsuspend} and \texttt{tresume}.

The following gives a short description of the semantics of the new instructions, followed by their correlation to the TM-Pure code of GCC-TM.

**Semantic 1.** Until failure occurs, load instructions that access memory locations that were transactionally written by the same thread will return the transactionally written data.

As the address was transactionally written by the STM, it already is locked
(encounter-time locking), and the value is in the memory address (write-through). This implies that a load instruction from an address that had previously been written by the transaction, will return the transactionally written value.

**Semantic 2.** *In the event of transaction failure, failure recording is performed, but failure handling is deferred until transactional execution is resumed.*

In case an STM transaction is aborted by another transaction during the TM-Pure function execution, it will detect the abort when it returns from the TM-Pure code.

**Semantic 3.** *The initiation of a new transaction is prevented.*

Initiation of a new transaction in TM-Pure code segment is allowed, although it can be prevented in compile time. We rely on the developer not to misuse this option.

**Semantic 4.** *Store instructions that access memory locations that have been accessed transactionally (due to load or store) by the same thread will cause the transaction to fail.*

We can support this requirement by instrumenting only stores in TM-Pure code, but we instead rely on the developer.

The developer we trust is the one which writes the shared data structures library, i.e., an experienced programmer, who is aware of all kinds of concurrency bugs. Still, COP is a relatively simple way to achieve efficient and composable modules, and memory reclamation issues are resolved.

### 4.3 STM COP Template

In Figure 4.1 we show a template for creating a COP version of the general serial operation $\kappa$. In line 60, if the verification of the ROP output fails, we simply retry it in transactional mode.

If we did not have the suspended mode, we would have to abort the transaction to go back to non transactional mode and get the benefit from COP. In addition, if we had spurious aborts, like the case with HTM, and as a further optimization, we may retry the lines 56 to 59 a few times.

In fact, with current GCC, if we used abort explicit abort in line 60, then a transaction that uses the COP $\kappa$ would not be able to use explicit aborts for any other
4.4 Suspended Transactions and COP Composing

A COP operation (cop), using any TM implementation, performs the following steps:

1. Execute the read-only prefix (ROP) of cop and record its output. This part is done without any synchronization, and may pass through inconsistent states and return inconsistent output. It is the developer’s responsibility to check No Crash, or else a segmentation fault or floating exception, or even an infinite loop, could occur.

2. Start or resume the transaction T.

3. Verify that ROP output is consistent, and if it is not, retry the ROP, now, in transactional mode.

4. Complete the cop updates. In this stage we simply perform the writing part of the transaction, as if it were not in COP framework.

5. Commit the transaction T.

The verification in step 3 should be done locally and must be concise, so that it does not introduce more overhead. Rerunning the ROP in transactional mode, for example, might not be a good verification method.
Another point about step 3, is that ideally, we would retry the ROP several times in non-transactional mode, and if failure persists, abort and execute the whole transaction in *serial* mode. However, in GCC, the user initiated abort has no parameter, so using it in COP would not allow an application to use it for other purposes. This is the reason that, if verification fails, we execute the ROP in transactional mode. In this mode, either the ROP succeeds, or the transaction aborts.

In transactional mode, there is no performance benefit from the ROP, because, as the ROP will execute in the context of T, all accesses will be instrumented by the TM. The only way to compose COP operation, when suspend is not available, without instrumenting ROP, is the one proposed by [59], i.e., execute all ROP parts of the composed operations before starting the transaction, then, inside the transaction, verify them and complete updates. This method allows composition only if an operation is not writing data that may later be accessed by another operation in the same transaction.

To demonstrate this restriction, we divide each operation $op^k$ to to $op^k_{ROP}$ and $op^k_{VC}$ (verify and complete). Now, assume $op^1$ precedes $op^2$, and $op^1$ is writing data that $op^2$ is reading. In [59] the transaction T, which executes $op^1$ and then $op^2$, will execute in the following order:

$$op^1_{ROP} \rightarrow op^2_{ROP} \rightarrow T_{START} \rightarrow op^1_{VC} \rightarrow op^2_{VC} \rightarrow T_{END}$$

As $op^2_{ROP}$ must execute before $op^1_{VC}$, $op^1$ will not see $op^2$ updates, so T can not be correct. It is not trivial to verify the independence of transactions, and grouping the operations parts together complicates the code.

Even if $op^1$ is not COP, but precedes $op^2$ in a transaction T, we must start T before $op^1$, so $op^2_{ROP}$ will again be instrumented. If T dequeues V and then inserts V to a RB-Tree with a COP operation, for example, then this COP operation’s ROP will be instrumented, and this common scenario will not benefit from the usage of COP.

Using TM-Pure as suspended mode will allow the composing of any COP operation, with any other COP and non COP operations, by breaking down the COP operation to the following steps:

1. Start the transaction T.
4.4. SUSPENDED TRANSACTIONS AND COP COMPOSING

2. Optional: Execute any code.

3. Execute a TM-Pure function to execute the ROP of cop and record its output.
   This part is done without any synchronization, and may pass through inconsistent
   states. It is the developer’s responsibility to verify that this code does not crash.

4. Execute the verification of the ROP output and any updates.

5. Optional: Execute any code.

6. Commit the transaction T.

Now T will execute ROP as a TM-Pure function so we call the ROP of op
op\text{ROP-\text{PURE}}. If T tries to execute the COP operation op^2 after the COP operation
op^1, it will pass the following steps:

\[ T_{\text{START}} \rightarrow \text{op}^1_{\text{ROP-\text{PURE}}} \rightarrow \text{op}^1_{\text{VC}} \rightarrow \text{op}^2_{\text{ROP-\text{PURE}}} \rightarrow \text{op}^2_{\text{VC}} \rightarrow T_{\text{END}} \]

As \text{op}^1_{\text{VC}} executes before \text{op}^2_{\text{ROP}}, and as both \text{op}^1_{\text{VC}} and \text{op}^2_{\text{ROP}} execute in the
context of T, according to Semantic 1, \text{op}^2_{\text{ROP}}, which executes after \text{op}^1_{\text{VC}} performed
its updates in the context of T, can see these updates and T is correct.

4.4.1 Safety

To allow a ROP to complete without fault and generate verifiable output, we have
to show that a load access from address A, in the non-transactional ROP, reads only
values that were intentionally written to A. This is true, because when the GCC STM
library is unrolling a write-through transaction, it uses \texttt{builtin\_memcpy()}. This library
function is optimized, so that it copies the largest alignment it can. As a basic data
type is not wider than the largest machine access, and is always aligned to its size by
the compiler, it is both written by TM store and unrolled by one atomic access. Thus,
the TM-Pure code is seeing values that were intentionally written, and were not some
mix of old and new values.

4.4.2 Memory Reclamation

Two important functions that are TM-Safe \cite{54}, i.e., can execute within transactions,
are \texttt{malloc} and \texttt{free}. These functions are made safe by privatization. If a transaction T
wrote to memory, than before it returns, it waits for all the transactions that started before its commit to finish [23]. As a side effect of privatization, in the case T detached a node from a data structure, it can free it.

On the other hand, if T allocated a node N, and then aborted, it supposedly can free N without privatization. The reason is that the pointer to the tentative node is not exposed to other transactions. This is not true when COP is involved. If the non-transactional ROP traverses the data structure, it may acquire a pointer to N, and then when T aborts and automatically free N, the ROP may try to access unmapped memory. To prevent this scenario, we added privatization also to writing transactions that are going to rollback. If the transaction didn’t write, it can free its tentative nodes unconditionally, but if it wrote, it has to perform privatization as if it committed. When comparing performance, this addition did not have a visible impact.

With the suspended mode, and rollback privatization, malloc and free become also COP safe. The reason is that memory is not recycled as long as there is a transaction in progress, and the COP operations are always encapsulated in transactions. One restriction is that allocation can not take place in a ROP, because, in case the validation would fail, the allocated node will not be freed, as we do not abort the transaction in this case. However, as the ROP is not writing memory, there is no reason it would need to allocate or free memory.

4.5 STM COP RB-Tree

We ported the RB-Tree from [3] to the STM template. Listing 4.1, shows the lookup function, in C code, to show clearly the usage of compiler features and avoid ambiguities. If K is not found, the function returns a leaf that is the potential successor or predecessor of K, otherwise it returns the node of k. Once the transaction got a node from the TM-Pure lookup, it goes back to transactional mode to verify that it got a valid node, and afterwards, to complete updates if necessary. The verification, is attributed transaction_safe, so that it is part of the transaction, and all shared accesses in it are instrumented. The parameters for the _insert function are the global rb_tree, a key-value pair to insert, and a freshly allocated node. All of these either are constant or local, so they are consistent. In addition, _insert uses the output of the TM-Pure ROP (the _lookup). This output is verified, and the verification criteria are in the transaction
4.6 Integrating COP and Applications

The applications in STAMP TM benchmark suite [51], use a library of transactional data structures, which include a queue, a heap, a tree, a linked-list and a hash table. The queue and the heap operations have very short ROP, so they can not currently benefit from COP. The linked list is used mostly as a hash-table bucket, and through iterators, and the ROP of its operations in these usage patterns is very short. Thus, considering the verification phase, the COP linked-list is not changing the workloads performance. To see the impact of COP we needed an application which uses RB-Tree heavily, and the only candidate in STAMP was the Vacation application. The yada and intruder tests also use the RB-Tree, but not as much as vacation. The COP applications we present gain the COP performance, with no effort, as the COP RB-Tree is seamlessly replacing its plain STM counterpart.

```c
node_t * __attribute__((transaction_pure)) lookup_pure (tree_t * s, int k)
{
    unsigned long status;
    node_t * p, *pp = 0;
    p = s->root;
    int cmp = 1;
    while (p != null)
    {
        // k is an integer key
        cmp = k - p->k;
        if (cmp == 0)
        {
            return p ;
        }
        pp = p;
        p = (cmp < 0) ? p->l: p->r;
    }
    return pp;
}
```

Listing 4.1: RB-Tree COP Lookup (ROP)

read-set, so the transaction will abort if they are violated. Thus, the insert only uses consistent input, so it will produce consistent results, or it will abort.
CHAPTER 4. STM WITH COP

4.7 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 and the chip comprises a shared 8MB level 3 cache. The cache lines are all 64-bytes.

All code was written in C and compiled with GCC-4.8.1.

We first evaluate COP contribution to real applications, and then devise a set of targeted micro benchmarks to understand the details of COP performance.

4.7.1 COP in Applications

To examine the potential contribution of COP to applications, we added our COP RB-Tree to the STAMP testing suite. Inside the STAMP, we executed Yada, Intruder, and Vacation tests. These tests are the ones that use a map, which is implemented as an RB-Tree.

We execute the standard configuration of Vacation (vacation-high from [51]). Each transaction in this application is accessing several 1M RB trees, several times each, and these transactions are a significant portion of the workloads. In Section 4.7.2, we will eliminate the randomness and see more precisely the potential of COP.

In Figure 4.3 we count transactional loads and aborts for the Vacation benchmark. We count the transactional loads when the whole application is executing on a single thread, to get the most accurate number. The aborts count is taken when all eight hardware threads execute, to get the highest number for the benchmark. We see that plain STM is performing more than five times the transactional loads of COP, and that this is the source of COP better performance. There are also six times more aborts in plain STM, but, still, plain STM has only 3% aborts, which can not hurt performance that much. From the Vacation graph in Figure 4.2a, and from the table in Figure 4.3 we can derive the following insights:

- HTM has a problem with application size transactions. This can be mitigated by adding retries, and never retrying on capacity aborts, which is done in the Section 4.7.2. Here we use the default number of retries from GCC, and let GCC
Figure 4.2: COP RB-Tree in STAMP and in various configurations

<table>
<thead>
<tr>
<th>Transactional Loads</th>
<th>GCC-COP</th>
<th>GCC-STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4G</td>
<td>0.5%</td>
<td>3.0%</td>
</tr>
<tr>
<td>2.4G</td>
<td>0.5%</td>
<td>3.0%</td>
</tr>
</tbody>
</table>

Figure 4.3: STAMP Vacation Statistics
decide if the HTM transaction should be retried.

- The difference from STM to COP stays constant, starting at one thread, which means it is the overhead saved by the COP. We can see the scalability of the workload is the same for COP and plain GCC STM, which suggests that contention, which is higher in plain STM, is not the factor here.

The graph of Intruder in Figure 4.2b also shows that COP manages to reduce the overhead of the RB-Tree, although the total benefit from COP there is minor, as the RB-Tree operations are less dominant in the workload.

### 4.7.2 STM with COP vs. Plain TM

To better understand the performance of COP, we tested our COP RB-Tree as a standalone. The RB-Tree has two modifying transactions, insert + lookups and delete + lookups, and one read-only operation, which only holds lookups. In the titles of the graphs, U is the percentage of updating operations, T is the number of threads and ops is the number of operations per transaction. 1K or 1M is the range of keys in the tree, which starts half full.

The y axis of all graphs counts successful operations, not transactions. If a transaction executes X operations, it adds X to the operations count. This way the scalability to higher number of operations per transaction can be visualized.

We compare plain STM transactions, plain HTM transactions and STM transactions with COP. All HTM transactions are retried up to 10 times to get their best performance; if, however, the HTM transaction hits a capacity abort, we assume that it cannot pass in HTM, and it will directly fall back to global lock.

We present three couples of graphs, to check COP behavior relative to number of threads (Figure 4.2c and Figure 4.2d), update rates (Figure 4.2e and Figure 4.2f) and transaction sizes (Figure 4.2g and Figure 4.2h).

The graphs include the following lines:

1. **s-op**: Plain STM operations per second.

2. **s-con**: Plain STM aborts, as counted in GCC.

3. **c-op**: COP operations per second.

4. **c-con**: COP aborts, as counted in GCC.
4.7. EVALUATION

<table>
<thead>
<tr>
<th>Tree Size</th>
<th>GCC-COP</th>
<th>GCC-STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small (1K)</td>
<td>50M</td>
<td>88M</td>
</tr>
<tr>
<td>Large (1M)</td>
<td>36M</td>
<td>120M</td>
</tr>
</tbody>
</table>

Figure 4.4: RB-Tree transactional loads

5. **h-op**: Plain HTM operations per second.

6. **h-con**: Plain HTM conflict aborts.

7. **h-cap**: Plain HTM capacity aborts.

We show the HTM data, only in Figure 4.2g and Figure 4.2h where we vary the transaction sizes that run on four threads. The reason is that when using hyperthreading, HTM behavior is harder to analyze. For example, it counts capacity aborts as conflicts, and we do not want to confuse the reader.

We count operations, and not transactions. Each conflict abort is counted as half of the operations in the transaction, assuming, on the average, that the transaction conflict happened in the middle of it, and we count each HTM capacity abort as the number of operations in the transaction, because it would mean the transaction will execute, in serial mode, that same number of operations. It also means the number of capacity aborts is bounded by the number of successful operations. If the **h-cap** line is on the **h-op** line it means that all HTM transactions were capacity aborted.

In all our benchmarks, STM performance is below COP. The reasons are both the instrumentation overhead, as seen in Figure 4.4, and the higher conflicts rate of the plain STM. In the smaller tree, the difference in the instrumentation amount is smaller, as the ROP is shorter. Thus, the lower aborts rate of COP determines its better performance. In the larger trees, there is less contention, but the difference in instrumentation amount is larger, which, again, makes COP better.

COP is scaling better then HTM and STM in all parameters, i.e., when contention or size of the transactions rise, COP performance, compared to plain TM, is improving. Hyperthreading, which occurs when there are more than four threads, cuts HTM capacity in half, and triggers much more HTM aborts, giving COP a big advantage. However, this is a known problem of Haswell HTM. Thus, we compare COP with STM to HTM, only without hyperthreading. In Figure 4.2h, we can see COP performance is better than HTM, due to HTM capacity aborts, also with no hyperthreading, when there are more than eight operations on a large tree per transaction. As demonstrated in the STAMP Vacation (Figure 4.2a), this workload is used in applications.
In Figure 4.2g, HTM is better than COP for all checked transaction sizes, as this is a small tree, and there is no capacity problem. Yet, we can see that as the transactions grow, COP maintains scalability, while HTM performance degrades, due to a higher number of conflict aborts.

Figure 4.2c and Figure 4.2d vary the number of threads. In the 1K, smaller tree, the plain STM shows higher aborts rate than COP, which makes COP scale better, while in the larger tree, the difference in instrumentation overhead, is the COP advantage. The same analysis is true for Figure 4.2e and Figure 4.2f which vary the updates rate on small and large trees. On the small tree, as contention is higher, COP performs better due to lower conflicts rate, while, on the large tree, it wins because of less instrumentation overhead.

4.8 Summary

We discover suspended mode in GCC STM, which enables the use of COP operations to boost the performance in any place in a transaction that currently uses a non-COP version of these operations.

With suspended mode and COP, STM performance on shared data structures is improved significantly in a wide range of workloads. This fact suggests that libraries of COP data structures may allow applications to enjoy the simplicity of TM, while significantly improving performance.

To simplify programming with COP further, we added privatization upon rollback of writing transactions, which allows the usage the TM-Safe malloc and free.
Chapter 5

HTM with COP

Consistency-oblivious programming mitigates, and, often solves, a major problem of the current and near future HTM blocks. The problem is that the addresses set, monitored by a transaction, remains inaccessible to software. Thus, an address that was logged by the hardware can cause conflicts if it already is irrelevant, and, as the capacity for addresses in a hardware transaction is limited, the irrelevant addresses increase, the chance that a transaction will violate capacity and abort.

In a plain transactional style, an operation, such as an insertion, is enclosed within a hardware transaction. In the COP-style, by contrast, there are two phases: an oblivious phase that executes with no transactions or locking, and an atomic phase that verifies that the output of the oblivious phase is correct and performs updates. If the verification fails, the oblivious phase must be retried, but if we encounter a conflict with another transaction, only the second phase is re-executed, as the non transactional ROP cannot cause a conflict.

This chapter introduces a template for HTM and COP data structure operation algorithms, and COP versions for two data structures: an RB-Tree with chained leafs, and a dynamic cache-oblivious B-tree.

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. The basic advantage of COP is that it keeps scaling, while naive HTM stops, due to capacity or conflict.

The results in this chapter appeared, coauthored with Kuszmaul, in TRANSACT’14 [7].
5.1 Improving HTM Scaling with COP

Our version of COP is similar to STM-based COP, found in Chapter 4. The major advantage of COP in HTM is that it reduces the footprint of a transaction, which can determine whether or not an application must execute a fallback code, such as obtaining a global lock. As with STM, reducing the footprint of a transaction can reduce the probability that two separate transactions conflict. Sometimes the prefix can execute and produce an acceptable result, even though the part of the data structure examined by the prefix was not technically consistent. Later in the transaction, if the addresses accessed in the ROP are written transactionally or not, they will not abort the HTM transaction either.

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. It 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 provide limited associativity, programmers may be surprised to find 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 a naive HTM version can only handle a much smaller number of operations.

The rest of this chapter is organized as follows: Section 5.2 describes how we adapted the COP idea to an HTM context. Section 5.3 explains our concurrent RB-Tree implemented with COP. Section 5.4 explains our COP-based, cache-oblivious B-tree. Section 5.5 presents performance measurements and compares the various schemes. We set forth our conclusion in Section 5.6.

5.2 HTM COP Template

In this paper, we used COP with the Intel Haswell RTM, and we 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 that interest us, in the context of COP, are in the following table:
Let \( \kappa \) (kappa) be a function, which is a sequential operation on a data structure. The template for a COP version of \( \kappa \), using the GCC-4.8 HTM intrinsics, is given in Figure 5.1.

To adapt \( \kappa \) to COP, we extract the longest read-only prefix of it into \( \kappa ROP() \) (line 65). \( \kappa ROP() \) calculates \( \kappa ROPOutput \), in an unsafe mode, i.e., without any synchronization. Thus \( \kappa ROPOutput \) might be inconsistent and wrong, due to conflicts.
with concurrent operations.

After calculating $\kappa$ROPOutput, we start a transaction in line 66, and call $\kappa$Verify($\kappa$ROPOutput) in line 68. $\kappa$Verify will call _xabort if $\kappa$ROPOutput is inconsistent. If $\kappa$ROPOutput is consistent, we will continue the transaction to execute $\kappa$Complete($\kappa$ROPOutput). $\kappa$Complete($\kappa$ROPOutput) will use $\kappa$ROPOutput and perform any updates, considering that $\kappa$ROPOutput is correct.

Before trying to commit in line 72, we check in line 71 that the global lock is free. If it is 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 fallback, because a conflict is considered a retry, while a lock, as seen in line 75, allows us to reuse the ROP output, and is not considered a retry.

During our transaction, if the lock were taken by a concurrent transaction, it would not present a correctness or performance problem. If a transaction $T_1$ saw a partial effect of the serial mode transaction $T_2$, then it saw an address $A_1$ that $T_2$ wrote and an address $A_1$ that $T_2$ did not yet write, and that it subsequently wrote. As the lock is free, $T_2$ already wrote $A_1$, which means that $T_1$ aborted. Thus, it is not a correctness problem. It is easy to see that waiting for the lock to be free would not have improved performance, as is also explained in [20].

If the transaction failed, and we want to retry, we will reach line 78. If the source of the abort were a capacity overflow, we would not retry the transaction, because it probably will fail again; instead, we lock and execute the sequential version. If it had been an explicit abort, i.e., $\kappa$Verify called _xabort, we must rerun $\kappa$ROP to get a correct $\kappa$ROPOutput, otherwise, the abort must have been due to a conflict, so $\kappa$ROPOutput may well be correct, and the transaction has a chance to commit successfully, thus, we reuse $\kappa$ROPOutput and retry the HTM transaction. If we have no more retries, we lock and execute $\kappa$ sequential version.

### 5.3 HTM COP RB-Tree

We ported the COP RB-Tree with chained leaves from [3] to our COP template. Listing 5.1 shows the code for inserting into a RB-Tree, using C notation to make the exposition closer to the real code.

The algorithm for insertion, which was introduced in [3] and proved in Section 3.2.2,
void rb_insert(RBT *s, int K, int V) {
    int retry_count = 0;
    retry:
    node_t *place = ROP(t, K);
    retry_verify:
    while (tree_locked) pause();
    int status = _xbegin();
    if (status == _xBEGIN_STARTED) {
        RBVerify(place, K);
        RbInsertComplete(t, K, V, place);
        if (tree_locked) _xabort(WAS_LOCKED);
        _xend();
    } else {
        if (is_explicit(status, WAS_LOCKED))
            goto retry_verify;
        // Other failures prejudice us.
        // Allow only RETRY_COUNT retries.
        if (retry_count++ < RETRY_COUNT) {
            if (is_explicit(status, BAD_ROP))
                // Must redo the whole prefix
                goto retry;
            if (can_retry(status))
                goto retry_verify;
        } else {
            // Fallback code.
            lock_tree();
            place = ROP(t, K);
            RbInsertComplete(t, K, V, place);
            unlock_tree();
        }
    }
}

Listing 5.1: RBInsertComplete function.

looks for a key $K$ and returns a node $N$. If $K$ is found, $N$ holds $K$. Otherwise $N$ is a leaf, which either is 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. 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 recycling of the nodes, and we keep the nodes within the same tree, so that arbitrary pointers will not lead us to undefined memory that could crash our code or fool it with locations that look like valid nodes but are not. Our RB-Tree implementation recycles nodes within a thread, and if a thread accumulates more than a threshold of idle nodes, it uses an epoch-based memory reclamation [38] scheme to free them.

The verification is doing the same tests as Figure 3.5, but, if a test fails, it calls _xabort to abort immediately.

Returning to Listing 5.1, the code next waits until the tree is not locked (at line 6).
CHAPTER 5. HTM WITH COP

Listing 5.2: RB-Tree COP Lookup (ROP)

The fallback code acquires a mutex on the tree. As we shall see, to make progress, we will require that the lock be not held, so that 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 \( T_1 \) (at line 7). The \_xbegin() function either returns \_XBEGIN_STARTED, in which case it is running in the transaction \( T_1 \), or else the system attempted the transaction and failed; its status tells us something about why it failed.

In the case that \( T_1 \) is running, it must finish the insertion. Since the read-only prefix ran without any synchronization, it could yield an inconsistent result, and \( T_1 \) must verify its correctness (at line 10). The verification code, shown in Listing 5.3, is doing the same checks as Figure 3.5. If the verification fails, it calls \_xabort() to explicitly abort \( T_1 \), with a code indicating that the verification has failed. If the verification succeeds, \( T_1 \) completes the insertion at line 10. Finally, \( T_1 \) checks to see if the tree is locked. If it is locked, then some other code may be modifying the data structure in a way that is inconsistent with our transaction. In this case, \( T_1 \) explicitly abort with a code indicating that the lock was held. It could be that during the \( T_1 \), another transaction, \( T_2 \) locked the tree and released it. As \( T_2 \) released the lock, however, it is not possible that \( T_1 \) saw a partial effect of \( T_2 \) and did not get a conflict abort.

Because the tree has the sentinel nodes, there is no need to check that predecessor and successor pointers are not \texttt{NULL}. When the tree is empty, as, for example, at the first insertion, the verification will fail by following a \texttt{NULL}, and will eventually fallback to the lock and skip the verification. This is acceptable; it will happen twice, once for each sentinel node, as the predecessor will be \texttt{NULL}, and then it never will happen again. Also note that it saves conditions in the \texttt{rb\_rop\_verify}, which is frequently called.

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

```c
33    node_t* ROP(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    }
```
5.4. CACHE-OBLIVIOUS B-TREE

We also tested a dynamic cache-oblivious B-tree (COBT) [11]. 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 [48].

Figure 5.2 shows 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. The COBT contains 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 worst-case cost is also $O(\log_B N)$, but we have not 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 RB-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.
```c
volatile int dummy;
int test (volatile char *A, int stride) {
    for (int txsize=1; txsize++;
     for (int trial=0; trial<20; trial++) {
        int sum=0;
        for (int i=0; i<txsize; i++) {
            sum+=A[i*stride];
            if (_xbegin() == _XBEGIN_STARTED) {
                A[0]++;
                for (int i=0; i<txsize; i++) {
                    sum+=A[i*stride];
                    _xend();
                    dummy=sum;
                    goto next_txsize;
                }
            }
        }
        // 20 trials failed.
        // Return the last txsize that worked.
        return txsize-1;
    }
    next_txsize:;
}
```

Listing 5.4: Code for determining the capacity of a transaction.

Just as for the RB-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.

### 5.5 Evaluation

We use the same machine and compiler as in Section 4.7 and use HTM intrinsics that were introduced in GCC-4.8.

Before we evaluate our algorithms, we want to better understand the behavior of the HTM in practice. We initiated 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 already is in level 3 cache, the system can accommodate very large transactions. If, however, 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 5.4 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, so we declared dummy and A to be volatile. 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. If, for example, 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 5.3 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 5.3 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. 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 $\text{CacheSize}/(\text{CacheLine} \times \text{Stride})$ and 8, which is what Figure 5.3 shows.

In Figure 5.3, 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.

To generate the data in Figure 5.3, we execute the 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

![Figure 5.3: Associativity limits on Haswell HTM](image-url)
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 execute with that stride.

### 5.5.1 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 small footprint. 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 5.1, 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 unsuccessful 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

![Figure 5.4: RB-Tree various transaction sizes](image)

![Figure 5.5: RB-Tree contentious workload](image)
would mean this number of operations now will execute under a global lock.

In Figure 5.4 we see a read-only workload, where the x axis is the number of 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 4 and 8 threads. The tree is initially populated with 100K nodes.

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 transactions on a small tree, capacity aborts seldom occur.

In Figure 5.4a, we execute four threads and in Figure 5.4b eight threads, 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 four threads, naive HTM is scalable up to 16 operations per transaction, while on eight it is scalable only to 8. The reason is that hyperthreading, where each thread from the eight, is sharing the cache with another thread on the same core, so the available capacity for HTM is cut to half.

**Conflicts:** An RB-Tree has low contention, and 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, and inserts a key in the tree. The value field is in the same cache line with the pointers fields and the key fields of the root node, so any concurrent transaction that traversed the root will be aborted. Figure 5.5 counts operations (op for simple HTM and cop-op for COP operations), and conflict aborts (conf, cop-conf). It does not show capacity aborts, because Figure 5.4 shows, that the capacity aborts number for a single operation transactions is negligible. We have a lot of conflicts in the simple HTM, as each updating transaction also is 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 initially is populated with 100K nodes.

In one and two threads, COP has the performance of plain HTM, but then plain HTM stops scaling, while the COP version keeps climbing. The reason is con aborts, which are accumulating from 3 threads for plain HTM, while COP does not encounter any conflicts at all. All the transactions are of a single operation, so capacity aborts are insignificant, as seen in Figure 5.4.

5.5.2 PMA Performance

Figure 5.6 shows read-only operations on a PMA, for COP and plain 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 executed ten times. The error bars show the slowest, the fastest, and the average runtime (through which the curve passes).

The error bars are negligible for all the executions, except in 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 plain HTM code is running with virtually every successful search being performed by the fallback code that is holding the global lock. This means that there essentially are no successful HTM searches in the 8-thread executions. We believe that this poor performance is a result of cache associativity: the array always is a power of two in size, and a binary search on the array repeatedly hits the same cache line. A 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 almost exclusively executes with transactions, rather than with locks.

The plain HTM version usually fails, due to capacity problems when the thread

![Figure 5.6: PMA performance for read-only operations.](image)
count equals one. For larger thread counts, there is 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. The transaction usually failed, because the lock held. When the transaction 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 control runaway locks is to use backoff, but it is not clear how to do this so as to get the system back into an HTM mode. In the case of the plain HTM code, it is not clear that there could be any alternative, since the transactions usually fail, due to capacity. Under COP, the performance achieved is much better, and the verification step typically needs to look at only one cache line.

5.5.3 Cache-Oblivious B-Tree Performance

Figure 5.7a shows the performance of the COBT 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. As we can see, the COP implementation outperforms the plain 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, plain HTM does about the same on average, but has some slow outliers that are about half as fast.

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

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

Figure 5.7b shows for 32 searches per transaction, on a tree containing 100,000 values, how the performance of COP and plain HTM varies with the number of threads. 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. COP dominates HTM, and interestingly HTM has high variance when it works well (sometimes giving very poor performance), The COP, by contrast, shows little variance until the thread count becomes relatively as large as the number of hardware threads. The COP variance at high threads is a good kind of variance; sometimes it executes much faster (attaining near linear speedup), rather than HTM’s variance, which makes it sometimes execute much slower.

5.6 Conclusions

We have shown that COP can make HTM useful in scenarios and data structures where it could not improve upon 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 that COP greatly improves the scalability of composed transactions of RB-Tree. It also allows writing values into 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, whereas, with COP, all the read-only transactions can successfully complete.
Chapter 6

Summary

TM is more than just a new hardware or compiler feature. It can change the way that contemporary programmers synchronize their applications, but when transforming theoretical design into practical application, TM has revealed some built-in pitfalls that render it impractical when naively used in common workloads. Throughout this thesis, we have shown that applying COP methodology enables us to use TM to create concurrent objects that scale and maintain their performance, while the naive TM versions of these objects do not. A naive HTM COBT with 1M nodes could not execute even one search, while its COP version scaled perfectly to 32 searches in a single transaction. The COP version of an RB-Tree with STM proved significantly faster than its naive STM counterpart.

We constructed a simple generic template to facilitate the creation of COP data-structures, and supplied a simple way to check that the COP operations were correct. Then we ported the template to existing HTM and to the compiler STM support. One of the most appealing aspects of TM is software composition, namely, the ability to develop pieces of software independently and compose them into applications that behave correctly in the face of concurrency. With COP, this requires the existence of a suspended mode. In the thesis, we define and examine a suspended mode in compiler STM support, and use it to add COP data structures to a library that is used by the standard STAMP benchmark.
Chapter 7

Future Work

More research, both theoretical and practical, is needed to cultivate the new COP methodology. Some of the necessary work is specified below.

**TM infrastructure and COP.** To optimize COP algorithms, the compiler should provide the developer with a standard interfaces to statistics, such as aborts, as already are available in HTM, and also with transactional accesses counts. Preferably, the accesses count should be per definable segments of code. Once this information is made publicly available, it will be necessary to construct a new measure of complexity for concurrent algorithms, which will factor into consideration the abort rate and the type of the accesses, i.e., transactional or non-transactional. To make COP operations composable in HTM, the hardware must support a suspended mode.

To optimize COP algorithms, the compiler should provide the developer with standard interfaces to statistics, such as aborts, like those that are already available in HTM, and also with transactional access counts. Preferably, the access count should be per definable segments of code. Once this information is made publicly available, it will be necessary to construct a new measure of complexity for concurrent algorithms, which will factor into consideration the abort rate and the type of access, i.e., transactional or non-transactional.

To make COP operations composable in HTM, the hardware must be able to execute non-transactional load inside transactions.

**Data structures and COP.** There are many data structures, such as a skew-heap
or union-find that cannot benefit from TM due to inherent, yet benign, contention. Future research should find ways to fit these data structures into COP so that they can benefit from TM synchronization. There already are COP versions for different trees, linked-list, skip-list and others, but the task remains to fit other graphs and useful structures into COP.

**Applications and COP.** How COP is actually improving real applications also remains to be seen. The B-Tree, for example, is the base of in-memory data-bases, so we should expect that these applications will be able to incorporate a COP version of the B-Tree and thereby gain scalability, compared to the read-write lock per tree solution that essentially is in use. This improvement will allow DB transactions to use TM transactions to compose B-Tree operations. Other COP structures, such as linked-list and RB-Tree, should also expedite application, as already seen in the STAMP benchmark.

**Theory of COP.** At present, each data structure needs to fit individually into the COP template. It will therefore be helpful to devise a way to extract the COP version directly from the code. This would require identifying the longest read-only prefix and its output, and then identifying the verification criteria. It will also be helpful to determine how COP can be extended to code that is unrelated to concurrent data structures and to identify other concurrent function types that will be able to benefit from COP.
Bibliography


[17] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In ISCA, pages 225–236, 2013. 2.3, 4.2


[23] Dave Dice, Alexander Matveev, and Nir Shavit. Implicit privatization using private transactions. In TRANSACT, 2010. 2.4, 4.4.2


[25] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In DISC, pages 194–208, 2006. 2.3, 2.4, 2.5, 53


[47] David Kanter. Intel’s Haswell CPU microarchitecture, 13 November 2012. 2.3


[50] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In SPAA, pages 73–82, 2002. 2.2.2, 2.2.3


[59] Lingxiang Xiang and Michael L. Scott. Composable partitioned transactions. In WTTM, 2013. 2.6, 4.4